

# AI ASSISTED CODING

## LAB-11.2

NAME:M.Suchith

ENROLL.NO:2403A52068

BATCH:04

### **TASK-01:**

Use AI to generate a Stack class with push, pop, peek, and is\_empty methods.

Sample Input Code:

```
class Stack:  
    pass
```

### **PROMPT:**

Generate a python stack class using push ,pop, is empty.

### **CODE:**

```
11.2.1.py x 11.2.2.py 11.2.3.py 11.2.4.py 11.2.5.py 11.2.6.py 11.2.7.py 11.2.8.py 11.2.9.py 11.2.10.py
11.2.1.py > Stack > __init__
1 from typing import Any, List
2
3 class Stack:
4     """
5     A simple implementation of a Stack data structure.
6     """
7     def __init__(self) -> None:
8         """
9         Initializes an empty stack.
10        """
11        self._items: List[Any] = []
12
13    def is_empty(self) -> bool:
14        """
15        Checks if the stack is empty.
16
17        Returns:
18            bool: True if the stack is empty, False otherwise.
19        """
20        return not self._items
21
22    def push(self, item: Any) -> None:
23        """
24        Adds an item to the top of the stack.
25
26        Args:
27            item (Any): The item to be added to the stack.
28        """
29        self._items.append(item)
30
31    def pop(self) -> Any:
32        """
33        Removes and returns the item at the top of the stack.
34
35        Raises:
36            IndexError: If the stack is empty.
37
38        Returns:
39            Any: The item at the top of the stack.
40        """
41        if self.is_empty():
42            raise IndexError("pop from an empty stack")
43        return self._items.pop()
44
45    def peek(self) -> Any:
46        """
47        Returns the item at the top of the stack without removing it.
48
49        """
```

```

    Raises:
        IndexError: If the stack is empty.

    Returns:
        Any: The item at the top of the stack.
    """
    if self.is_empty():
        raise IndexError("peek from an empty stack")
    return self._items[-1]

def __str__(self) -> str:
    """
    Returns a string representation of the stack.
    """
    return str(self._items)

def __len__(self) -> int:
    """
    Returns the number of items in the stack.
    """
    return len(self._items)

# --- Example Usage ---
if __name__ == "__main__":
    print("Creating a new stack...")
    s = Stack()

    print(f"Is the stack empty? {s.is_empty()}") # Expected: True

    print("\nPushing items: 10, 20, 30")
    s.push(10)
    s.push(20)
    s.push(30)

    print(f"Current stack: {s}")
    print(f"Stack size: {len(s)}")
    print(f"Is the stack empty? {s.is_empty()}") # Expected: False

    print(f"\nPeeking at the top item: {s.peek()}") # Expected: 30
    print(f"Current stack after peek: {s}")

    print(f"\nPopping an item: {s.pop()}") # Expected: 30
    print(f"Current stack after pop: {s}")

    print(f"\nPopping another item: {s.pop()}") # Expected: 20
    print(f"Current stack: {s}")

    print("\nPopping the last item...")
    s.pop()
    print(f"Is the stack empty now? {s.is_empty()}") # Expected: True

```

# OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.1.py
Creating a new stack...
Is the stack empty? True

Pushing items: 10, 20, 30
Current stack: [10, 20, 30]
Stack size: 3
Is the stack empty? False

Peeking at the top item: 30
Current stack after peek: [10, 20, 30]

Popping an item: 30
Current stack after pop: [10, 20]
Popping another item: 20
Current stack: [10]

Popping the last item...
Is the stack empty now? True
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

## OBSERVATION:

The AI generated the code in an efficient way according to the prompt as it developed the stack class.

## TASK-02:

Use AI to implement a Queue using Python lists.

Sample Input Code:

```
class Queue:
```

```
pass
```

## PROMPT:

Generate a python code to implement the queue in data structures

## CODE:

```

1  from typing import Any, List
2
3  class Queue:
4      """
5      A simple implementation of a Queue data structure using a Python list.
6      Note: For performance-critical applications, collections.deque is preferred.
7      """
8      def __init__(self) -> None:
9          """
10         Initializes an empty queue.
11         """
12         self._items: List[Any] = []
13
14     def is_empty(self) -> bool:
15         """
16         Checks if the queue is empty.
17
18         Returns:
19             bool: True if the queue is empty, False otherwise.
20         """
21         return not self._items
22
23     def enqueue(self, item: Any) -> None:
24         """
25         Adds an item to the back (end) of the queue.
26
27         Args:
28             item (Any): The item to be added to the queue.
29         """
30         self._items.append(item)
31
32     def dequeue(self) -> Any:
33         """
34         Removes and returns the item from the front of the queue.
35
36         Raises:
37             IndexError: If the queue is empty.
38
39         Returns:
40             Any: The item at the front of the queue.
41         """
42         if self.is_empty():
43             raise IndexError("dequeue from an empty queue")
44         return self._items.pop(0)
45
46     def peek(self) -> Any:
47         """
48         Returns the item at the front of the queue without removing it.
49
50         Raises:
51             IndexError: If the queue is empty.
52
53         Returns:
54             Any: The item at the front of the queue.
55         """
56         if self.is_empty():
57             raise IndexError("peek from an empty queue")
58         return self._items[0]

```

0.0

```

3 class Queue:
4     return self._items[0]
5
6
7
8
9
10
11
12
13
14
15
16
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
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

class Queue:
    def __init__(self):
        self._items = []

    def __str__(self) -> str:
        """
        Returns a string representation of the queue.
        """
        return str(self._items)

    def __len__(self) -> int:
        """
        Returns the number of items in the queue.
        """
        return len(self._items)

# --- Example Usage ---
if __name__ == "__main__":
    print("Creating a new queue...")
    q = Queue()

    print(f"Is the queue empty? {q.is_empty()}") # Expected: True

    print("\nEnqueuing items: 'A', 'B', 'C'")
    q.enqueue('A')
    q.enqueue('B')
    q.enqueue('C')

    print(f"Current queue: {q}")
    print(f"Queue size: {len(q)}")
    print(f"Is the queue empty? {q.is_empty()}") # Expected: False

    print(f"\nPeeking at the front item: {q.peek()}") # Expected: 'A'
    print(f"Current queue after peek: {q}")

    print(f"\nDequeuing an item: {q.dequeue()}") # Expected: 'A'
    print(f"Current queue after dequeue: {q}")

    print(f"\nDequeuing another item: {q.dequeue()}") # Expected: 'B'
    print(f"Current queue: {q}")

    print("\nDequeuing the last item...")
    q.dequeue()
    print(f"Is the queue empty now? {q.is_empty()}") # Expected: True

```

# OUTPUT:

```

PS C:\Users\ramch\OneDrive\Desktop\ai> & C:\Users\ramch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ramch\OneDrive\Desktop\ai\11.2.2.py
Creating a new queue...
Is the queue empty? True

Enqueuing items: 'A', 'B', 'C'
Current queue: ['A', 'B', 'C']
Queue size: 3
Is the queue empty? False

Peeking at the front item: A
Current queue after peek: ['A', 'B', 'C']

Dequeuing an item: A
Current queue after dequeue: ['B', 'C']
Dequeuing another item: B
Current queue: ['C']

Dequeuing the last item...
Is the queue empty now? True
PS C:\Users\ramch\OneDrive\Desktop\ai>

```

# OBSERVATION:

The code generated by AI is more accurate about the queue in data structures and it also passed all the test cases.

## **TASK-03:**

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

Sample Input Code:

```
class Node:
    pass
class LinkedList:
    pass
```

## **PROMPT:**

Generate a python code of singly linked list with insert and display methods.

## **CODE:**

```

1123.py > _
1  from typing import Any, Optional
2
3  class Node:
4      """
5      An object for storing a single node of a linked list.
6      Models two attributes: data and the link to the next node in the list.
7      """
8      def __init__(self, data: Any, next_node: Optional['Node'] = None):
9          """
10         Initializes a Node.
11
12         Args:
13             data (Any): The data to be stored in the node.
14             next_node (Optional['Node']): The next node in the list. Defaults to None.
15         """
16         self.data = data
17         self.next = next_node
18
19  class LinkedList:
20      """
21      A simple implementation of a Singly Linked List.
22      """
23      def __init__(self):
24          """
25          Initializes an empty linked list.
26          """
27          self.head: Optional[Node] = None
28
29      def is_empty(self) -> bool:
30          """
31          Checks if the linked list is empty.
32
33          Returns:
34              bool: True if the list is empty, False otherwise.
35          """
36          return self.head is None
37
38      def insert(self, data: Any) -> None:
39          """
40          Inserts a new node with the given data at the end of the list.
41
42          Args:
43              data (Any): The data for the new node.
44          """
45          new_node = Node(data)
46          # If the list is empty, the new node becomes the head.
47          if self.is_empty():
48              self.head = new_node
49              return
50
51          # Otherwise, traverse to the end of the list.
52          last_node = self.head
53          while last_node.next:
54              last_node = last_node.next
55
56          # Set the next pointer of the last node to the new node.
57          last_node.next = new_node
58

```



```

19 class LinkedList:
20
21     def display(self) -> None:
22         """
23         Traverses the linked list and prints the data of each node.
24         """
25         if self.is_empty():
26             print("Linked list is empty.")
27             return
28
29         current = self.head
30         nodes = []
31         while current:
32             nodes.append(str(current.data))
33             current = current.next
34         print(" -> ".join(nodes))
35
36 # --- Example Usage ---
37 if __name__ == "__main__":
38     print("Creating a new linked list...")
39     ll = LinkedList()
40
41     print("Is the list empty?", ll.is_empty()) # Expected: True
42     ll.display() # Expected: Linked list is empty.
43
44     print("\nInserting elements: 10, 20, 30")
45     ll.insert(10)
46     ll.insert(20)
47     ll.insert(30)
48
49     print("Is the list empty?", ll.is_empty()) # Expected: False
50
51     print("\nDisplaying the list:")
52     ll.display() # Expected: 10 -> 20 -> 30
53
54     print("\nInserting another element: 40")
55     ll.insert(40)
56
57     print("Displaying the final list:")
58     ll.display() # Expected: 10 -> 20 -> 30 -> 40
59

```

# OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.3.py
Creating a new linked list...
Is the list empty? True
Linked list is empty.

Inserting elements: 10, 20, 30
Is the list empty? False

Displaying the list:
10 -> 20 -> 30

Inserting another element: 40
Displaying the final list:
10 -> 20 -> 30 -> 40
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

## OBSERVATION:

The code generated by the AI is used to perform the operations in the data structures like singly linked list performing insert and delete operations accurately.

## TASK-04:

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

Sample Input Code:

```
class BST:
```

```
pass
```

## PROMPT:

Generate a python code which creates a BST with insert

and in-order methods.

## CODE:

```
11.2.4.py > Node > __init__
1 from typing import Any, Optional, Generator
2 class Node:
3     """
4     An object for storing a single node of a Binary Search Tree.
5     Models three attributes: the node's value, and pointers to the
6     left and right children.
7     """
8     def __init__(self, value: Any):
9         """
10        Initializes a Node.
11        Args:
12            value (Any): The value to be stored in the node.
13        """
14        self.value = value
15        self.left: Optional['Node'] = None
16        self.right: Optional['Node'] = None
17
18 class BST:
19     """
20     A simple implementation of a Binary Search Tree (BST).
21     """
22     def __init__(self):
23         """
24         Initializes an empty BST.
25         """
26         self.root: Optional[Node] = None
27
28     def insert(self, value: Any) -> None:
29         """
30         Inserts a new value into the BST, maintaining the BST property.
31
32         Args:
33             value (Any): The value to insert into the tree.
34         """
35         if self.root is None:
36             self.root = Node(value)
37         else:
38             self._insert_recursive(self.root, value)
39
40     def _insert_recursive(self, current_node: Node, value: Any) -> Node:
41         """
42         A private helper method to recursively find the correct position
43         and insert the new node.
44
45         Args:
46             current_node (Node): The node to start the search from.
47             value (Any): The value to insert.
48
49         Returns:
50             Node: The (potentially new) root of the subtree.
51         """
52         if value < current_node.value:
53             if current_node.left is None:
54                 current_node.left = Node(value)
55             else:
56                 self._insert_recursive(current_node.left, value)
57         elif value > current_node.value:
58             if current_node.right is None:
59                 current_node.right = Node(value)
60             else:
61                 self._insert_recursive(current_node.right, value)
```

```

17 class BST:
37     def _insert_recursive(self, current_node: Node, value: Any) -> Node:
59         """
60         # If value == current_node.value, we do nothing (no duplicates).
61         return current_node
62
63     def inorder_traversal(self) -> None:
64         """
65         Performs an in-order traversal of the tree and prints the values.
66         In-order traversal visits nodes in ascending order.
67         """
68         print("In-order Traversal:", end=" ")
69         nodes = list(self._inorder_generator(self.root))
70         print(" -> ".join(map(str, nodes)))
71
72     def _inorder_generator(self, current_node: Optional[Node]) -> Generator[Any, None, None]:
73         """
74         A private generator for recursively performing in-order traversal.
75         It yields nodes in the order: left subtree, root, right subtree.
76
77         Args:
78             current_node (Optional[Node]): The node to start traversal from.
79         """
80         if current_node:
81             yield from self._inorder_generator(current_node.left)
82             yield current_node.value
83             yield from self._inorder_generator(current_node.right)
84
85 # --- Example Usage ---
86 if __name__ == "__main__":
87     print("Creating a new Binary Search Tree...")
88     bst = BST()
89
90     # The order of insertion matters for the tree's shape.
91     print("Inserting values: 50, 30, 70, 20, 40, 60, 80")
92     values_to_insert = [50, 30, 70, 20, 40, 60, 80]
93     for val in values_to_insert:
94         bst.insert(val)
95
96     # The resulting tree structure will be:
97     #      50
98     #     / \
99     #    30  70
100    #   / \  / \
101    #  20 40 60 80
102
103    # In-order traversal should print the nodes in sorted order.
104    bst.inorder_traversal() # Expected: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80
105
106    print("\nInserting a new value: 55")
107    bst.insert(55)
108    bst.inorder_traversal() # Expected: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80
109
110    print("\nAttempting to insert a duplicate value: 40")
111    bst.insert(40) # This should not change the tree
112    bst.inorder_traversal() # Expected: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80

```

# OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.4.py
Creating a new Binary Search Tree...
Inserting values: 50, 30, 70, 20, 40, 60, 80
In-order Traversal: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80

Inserting a new value: 55
In-order Traversal: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80

Attempting to insert a duplicate value: 40
In-order Traversal: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

## OBSERVATION:

The BST in data structures with insertion and in-order traversal methods are generated in python which are more helpful and in an efficient way to understand.

## TASK-05:

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

Sample Input Code:

```
class HashTable:
```

```
pass
```

## PROMPT:

Generate a python code which helps to implement the hashtable with basic insert and delete methods.

# CODE:

```
Python 3.12.5 Spyder
1 from typing import Any, List, Tuple, Optional
2
3 class HashTable:
4     """
5     A simple implementation of a Hash Table using chaining for collision resolution.
6     """
7     def __init__(self, size: int = 101):
8         """
9         Initializes the hash table.
10
11         Args:
12             size (int): The number of buckets in the hash table. A prime number
13                        is often recommended to help distribute keys more uniformly.
14         """
15         if size < 1:
16             raise ValueError("Hash table size must be a positive integer.")
17         self.size = size
18         # Initialize the table with empty lists (chains) for each bucket
19         self.table: List[List[Tuple[Any, Any]]] = [[] for _ in range(self.size)]
20
21     def _hash(self, key: Any) -> int:
22         """
23         A private method to compute the hash of a key and map it to a bucket index.
24         This uses Python's built-in hash() and the modulo operator.
25
26         Args:
27             key (Any): The key to hash.
28
29         Returns:
30             int: The index of the bucket for the given key.
31         """
32         return hash(key) % self.size
33
34     def insert(self, key: Any, value: Any) -> None:
35         """
36         Inserts a key-value pair into the hash table.
37         If the key already exists, its value is updated.
38
39         Args:
40             key (Any): The key to insert.
41             value (Any): The value associated with the key.
42         """
43         index = self._hash(key)
44         bucket = self.table[index]
45
46         # Check if the key already exists in the bucket (chain)
47         for i, (existing_key, _) in enumerate(bucket):
48             if existing_key == key:
49                 # Key found, update the value and return
50                 bucket[i] = (key, value)
51                 return
52
53         # Key not found, append the new key-value pair to the chain
54         bucket.append((key, value))
55
56     def search(self, key: Any) -> Optional[Any]:
57         """
58         Searches for a key in the hash table and returns its value.
59         """
```

```

112.5.py > ...
3 class HashTable:
56     def search(self, key: Any) -> Optional[Any]:
59
60         Args:
61             key (Any): The key to search for.
62
63         Returns:
64             Optional[Any]: The value associated with the key if found, otherwise None.
65         """
66         index = self._hash(key)
67         bucket = self.table[index]
68
69         # Traverse the chain in the bucket
70         for existing_key, value in bucket:
71             if existing_key == key:
72                 return value # Key found, return its value
73
74         return None # Key not found
75
76     def delete(self, key: Any) -> bool:
77         """
78         Deletes a key-value pair from the hash table.
79
80         Args:
81             key (Any): The key to delete.
82
83         Returns:
84             bool: True if the key was found and deleted, False otherwise.
85         """
86         index = self._hash(key)
87         bucket = self.table[index]
88
89         # Find the key in the chain and remove it
90         for i, (existing_key, _) in enumerate(bucket):
91             if existing_key == key:
92                 bucket.pop(i)
93                 return True # Deletion successful
94
95         return False # Key was not found
96
97     def __str__(self) -> str:
98         """
99         Returns a string representation of the hash table's contents.
100         """
101         elements = []
102         for i, bucket in enumerate(self.table):
103             if bucket:
104                 elements.append(f"Bucket {i}: {bucket}")
105         return "\n".join(elements) if elements else "HashTable is empty."
106
107 # --- Example Usage ---
108 if __name__ == "__main__":
109     print("Creating a new hash table of size 10...")
110     ht = HashTable(10)
111
112     print("\n--- Inserting Elements ---")
113     ht.insert("name", "Alice")
114     ht.insert("age", 30)
115
116     # --- Example Usage ---

```

```

111
112     print("\n--- Inserting Elements ---")
113     ht.insert("name", "Alice")
114     ht.insert("age", 30)
115     ht.insert("city", "New York")
116     ht.insert(99, "is a number") # Keys can be different types
117     ht.insert("name", "Bob") # This will update the value for the key "name"
118
119     print("Current Hash Table state:")
120     print(ht)
121
122     print("\n--- Searching for Elements ---")
123     print(f"Search for 'name': {ht.search('name')}") # Expected: Bob
124     print(f"Search for 'age': {ht.search('age')}") # Expected: 30
125     print(f"Search for 'country': {ht.search('country')}") # Expected: None
126
127     print("\n--- Deleting Elements ---")
128     print(f"Deleting 'age'... Success: {ht.delete('age')}")
129     print(f"Deleting 'country'... Success: {ht.delete('country')}")
130
131     print("\nFinal Hash Table state:")
132     print(ht)
133
134     print(f"\nSearching for 'age' again: {ht.search('age')}") # Expected: None
135

```

# OUTPUT:

```
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:\Users\ranch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ranch\OneDrive\Desktop\ai\11.2.5.py
Creating a new hash table of size 10...

--- Inserting Elements ---
Current Hash Table state:
Bucket 0: [('name', 'Bob')]
Bucket 1: [('age', 30), ('city', 'New York')]
Bucket 9: [(99, 'is a number')]

--- Searching for Elements ---
Search for 'name': Bob
Search for 'age': 30
Search for 'country': None

--- Deleting Elements ---
Deleting 'age'... Success: True
Deleting 'country'... Success: False

Final Hash Table state:
Bucket 0: [('name', 'Bob')]
Bucket 1: [('city', 'New York')]
Bucket 9: [(99, 'is a number')]

Searching for 'age' again: None
PS C:\Users\ranch\OneDrive\Desktop\ai>
```

# OBSERVATION:

AI generated the code of data structures topic hashtable with basic insert, search and delete which aids us to understand the hashtable using the programming language like python.

# TASK-06:

Use AI to implement a graph using an adjacency list.

Sample Input Code:

```
class Graph:
```

```
pass
```

# PROMPT:

Generate a python code which implements the graph using an adjacency list in data structures.



# CODE:

```
112.6py > _
1 from typing import Any, Dict, List, Set
2
3 class Graph:
4     """
5     A simple implementation of an undirected graph using an adjacency list.
6     """
7     def __init__(self):
8         """
9         Initializes an empty graph.
10        The adjacency list is represented as a dictionary where keys are
11        vertices and values are lists of their adjacent vertices.
12        """
13        self.adj_list: Dict[Any, List[Any]] = {}
14
15    def add_vertex(self, vertex: Any) -> bool:
16        """
17        Adds a vertex to the graph.
18
19        Args:
20            vertex (Any): The vertex to be added. Must be a hashable type.
21
22        Returns:
23            bool: True if the vertex was added, False if it already exists.
24        """
25        if vertex not in self.adj_list:
26            self.adj_list[vertex] = []
27            return True
28        return False
29
30    def add_edge(self, v1: Any, v2: Any) -> bool:
31        """
32        Adds an undirected edge between two vertices.
33        If the vertices do not exist, they are created first.
34
35        Args:
36            v1 (Any): The first vertex.
37            v2 (Any): The second vertex.
38
39        Returns:
40            bool: True if the edge was added, False if it already existed.
41        """
42        # Ensure both vertices exist in the graph
43        self.add_vertex(v1)
44        self.add_vertex(v2)
45
46        # Add the edge for an undirected graph
47        if v2 not in self.adj_list[v1]:
48            self.adj_list[v1].append(v2)
49        if v1 not in self.adj_list[v2]:
50            self.adj_list[v2].append(v1)
51        return True
52
53    def display(self) -> None:
54        """
55        Prints the adjacency list representation of the graph.
56        """
57        if not self.adj_list:
58            print("Graph is empty.")
```

```
11.2.6.py > _
3 class Graph:
53     def display(self) -> None:
58         print("Graph is empty.")
59         return
60         for vertex in self.adj_list:
61             print(f"{vertex}: {self.adj_list[vertex]}")
62
63     def get_vertices(self) -> List[Any]:
64         """
65         Returns a list of all vertices in the graph.
66         """
67         return list(self.adj_list.keys())
68
69     def get_edges(self) -> List[tuple[Any, Any]]:
70         """
71         Returns a list of all edges in the graph.
72         """
73         edges = set()
74         for vertex, neighbors in self.adj_list.items():
75             for neighbor in neighbors:
76                 # To avoid duplicate edges like (A, B) and (B, A)
77                 if vertex < neighbor:
78                     edges.add((vertex, neighbor))
79         return list(edges)
80
81 # --- Example Usage ---
82 if __name__ == "__main__":
83     print("Creating a new graph...")
84     g = Graph()
85
86     print("\n--- Adding Vertices ---")
87     g.add_vertex("A")
88     g.add_vertex("B")
89     g.add_vertex("C")
90     print("Graph after adding vertices:")
91     g.display()
92
93     print("\n--- Adding Edges ---")
94     g.add_edge("A", "B")
95     g.add_edge("B", "C")
96     g.add_edge("C", "A")
97     # Adding an edge with a new vertex
98     g.add_edge("A", "D")
99     print("Graph after adding edges:")
100     g.display()
101
102     print("\n--- Retrieving Vertices and Edges ---")
103     print("Vertices:", g.get_vertices())
104     print("Edges:", g.get_edges())
105
106     print("\n--- Adding a duplicate edge (A, B) ---")
107     success = g.add_edge("A", "B")
108     print(f"Was the edge added? {success}") # Expected: False
109     print("Graph state remains the same:")
110     g.display()
111
```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:\Users\ranch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ranch\OneDrive\Desktop\ai\11.2.6.py
Creating a new graph...

--- Adding Vertices ---
Graph after adding vertices:
A: []
B: []
C: []

--- Adding Edges ---
Graph after adding edges:
A: ['B', 'C', 'D']
B: ['A', 'C']
C: ['B', 'A']
D: ['A']

--- Retrieving Vertices and Edges ---
Vertices: ['A', 'B', 'C', 'D']
Edges: [('A', 'C'), ('A', 'B'), ('B', 'C'), ('A', 'D')]

--- Adding a duplicate edge (A, B) ---
Was the edge added? false
Graph state remains the same:
A: ['B', 'C', 'D']
B: ['A', 'C']
C: ['B', 'A']
D: ['A']
PS C:\Users\ranch\OneDrive\Desktop\ai>
```

## OBSERVATION:

Implementation of graph with the adjacency list is generated by the AI which makes us know about the graph in an efficient way and easy to understand.

## TASK-07:

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

Sample Input Code:

```
class PriorityQueue:
```

```
pass
```

## PROMPT:

Write a python code which implement the priority queue using the heapq module.

# CODE:

```
11.2.1.py 11.2.2.py 11.2.3.py 11.2.4.py 11.2.5.py 11.2.6.py 11.2.7.py x 11.2.8.py 11.2.9.py 11.2.10.py
1 import heapq
2 from typing import Any, List, Tuple
3
4 class PriorityQueue:
5     """
6     A simple implementation of a Priority Queue using Python's heapq module.
7     Lower numbers indicate higher priority.
8     """
9     def __init__(self):
10         """
11         Initializes an empty priority queue.
12         """
13         self._heap: List[Tuple[int, Any]] = []
14
15     def push(self, item: Any, priority: int) -> None:
16         """
17         Adds an item to the queue with an associated priority.
18
19         Args:
20             item (Any): The item to be stored in the queue.
21             priority (int): The priority of the item. Lower numbers are higher priority.
22         """
23         # heapq is a min-heap, so it naturally keeps the item with the
24         # smallest priority value at the front.
25         heapq.heappush(self._heap, (priority, item))
26
27     def pop(self) -> Any:
28         """
29         Removes and returns the item with the highest priority (lowest number).
30
31         Raises:
32             IndexError: If the priority queue is empty.
33
34         Returns:
35             Any: The item with the highest priority.
36         """
37         if self.is_empty():
38             raise IndexError("pop from an empty priority queue")
39         # heappop removes and returns the smallest item from the heap.
40         # we return only the item (index 1), not its priority (index 0).
41         return heapq.heappop(self._heap)[1]
42
43     def peek(self) -> Any:
44         """
45         Returns the item with the highest priority without removing it.
46
47         Raises:
48             IndexError: If the priority queue is empty.
49
50         Returns:
51             Any: The item with the highest priority.
52         """
53         if self.is_empty():
54             raise IndexError("peek from an empty priority queue")
55         # The highest priority item is always at the root of the heap (index 0).
56         return self._heap[0][1]
57
58     def is_empty(self) -> bool:
```

```

4 class PriorityQueue:
58     def is_empty(self) -> bool:
60         Checks if the priority queue is empty.
61
62         Returns:
63             bool: True if the queue is empty, False otherwise.
64         """
65         return not self._heap
66
67     def __len__(self) -> int:
68         """
69         Returns the number of items in the priority queue.
70         """
71         return len(self._heap)
72
73     def __str__(self) -> str:
74         """
75         Returns a string representation of the priority queue's heap.
76         Note: The list representation is not necessarily sorted.
77         """
78         return str(self._heap)
79
80 # --- Example Usage ---
81 if __name__ == "__main__":
82     pq = PriorityQueue()
83     print(f"Is the queue empty? {pq.is_empty()}") # Expected: True
84
85     print("\nPushing tasks with priorities...")
86     # (item, priority)
87     pq.push("Task: Write report", 3)
88     pq.push("Task: Answer urgent email", 1) # Highest priority
89     pq.push("Task: Go to meeting", 2)
90     pq.push("Task: Get coffee", 4) # Lowest priority
91
92     print(f"Current queue size: {len(pq)}")
93     print(f"Internal heap representation: {pq}")
94     print(f"Is the queue empty? {pq.is_empty()}") # Expected: False
95
96     print(f"\nHighest priority task (peek): {pq.peek()}") # Expected: Task: Answer urgent email
97
98     print("\nProcessing tasks in order of priority:")
99     while not pq.is_empty():
100         task = pq.pop()
101         print(f" - Popped: {task}")
102
103     print(f"\nIs the queue empty now? {pq.is_empty()}") # Expected: True
104

```

# OUTPUT:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\ramch\OneDrive\Desktop\ai> & C:\Users\ramch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ramch\OneDrive\Desktop\ai\11.2.7.py
● Is the queue empty? True

Pushing tasks with priorities...
Current queue size: 4
Internal heap representation: [(1, 'Task: Answer urgent email'), (3, 'Task: Write report'), (2, 'Task: Go to meeting'), (4, 'Task: Get coffee')]
Is the queue empty? False

Highest priority task (peek): Task: Answer urgent email

Processing tasks in order of priority:
- Popped: Task: Answer urgent email
- Popped: Task: Go to meeting
- Popped: Task: Write report
- Popped: Task: Get coffee

Is the queue empty now? True
○ PS C:\Users\ramch\OneDrive\Desktop\ai>

```

## **OBSERVATION:**

The priority queue is generated using the heapq module of python which implements them accurately.

## **TASK-08:**

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

Sample Input Code:

```
class DequeDS:
```

```
pass
```

## **PROMPT:**

Write a python code to implement the double-ended queue using collections .deque

## **CODE:**

```

1128.py>...
1  from collections import deque
2  from typing import Any, Optional
3
4  class Deque05:
5      """
6      A wrapper class for Python's collections.deque to provide a clear
7      and explicit double-ended queue interface.
8      """
9
10     def __init__(self):
11         """
12         Initializes an empty deque.
13         """
14         self._deque = deque()
15
16     def add_first(self, item: Any) -> None:
17         """
18         Adds an item to the front of the deque.
19
20         Args:
21             item (Any): The item to be added.
22         """
23         self._deque.appendleft(item)
24
25     def add_last(self, item: Any) -> None:
26         """
27         Adds an item to the back of the deque.
28
29         Args:
30             item (Any): The item to be added.
31         """
32         self._deque.append(item)
33
34     def remove_first(self) -> Any:
35         """
36         Removes and returns the item from the front of the deque.
37
38         Raises:
39             IndexError: If the deque is empty.
40
41         Returns:
42             Any: The item from the front.
43         """
44         if self.is_empty():
45             raise IndexError("remove_first from an empty deque")
46         return self._deque.popleft()
47
48     def remove_last(self) -> Any:
49         """
50         Removes and returns the item from the back of the deque.
51
52         Raises:
53             IndexError: If the deque is empty.
54
55         Returns:
56             Any: The item from the back.
57         """
58         if self.is_empty():
59             raise IndexError("remove_last from an empty deque")

```

```

4 class DequeDS:
5     return self._deque.pop()
6
7     def peek_first(self) -> Any:
8         """
9         Returns the item at the front of the deque without removing it.
10
11         Raises:
12             IndexError: If the deque is empty.
13
14         Returns:
15             Any: The item at the front.
16         """
17         if self.is_empty():
18             raise IndexError("peek_first from an empty deque")
19         return self._deque[0]
20
21     def peek_last(self) -> Any:
22         """
23         Returns the item at the back of the deque without removing it.
24
25         Raises:
26             IndexError: If the deque is empty.
27
28         Returns:
29             Any: The item at the back.
30         """
31         if self.is_empty():
32             raise IndexError("peek_last from an empty deque")
33         return self._deque[-1]
34
35     def is_empty(self) -> bool:
36         """
37         Checks if the deque is empty.
38
39         Returns:
40             bool: True if the deque is empty, False otherwise.
41         """
42         return len(self._deque) == 0
43
44     def _len__(self) -> int:
45         """
46         Returns the number of items in the deque.
47         """
48         return len(self._deque)
49
50     def __str__(self) -> str:
51         """
52         Returns a string representation of the deque.
53         """
54         return f"DequeDS({list(self._deque)})"
55
56 # --- Example Usage ---
57 if __name__ == "__main__":
58     d = DequeDS()
59     print(f"Is deque empty? {d.is_empty()}") # Expected: True
60
61     print("\nAdding 'B' and 'C' to the back...")
62
63

```

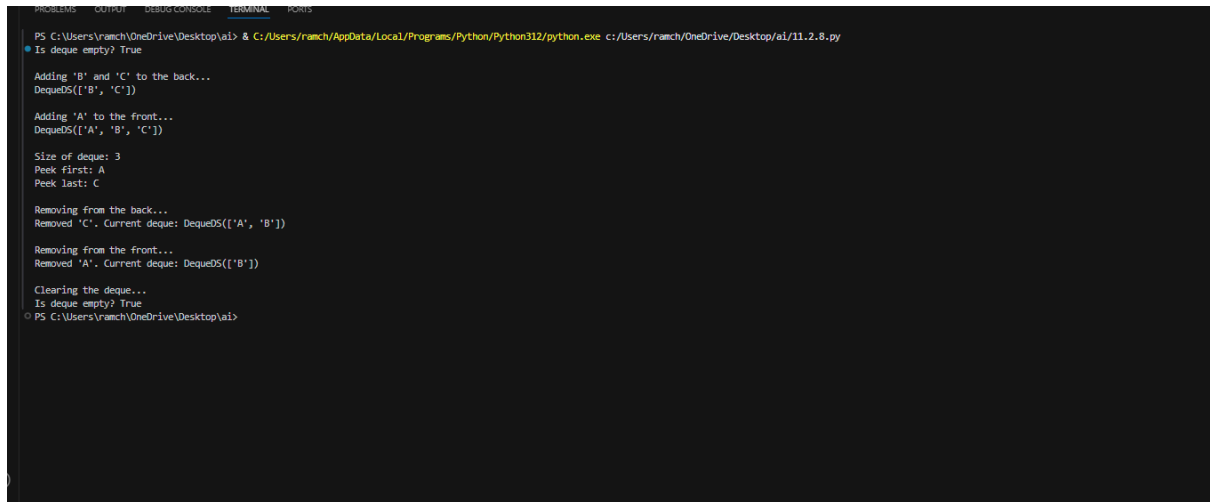
```

112 d = DequeDS()
113 print(f"Is deque empty? {d.is_empty()}") # Expected: True
114
115 print("\nAdding 'B' and 'C' to the back...")
116 d.add_last("B")
117 d.add_last("C")
118 print(d) # Expected: DequeDS(['B', 'C'])
119
120 print("\nAdding 'A' to the front...")
121 d.add_first("A")
122 print(d) # Expected: DequeDS(['A', 'B', 'C'])
123
124 print(f"\nSize of deque: {len(d)}") # Expected: 3
125 print(f"Peek first: {d.peek_first()}") # Expected: A
126 print(f"Peek last: {d.peek_last()}") # Expected: C
127
128 print("\nRemoving from the back...")
129 item = d.remove_last()
130 print(f"Removed '{item}'. Current deque: {d}") # Expected: DequeDS(['A', 'B'])
131
132 print("\nRemoving from the front...")
133 item = d.remove_first()
134 print(f"Removed '{item}'. Current deque: {d}") # Expected: DequeDS(['B'])
135
136 print("\nClearing the deque...")
137 d.remove_first()
138 print(f"Is deque empty? {d.is_empty()}") # Expected: True
139

```



# OUTPUT:



```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:\Users\ramch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ramch\OneDrive\Desktop\ai\11.2.8.py
Is deque empty? True

Adding 'B' and 'C' to the back...
DequeDS(['B', 'C'])

Adding 'A' to the front...
DequeDS(['A', 'B', 'C'])

Size of deque: 3
Peek first: A
Peek last: C

Removing from the back...
Removed 'C'. Current deque: DequeDS(['A', 'B'])

Removing from the front...
Removed 'A'. Current deque: DequeDS(['B'])

Clearing the deque...
Is deque empty? True
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

# OBSERVATION:

The code generated by the AI of data structures topic of double-ended queue with the help of python to easily understand the topic accurately.

# TASK-09:

Use AI to generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

Sample Input Code:

# No code, prompt AI for a data structure comparison table

# PROMPT:

Generate a comparative table of different data structures including time complexities .

# TABLE:

Data Structure	Access (by index/key)	Search (by value)	Insertion (at end)	Insertion (at start)	Deletion (at end)	Deletion (at start)	Notes
Python List (Array)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	Insertion/deletion at the start is slow because all other elements must be shifted.
Stack (using List)	$O(n)$	$O(n)$	$O(1)$	N/A	$O(1)$	N/A	Operations are only on the "top" (end of the list). Accessing arbitrary elements is not a standard stack operation.
Queue (using List)	$O(n)$	$O(n)$	$O(1)$	N/A	N/A	$O(n)$	<code>deque</code> (from the start) is inefficient. <code>collections.deque</code> is highly preferred.
Queue (using deque)	$O(n)$	$O(n)$	$O(1)$	N/A	N/A	$O(1)$	<code>collections.deque</code> is optimized for fast appends and pops from both ends.
Singly Linked List	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	Insertion/deletion at the start is fast. Finding the end requires traversing the whole list.
Doubly Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Faster end-of-list operations than a singly linked list because it can be tracked. Uses more memory per node.
Hash Table (Dict)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Average case is $O(1)$ . Worst case (due to hash collisions) can be $O(n)$ .
Binary Search Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	Assumes the tree is balanced. In the worst case (an unbalanced tree), it degrades to $O(n)$ .

## TASK-10:

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
  - 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.

### **PROMPT:**

Generate the python code according to the given above details.

### **CODE:**

```
11.2.1.py 11.2.2.py 11.2.3.py 11.2.4.py 11.2.5.py 11.2.6.py 11.2.7.py 11.2.8.py 11.2.9.py 11.2.10.py X
11.2.10.py > _
1 from typing import Any, Dict, Optional
2
3 class EventRegistration:
4     """
5     Manages participant registration for an event using a hash table.
6     This provides fast lookups, additions, and removals of participants.
7     """
8     def __init__(self, event_name: str):
9         """
10         Initializes the registration system for a specific event.
11
12         Args:
13             event_name (str): The name of the event.
14         """
15         self.event_name = event_name
16         # The hash table (dictionary) to store participants.
17         # Key: student_id, Value: student_name
18         self._participants: Dict[int, str] = {}
19         print(f"Event Registration System for '{self.event_name}' is now open!")
20
21     def register_participant(self, student_id: int, student_name: str) -> bool:
22         """
23         Registers a new participant for the event.
24
25         Args:
26             student_id (int): The unique ID of the student.
27             student_name (str): The name of the student.
28
29         Returns:
30             bool: True if registration was successful, False if the student
31                  is already registered.
32         """
33         if student_id in self._participants:
34             print(f"-> Error: Student (student_name) (ID: {student_id}) is already registered.")
35             return False
36
37         self._participants[student_id] = student_name
38         print(f"-> Success: (student_name) (ID: {student_id}) has been registered.")
39         return True
40
41     def remove_participant(self, student_id: int) -> bool:
42         """
43         Removes a participant from the event.
44
45         Args:
46             student_id (int): The ID of the student to remove.
47
48         Returns:
49             bool: True if the student was found and removed, False otherwise.
50         """
51         if student_id in self._participants:
52             student_name = self._participants.pop(student_id)
53             print(f"-> Success: (student_name) (ID: {student_id}) has been removed.")
54             return True
55
56         print(f"-> Error: Student with ID (student_id) not found.")
57         return False
58
59
60
```

```
58
59
60     def find_participant(self, student_id: int) -> Optional[str]:
61         """
62         Searches for a participant by their student ID.
63
64         Args:
65             student_id (int): The ID of the student to find.
66
67         Returns:
68             Optional[str]: The name of the student if found, otherwise None.
69         """
70         return self._participants.get(student_id)
71
72     def display_participants(self) -> None:
73         """
74         Displays a list of all registered participants.
75
76         print(f"\n--- Registered Participants for '{self.event_name}' ---")
77         if not self._participants:
78             print("No participants are currently registered.")
79         else:
80             for student_id, student_name in self._participants.items():
81                 print(f"ID: {student_id}, Name: {student_name}")
82             print("-----")
83
84 # --- Example Usage ---
85 if __name__ == "__main__":
86     # Create a registration system for a "CodeFest 2024" event
87     codefest_event = EventRegistration("CodeFest 2024")
88
89     print("\n--- Registering Participants ---")
90     codefest_event.register_participant(101, "Alice")
91     codefest_event.register_participant(102, "Bob")
92     codefest_event.register_participant(103, "Charlie")
93
94     # Attempt to register a duplicate
95     codefest_event.register_participant(101, "Alice")
96
97     codefest_event.display_participants()
98
99     print("\n--- Searching for a Participant ---")
100     student_id_to_find = 102
101     student_name = codefest_event.find_participant(student_id_to_find)
102     if student_name:
103         print(f"Found participant with ID {student_id_to_find}: {student_name}")
104     else:
105         print(f"Participant with ID {student_id_to_find} not found.")
106
107     print("\n--- Removing a Participant ---")
108     codefest_event.remove_participant(103) # Successful removal
109     codefest_event.remove_participant(999) # Failed removal (not found)
110
111     codefest_event.display_participants()
112
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:\Users\ranch\AppData\Local\Programs\Python\Python312\python.exe c:\Users\ranch\OneDrive\Desktop\ai\11.2.10.py
Event Registration System for 'CodeFest 2024' is now open!

--- Registering Participants ---
-> Success: Alice (ID: 101) has been registered.
-> Success: Bob (ID: 102) has been registered.
-> Success: Charlie (ID: 103) has been registered.
-> Error: Student Alice (ID: 101) is already registered.

--- Registered Participants for 'CodeFest 2024' ---
- ID: 101, Name: Alice
- ID: 102, Name: Bob
- ID: 103, Name: Charlie
-----

--- Searching for a Participant ---
Found participant with ID 102: Bob

--- Removing a Participant ---
-> Success: Charlie (ID: 103) has been removed.
-> Error: Student with ID 999 not found.

--- Registered Participants for 'CodeFest 2024' ---
- ID: 101, Name: Alice
- ID: 102, Name: Bob
-----
PS C:\Users\ranch\OneDrive\Desktop\ai>
```

# TABLE:

Data Structure Selection and Justification		
Here is a table that maps each system feature to the most suitable data structure and provides a justification for the choice.		
Feature	Chosen Data Structure	Justification
1. Student Attendance Tracking	Stack	A stack is ideal for tracking entries and exits. When a student enters, their ID is pushed onto the stack; when they exit, their ID is popped. This Last-In, First-Out (LIFO) model naturally represents the most recent students on campus and can quickly show who is currently inside.
2. Event Registration System	Hash Table	A hash table (like a Python dictionary) is perfect for managing event participants. It allows for extremely fast $O(1)$ average-time lookups, insertions, and removals using a unique student ID as the key. This efficiency is crucial for quickly checking if a student is registered or for managing a large list of attendees.
3. Library Book Borrowing	Binary Search Tree (BST)	A BST, keyed on book titles or ISBNs, is a great choice for managing available books. It keeps the books in a sorted order, allowing for efficient $O(\log n)$ searching. This is much faster than a linear scan when the library has thousands of books.
4. Bus Scheduling System	Graph	A graph is the most natural way to model a bus network. Each bus stop can be represented as a vertex, and the routes between stops can be represented as edges. This structure allows for solving complex problems like finding the shortest path between two stops or identifying all possible routes.
5. Cafeteria Order Queue	Queue	A queue is the perfect data structure for this task as it follows the First-In, First-Out (FIFO) principle. Students are served in the exact order they arrive, just like a real-world line. This ensures fairness and is the most intuitive way to manage an order system.

# OBSERVATION:

The AI generated the code in an efficient way according to the details given in which it include all the data structures concepts to make all easily understand . As the task contains much more information it should be handled in an efficient way.

