

AI ASSISTED CODING
LAB ASSIGNMENT: 11.2

NAME: S.Nagarjuna reddy

H.NO: 2403A52064

B.NO: 03

TASK: Stack Implementation

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

Sample Input Code: class Stack: pass.

PROMPT:

Generate python code and stack
Implementation

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

Input Code: class
Stack: pass.

CODE & OUTPUT:

The screenshot shows two instances of Microsoft Visual Studio Code running side-by-side. Both instances have the same file open: `AI LAB 11.1.py`. The code defines a `Stack` class with methods for pushing, popping, peeking, and checking if the stack is empty. The terminals below each editor window show the execution of this code. In the first terminal, the stack is initialized with 10, then 20 is pushed, followed by 20 being popped twice, resulting in an empty stack. In the second terminal, the stack is initialized with 10, then 20 is pushed, followed by 20 being popped twice, resulting in an empty stack. The output from both terminals is identical.

```
1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def push(self, item):
6         """Add an item to the top of the stack."""
7         self.items.append(item)
8
9     def pop(self):
10        """Remove and return the top item of the stack. Raises IndexError if empty."""
11        if self.is_empty():
12            raise IndexError("Pop from empty stack")
13        return self.items.pop()
14
15    def peek(self):
16        """Return the top item of the stack without removing it. Raises IndexError if empty."""
17        if self.is_empty():
18            raise IndexError("Peek from empty stack")
19        return self.items[-1]
20
21    def is_empty(self):
22        """Check if the stack is empty."""
23        return len(self.items) == 0
24
25
26
27
28
29
30
31
32
33
34
35
36
```

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBOBXDLA/AL LAB 11.1.py"
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBOBXDLA/AL LAB 11.1.py"
20
20
False
10
True
PS C:\Users\kadal>
```

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBOBXDLA/AL LAB 11.1.py"
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBOBXDLA/AL LAB 11.1.py"
20
20
False
10
True
PS C:\Users\kadal>
```

EXPLANATION:

A **stack** is a linear data structure that follows the **LIFO** principle — **Last In, First Out**. Think of it like a stack of plates:

- You add (push) a plate to the top.
- You remove (pop) the top plate first.
- You can peek at the top plate without removing it.
- You can check if the stack is empty.

TASK 2:

Queue Implementation

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

Sample Input Code:

```
class Queue: pass.
```

PROMPT:

Generate python code and queue Implementation

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

Sample Input Code:

```
class Queue: pass.
```

CODE & OUTPUT:

The screenshot shows the Microsoft Visual Studio Code interface. The left sidebar displays the Explorer, showing a folder named 'AI LAB 11.1 T2' containing several files. The main editor window contains the following Python code for a Queue class:

```
1  class Queue:
2      def __init__(self):
3          self.items = []
4
5      def enqueue(self, item):
6          """Add an item to the end of the queue."""
7          self.items.append(item)
8
9      def dequeue(self):
10         """Remove and return the front item of the queue. Raises IndexError if empty."""
11         if self.is_empty():
12             raise IndexError("Dequeue from empty queue")
13         return self.items.pop(0)
14
15     def peek(self):
16         """Return the front item without removing it. Raises IndexError if empty."""
17         if self.is_empty():
18             raise IndexError("Peek from empty queue")
19         return self.items[0]
20
21     def is_empty(self):
22         """Check if the queue is empty."""
23         return len(self.items) == 0
```

Below the code, the terminal window shows the execution of the script:

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/Windows/NetCache/IE/JBO0XDLA/AI LAB 11.1 T2.py
1
1
False
2
3
True
PS C:\Users\kadal>
```

The status bar at the bottom indicates the file is saved with Python 3.11.9.

This screenshot is identical to the one above, showing the same Python code for a Queue class and its execution output in the terminal. The terminal shows the same sequence of values (1, 1, False, 2, 3, True) printed to the console. The status bar at the bottom indicates the file is saved with Python 3.11.9.

EXPLANATION:



Explanation

Method	Description	Time Complexity
<code>__init__</code>	Initializes an empty list to store queue elements	$O(1)$
<code>enqueue()</code>	Adds an item to the end of the list (rear of the queue)	$O(1)$
<code>dequeue()</code>	Removes and returns the first item (front of the queue)	$O(n)$
<code>peek()</code>	Returns the first item without removing it	$O(1)$
<code>is_empty()</code>	Checks if the queue is empty	$O(1)$

⚠ Note: `dequeue()` uses `pop(0)`, which is $O(n)$ because it shifts all remaining elements. For better performance, you can use `collections.deque`.

TASK 3:

Linked List

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

Sample Input Code:

```
class Node: pass.
```

PROMPT:

Generate python code and linked List

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

Sample Input Code:

class Node: pass.

CODE & OUTPUT:

The screenshot shows the Microsoft Visual Studio Code interface. The Explorer sidebar indicates 'NO FOLDER OPENED'. The terminal window displays the following Python code for a linked list:

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class LinkedList:
7     def __init__(self):
8         self.head = None
9
10    def insert(self, data):
11        """Insert a new node at the end of the list."""
12        new_node = Node(data)
13        if self.head is None:
14            self.head = new_node
15            return
16        current = self.head
17        while current.next:
18            current = current.next
19        current.next = new_node
20
21    def display(self):
22        """Display the contents of the linked list."""
23        current = self.head
```

Below the code, the terminal shows the command being run and its output:

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBO00XDLA/AI LAB 11.1 T3.py"
10 -> 20 -> 30 -> None
PS C:\Users\kadal>
```

The screenshot shows the Microsoft Visual Studio Code interface. The Explorer sidebar indicates 'NO FOLDER OPENED'. The terminal window displays the continuation of the Python code for a linked list:

```
19    current.next = new_node
20
21    def display(self):
22        """Display the contents of the linked list."""
23        current = self.head
24        if current is None:
25            print("Linked List is empty.")
26            return
27        while current:
28            print(current.data, end=" -> ")
29            current = current.next
30        print("None")
31 ll = LinkedList()
32 ll.insert(10)
33 ll.insert(20)
34 ll.insert(30)
35 ll.display() # Output: 10 -> 20 -> 30 -> None
36
```

Below the code, the terminal shows the command being run and its output:

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBO00XDLA/AI LAB 11.1 T3.py"
10 -> 20 -> 30 -> None
PS C:\Users\kadal>
```

EXPLANATION:

- Represents each element in the list.
- `data` : stores the value.
- `next` : points to the next node (or `None` if it's the last).



`LinkedList` **Class**

- Manages the chain of nodes.
- `head` : reference to the first node.



`insert(data)`

- Creates a new node.
- If the list is empty, sets it as the head.
- Otherwise, traverses to the end and links the new node.

TASK 4:

Binary Search Tree (BST)

Task: Use AI to create a BST with `insert` and `inorder` traversal methods.

Sample Input Code:

```
class BST: pass.
```

PROMPT:

Generate python code and binary Search Tree (BST)

Task: Use AI to create a BST with insert and inorder traversal methods.

Sample Input Code:

```
class BST: pass.
```

CODE & OUTPUT:

The screenshot shows the Microsoft Visual Studio Code interface. In the center, there is a code editor with the following Python code:

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.left = None
5         self.right = None
6
7 class BST:
8     def __init__(self):
9         self.root = None
10
11     def insert(self, data):
12         """Insert a new node into the BST."""
13         if self.root is None:
14             self.root = Node(data)
15         else:
16             self._insert_recursive(self.root, data)
17
18     def _insert_recursive(self, current, data):
19         if data < current.data:
20             if current.left is None:
21                 current.left = Node(data)
22             else:
23                 self._insert_recursive(current.left, data)
```

Below the code editor, the terminal window displays the output of running the script:

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBO0XDLA/AI LAB 11.1 T4.py"
In-order Traversal: [28, 30, 40, 50, 60, 70, 80]
PS C:\Users\kadal>
```

The screenshot shows the Microsoft Visual Studio Code interface. In the center, there is a code editor with the following Python code:

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.left = None
5         self.right = None
6
7 class BST:
8     def __init__(self):
9         self.root = None
10
11     def in_order_traversal(self):
12         result = []
13         self._in_order_recursive(self.root, result)
14         return result
15
16     def _in_order_recursive(self, node, result):
17         if node:
18             self._in_order_recursive(node.left, result)
19             result.append(node.data)
20             self._in_order_recursive(node.right, result)
21
22 tree = BST()
23 tree.insert(50)
24 tree.insert(30)
25 tree.insert(70)
26 tree.insert(20)
27 tree.insert(40)
28 tree.insert(60)
29 tree.insert(80)
30
31 print("In-order Traversal:", tree.in_order_traversal())
32 # Output: In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
33
```

Below the code editor, the terminal window displays the output of running the script:

```
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/JBO0XDLA/AI LAB 11.1 T4.py"
In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
PS C:\Users\kadal>
```

EXPLANATION:

- Adds a new value to the tree.
- If the tree is empty, it becomes the root.
- Otherwise, it uses `_insert_recursive()` to find the correct position:
 - If `data < current.data` : go left.
 - If `data > current.data` : go right.
 - If equal: skip (no duplicates).

in_order_traversal()

- Returns a sorted list of values.
- Uses `_in_order_recursive()` :
 - Traverse left subtree.
 - Visit current node.
 - Traverse right subtree.

TASK 5:

Hash Table

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

Sample Input Code: class HashTable: pass.

PROMPT:

Generate python code and hash Table Task:
Use AI to implement a hash table with

basic insert, search, and delete methods.

Sample Input Code: class HashTable: pass.

CODE & OUTPUT:

```
1  class HashTable:
2      def __init__(self, size=10):
3          self.size = size
4          self.table = [[] for _ in range(size)]
5
6      def _hash_function(self, key):
7          """Simple hash function using modulo."""
8          return hash(key) % self.size
9
10     def insert(self, key, value):
11         """Insert a key-value pair into the hash table."""
12         index = self._hash_function(key)
13         for i, (k, v) in enumerate(self.table[index]):
14             if k == key:
15                 self.table[index][i] = (key, value) # Update existing key
16                 return
17         self.table[index].append((key, value))
18
19     def search(self, key):
20         """Search for a value by key. Returns value or None."""
21         index = self._hash_function(key)
22         for k, v in self.table[index]:
23             if k == key:
24                 return v
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

d:\inetcache\IE\3B000XDLA\AI LAB 11.1 TS.py" 200

Bucket 0: []
Bucket 1: [('orange', 300)]
Bucket 2: []
Bucket 3: [('banana', 200)]
Bucket 4: []
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: []
Bucket 9: []

PS C:\Users\kadal>

Spaces: 4 UTF-8 (Python 3.11.9 (Microsoft Store) 15:39:43 16-09-2025

31°C Mostly cloudy

```
1  class HashTable:
2      def delete(self, key):
3          """Delete a key-value pair from the hash table."""
4          index = self._hash_function(key)
5          for i, (k, _) in enumerate(self.table[index]):
6              if k == key:
7                  del self.table[index][i]
8                  return True
9          return False
10
11      def display(self):
12          """Display the contents of the hash table."""
13          for i, bucket in enumerate(self.table):
14              print(f"Bucket {i}: {bucket}")
15
16      ht = HashTable()
17      ht.insert("apple", 100)
18      ht.insert("banana", 200)
19      ht.insert("orange", 300)
20
21      print(ht.search("banana")) # Output: 200
22      ht.delete("apple")
23      ht.display()
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
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

d:\inetcache\IE\3B000XDLA\AI LAB 11.1 TS.py" 200

Bucket 0: []
Bucket 1: [('orange', 300)]
Bucket 2: []
Bucket 3: [('banana', 200)]
Bucket 4: []
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: []
Bucket 9: []

PS C:\Users\kadal>

Spaces: 4 UTF-8 (Python 3.11.9 (Microsoft Store) 15:39:55 16-09-2025

31°C Mostly cloudy

EXPLANATION:



`_hash_function(self, key)`

- Uses Python's built-in `hash()` function.
- Applies modulo to ensure the index fits within the table size.



`insert(self, key, value)`

- Computes the index using the hash function.
- Checks if the key already exists in the bucket:
 - If yes, updates the value.
 - If no, appends the new `(key, value)` pair.



`search(self, key)`

- Computes the index and scans the bucket.
- Returns the value if the key is found, otherwise returns `None`.

TASK 6:

Graph Representation

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

Sample Input Code: class

Graph:

pass.

PROMPT:

Generate python code and graph
Representation

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

Sample Input Code:

```
class Graph: pass.
```

CODE & OUTPUT:

The screenshot shows the Microsoft Visual Studio Code interface. In the center, there is a code editor window displaying Python code for a graph class. The code defines a Graph class with methods for adding edges and displaying the adjacency list. It also includes a main block that creates a graph with four nodes (A, B, C, D) and their connections. Below the code editor is a terminal window showing the command-line output of running the script. The terminal shows the Python interpreter executing the code and printing the resulting adjacency lists for each node.

```
1 class Graph:
2     def __init__(self):
3         self.adj_list = {}
4
5     def add_edge(self, u, v):
6         """Add an edge from vertex u to vertex v (undirected by default)."""
7         if u not in self.adj_list:
8             self.adj_list[u] = []
9         if v not in self.adj_list:
10            self.adj_list[v] = []
11        self.adj_list[u].append(v)
12        self.adj_list[v].append(u) # Remove this line for directed graph
13
14    def display(self):
15        """Display the adjacency list of the graph."""
16        for vertex in self.adj_list:
17            print(f"\n{vertex} -> {self.adj_list[vertex]}")
18
19 g = Graph()
20 g.add_edge("A", "B")
21 g.add_edge("A", "C")
22 g.add_edge("B", "D")
23 g.add_edge("C", "D")
24 g.display()
25
26 # Output:
27 # A -> ['B', 'C']
28 # B -> ['A', 'D']
29 # C -> ['A', 'D']
30 # D -> ['B', 'C']
```

PS C:\Users\kadal> & c:/users/kadal/appdata/local/microsoft/windowsapps/python3.11.exe "c:/users/kadal/appdata/local/microsoft/windows/inetcache/ie/jbo00x0la/ai_lab_11.1_t6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']

This screenshot is identical to the one above, showing the Microsoft Visual Studio Code interface with the same Python code for a graph class and its execution output in the terminal. The code defines a Graph class with methods for adding edges and displaying the adjacency list. It also includes a main block that creates a graph with four nodes (A, B, C, D) and their connections. Below the code editor is a terminal window showing the command-line output of running the script. The terminal shows the Python interpreter executing the code and printing the resulting adjacency lists for each node.

```
1 class Graph:
2     def __init__(self):
3         self.adj_list = {}
4
5     def add_edge(self, u, v):
6         """Add an edge from vertex u to vertex v (undirected by default)."""
7         if u not in self.adj_list:
8             self.adj_list[u] = []
9         if v not in self.adj_list:
10            self.adj_list[v] = []
11        self.adj_list[u].append(v)
12        self.adj_list[v].append(u) # Remove this line for directed graph
13
14    def display(self):
15        """Display the adjacency list of the graph."""
16        for vertex in self.adj_list:
17            print(f"\n{vertex} -> {self.adj_list[vertex]}")
18
19 g = Graph()
20 g.add_edge("A", "B")
21 g.add_edge("A", "C")
22 g.add_edge("B", "D")
23 g.add_edge("C", "D")
24 g.display()
25
26 # Output:
27 # A -> ['B', 'C']
28 # B -> ['A', 'D']
29 # C -> ['A', 'D']
30 # D -> ['B', 'C']
```

PS C:\Users\kadal> & c:/users/kadal/appdata/local/microsoft/windowsapps/python3.11.exe "c:/users/kadal/appdata/local/microsoft/windows/inetcache/ie/jbo00x0la/ai_lab_11.1_t6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']

EXPLANATION:

- Initializes an empty list called `heap`.
- This list will store tuples of `(priority, item)`.

`insert(priority, item)`

- Uses `heapq.heappush()` to add a tuple to the heap.
- The heap maintains order based on the **priority** (lowest number = highest priority).

`remove()`

- Uses `heapq.heappop()` to remove and return the item with the **lowest priority value**.
- Raises an error if the queue is empty.

`peek()`

- Returns the item with the highest priority without removing it.

TASK 7:

Priority Queue

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

Sample Input Code: class

PriorityQueue: pass.

PROMPT:

Generate a python code and priority Queue

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

Sample Input Code: class

PriorityQueue: pass.

CODE & OUTPUT:

The screenshot shows the Microsoft Visual Studio Code interface. The left sidebar has 'OPEN EDITORS' and 'NO FOLDER OPENED'. The main editor area contains Python code for a PriorityQueue class using the heapq module. The code includes methods for inserting items with priority, removing the highest priority item, peeking at the highest priority item without removing it, and checking if the queue is empty. The code is as follows:

```
import heapq

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

    def insert(self, priority, item):
        """Insert an item with a given priority."""
        heapq.heappush(self.heap, (priority, item))

    def remove(self):
        """Remove and return the item with the highest priority (lowest number)."""
        if self.is_empty():
            raise IndexError("Remove from empty priority queue")
        return heapq.heappop(self.heap)[1]

    def peek(self):
        """Return the item with the highest priority without removing it."""
        if self.is_empty():
            raise IndexError("Peek from empty priority queue")
        return self.heap[0][1]

    def is_empty(self):
        pass
```

The bottom terminal window shows the command `python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/JBO0XDLA/AI LAB 11.1 T6.py"`, followed by the output:

```
Task A
Task A
Priority Queue: ['Task B', 'Task C']
PS C:\Users\kadal>
```

The status bar at the bottom right shows 'Spaces: 4', 'UTF-8', 'Python 3.11.9 (Microsoft Store)', 'ENG IN', '15:43:36', and '16-09-2025'.

The screenshot shows a Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows an open folder named "AI LAB 11.1".
- Code Editor:** Displays Python code for a priority queue. The code includes methods for inserting tasks (PriorityQueue.insert), checking if it's empty (PriorityQueue.is_empty), and displaying its contents (PriorityQueue.display). A sample usage at the bottom demonstrates inserting tasks A, B, and C, then printing the peeked task (A), removing it (Task A), and finally displaying the remaining tasks (B and C).
- Terminal:** Shows the command `python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/J800XNDLA/AI LAB 11.1 T6.py"`. The output of the script is:

```
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\kadal> & c:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/J800XNDLA/AI LAB 11.1 T7.py"
Task A
Task A
Priority Queue: ['Task B', 'Task C']
PS C:\Users\kadal>
```
- Status Bar:** Shows system information including battery level (31%), connection status (No connection), and system time (16:09:2025).

EXPLANATION:

[youtube - Search](#) | [\(M6\) Kondakala Full Song II](#) | [Microsoft Copilot: Your AI compai](#) | +

<https://copilot.microsoft.com/chats/NQ7ncjXvMHQdnGKaS1L>

 Error Control – Correction Techniques

7. What is the difference between error detection and error correction?

Answer:

- **Detection** identifies the presence of errors.
- **Correction** identifies and fixes the errors without retransmission.

8. Explain Hamming Code with an example.

Answer:

Hamming Code adds redundant bits at positions that are powers of 2. These bits help detect and correct single-bit errors.

Example: For 4-bit data `1011`, Hamming code adds 3 parity bits to make it `p1 p2 1 p4 0 1 1`. Parity bits are calculated to detect errors.

9. What is the minimum Hamming distance required to correct single-bit errors?

Message Copilot

 Quick response ▾

+ ⌂

26°C Light rain

Search

ENG IN WiFi 21:42:29 13-09-2023

TASK 8:

Deque

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

Sample Input Code:

```
class DequeDS: pass.
```

PROMPT:

Generate python code and deque Task:

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

Sample Input Code:

```
class DequeDS: pass.
```

CODE &OUTPUT:

EXPLANATION:

- Initializes an empty deque using `collections.deque`, which is optimized for fast appends and pops from both ends.



`add_front(item)`

- Adds an item to the **front** using `appendleft()`.



`add_rear(item)`

- Adds an item to the **rear** using `append()`.



`remove_front()`

- Removes and returns the item from the **front** using `popleft()`.



`remove_rear()`

TASK 9:

AI-Generated Data Structure Comparisons Task:
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 python code and AI-Generated Data Structure Comparisons

Task: 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.
```

CODE & OUTPUT:

A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows multiple tabs: 'AB 11.1 T10.py', 'AI LAB 11.1 T9.0.py', '9.00 Alipy', 'Untitled-4', '9 Alipy 1', 'Untitled-1', 'AI LAB 11.1 T4.py', 'AI LAB 11.1 T5.py', 'AI LAB 11.1 T6.py', and 'AI LAB 11.1 T7.py'. The main editor area contains Python code for a 'Data Structure Comparison Table Example' and a 'Stack Example'. The code defines a 'Stack' class with methods for push and pop operations. The terminal below shows the output of running the code, which includes examples for Queue and Linked List structures. The status bar at the bottom right indicates the file has 107 lines, 4 spaces, and is in UTF-8 encoding.

```
C:\> Users > kadal > AppData > Local > Microsoft > Windows > INetCache > IE > JBO00XDLA > 9.00 Alipy > ...
1 # Data Structure Comparison Table Example
2 def print_data_structure_comparison():
3     table = [
4         ["Data structure", "Insert/Push/Enqueue", "Delete/Pop/Dequeue", "Search/Access", "Peek/Front/End"],
5         ["Stack (List)", "O(1)", "O(1)", "O(n)", "O(1)"],
6         ["Queue (List)", "O(1)", "O(1)", "O(n)", "O(1)", "O(1)"],
7         ["Queue (Deque)", "O(1)", "O(1)", "O(n)", "O(1)", "O(1)"],
8         ["Singly Linked List", "O(1) (at head)", "O(1) (at head)", "O(n)", "O(1) (head)", "O(1)"],
9         ["Doubly Linked List", "O(1) (at ends)", "O(1) (at ends)", "O(n)", "O(1) (ends)", "O(1) (ends)"]
10    ]
11    for row in table:
12        print(" | ".join(row))
13
14 # Stack Example
15 class Stack:
16     def __init__(self):
17         self.items = []
18
19     def push(self, item):
20         self.items.append(item)
21
22     def pop(self):
23         return self.items.pop() if self.items else None
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
Stack pop: 3
Stack peek: 2
Is stack empty? False
Queue Example:
Queue after enqueue: [1, 2, 3]
Queue dequeue: 1
Queue peek: 1
Is queue empty? False
Linked List Example:
Linked List elements: 1 2 3
PS C:\Users\kadal>
0 1 2 3 No connection
31C Mostly cloudy
```

A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows multiple tabs: 'AB 11.1 T10.py', 'AI LAB 11.1 T9.0.py', '9.00 Alipy', 'Untitled-4', '9 Alipy 1', 'Untitled-1', 'AI LAB 11.1 T4.py', 'AI LAB 11.1 T5.py', 'AI LAB 11.1 T6.py', and 'AI LAB 11.1 T7.py'. The main editor area contains Python code for a 'Queue' class and a 'linked List' class. The code defines a 'Queue' class with methods for enqueue, dequeue, peek, and is_empty operations. It also defines a 'Node' class for a linked list and a 'linkedlist' class. The terminal below shows the output of running the code, which includes examples for Queue and Linked List structures. A WhatsApp window is open in the foreground, showing a message from 'CSE II Year B2 (2024-28) -Sheshi'. The status bar at the bottom right indicates the file has 107 lines, 4 spaces, and is in UTF-8 encoding.

```
C:\> Users > kadal > AppData > Local > Microsoft > Windows > INetCache > IE > JBO00XDLA > 9.00 Alipy > ...
32 class Queue:
33     def __init__(self):
34         self.items = []
35
36     def enqueue(self, item):
37         self.items.append(item)
38
39     def dequeue(self):
40         return self.items.pop(0) if self.items else None
41
42     def peek(self):
43         return self.items[0] if self.items else None
44
45     def is_empty(self):
46         return len(self.items) == 0
47
48 # linked List Example
49 class Node:
50     def __init__(self, data):
51         self.data = data
52         self.next = None
53
54 class linkedlist:
55     def __init__(self):
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS
Stack pop: 3
Stack peek: 2
Is stack empty? False
Queue Example:
Queue after enqueue: [1, 2, 3]
Queue dequeue: 1
Queue peek: 1
Is queue empty? False
Linked List Example:
Linked List elements: 1 2 3
PS C:\Users\kadal>
0 1 2 3 No connection
31C Mostly cloudy
```

```

81     stack = Stack()
82     stack.push(1)
83     stack.push(2)
84     stack.push(3)
85     print("Stack after pushes:", stack.items)
86     print("Stack pop:", stack.pop())
87     print("Stack peek:", stack.peek())
88     print("Is stack empty?", stack.is_empty())
89
90     print("\nQueue Example:")
91     queue = Queue()
92     queue.enqueue(1)
93     queue.enqueue(2)
94     queue.enqueue(3)
95     print("Queue after enqueues:", queue.items)
96     print("Queue dequeue:", queue.dequeue())
97     print("Queue peek:", queue.peek())
98     print("Is queue empty?", queue.is_empty())
99
100    print("\nLinked List Example:")
101    ll = LinkedList()
102    ll.insert(1)
103    ll.insert(2)

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

Stack pop: 3
Stack peek: 2
Is stack empty? False

Queue Example:
Queue after enqueues: [1, 2, 3]
Queue dequeue: 1
Queue peek: 2
Is queue empty? False

Linked List Example:
Linked List elements: 1 2 3
PS C:\Users\kadal>

31°C Mostly cloudy Search File Home Recent View Insert Editor Run Terminal Taskbar Help Python 3.11.9 (Microsoft Store)

EXPLANATION:

`def print_data_structure_comparison():`

- Defines a function that prints a formatted comparison table.

`table = [...]`

- A list of lists, where each inner list represents a row in the table.
- The first row is the header: column titles like "Insert", "Delete", etc.
- Each subsequent row compares a specific data structure.

`" | ".join(row)`

- Joins each element in the row with `" | "` to mimic a table format.
- This makes the output readable and aligned like a markdown-style table.

TASK 10:

Task Description #10 Real-Time Application Challenge – Choose the Right Data Structure Scenario:

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

1. StudentAttendanceTracking–Dailylog of students entering/exiting the campus.
2. EventRegistrationSystem–Manage participants in events with quick search and removal.
3. LibraryBookBorrowing–Keeptrackof available books and their due dates.
4. BusSchedulingSystem–Maintainbus routes and stop connections.
5. CafeteriaOrderQueue–Servestudents 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 python code and task Description
#10 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. EventRegistrationSystem—Manage participants in events with quick search and removal.
3. LibraryBookBorrowing—Keep track of available books and their due dates.
4. BusSchedulingSystem—Maintain bus routes and stop connections.
5. CafeteriaOrderQueue—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.

CODE & OUTPUT:

The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows multiple Python files in the AI LAB 11.1 folder, including AI LAB 11.1 T3.py, AI LAB 11.1 T10.py, AI LAB 11.1 T9.0.py, Untitled-1, AI LAB 11.1 T4.py, AI LAB 11.1 T5.py, AI LAB 11.1 T6.py, and AI LAB 11.1 T7.py.
- Code Editor:** Displays the content of AI LAB 11.1 T10.py. The code defines a `CafeteriaQueue` class that uses `collections.deque` for efficient queue operations. It includes methods for adding orders, serving the next order, and displaying the queue.
- Terminal:** Shows the output of running the code. It prints messages indicating order placement, current queue size, and attempts to serve from an empty queue.
- Status Bar:** Shows the Python extension version (3.11.9), file path (C:\Users\kadal\...), and date (16-09-2025).

```

import collections

class CafeteriaQueue:
    """
    Implements a cafeteria order queue using a deque for efficient
    First-In, First-Out (FIFO) operations.
    """

    def __init__(self):
        # Using a deque from the collections module for an efficient queue
        self.orders = collections.deque()
        print("Cafeteria order queue system initialized.")
        print("-" * 40)

    def add_order(self, student_id, order_details):
        """Adds a new order to the end of the queue."""
        self.orders.append((student_id, order_details))
        print(f"Order for student {student_id} has been placed.")
        print(f"Current queue size: {len(self.orders)}")

    def serve_next_order(self):
        """Serves the next order from the front of the queue."""
        if not self.orders:
            print("X The queue is empty. No orders to serve.")

    def display_queue(self):
        """Displays all orders currently in the queue."""
        if not self.orders:
            print("The queue is currently empty.")
            return

        print("\nCurrent Orders in Queue:")
        for i, order in enumerate(self.orders):
            print(f"{i+1}. Student ID: {order[0]}, Order: {order[1]}")
        print("-" * 40)

# Main program to demonstrate the Cafeteria Queue

```

The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows multiple Python files in the AI LAB 11.1 folder, including AI LAB 11.1 T3.py, AI LAB 11.1 T10.py, AI LAB 11.1 T9.0.py, Untitled-1, AI LAB 11.1 T4.py, AI LAB 11.1 T5.py, AI LAB 11.1 T6.py, and AI LAB 11.1 T7.py.
- Code Editor:** Displays the content of AI LAB 11.1 T10.py. The code defines a `CafeteriaQueue` class that uses a custom queue implementation. It includes methods for adding orders, serving the next order, and displaying the queue.
- Terminal:** Shows the output of running the code. It prints messages indicating order placement, current queue size, and attempts to serve from an empty queue.
- Status Bar:** Shows the Python extension version (3.11.9), file path (C:\Users\kadal\...), and date (16-09-2025).

```

class CafeteriaQueue:
    def __init__(self):
        self.orders = []
        self.served_order = None

    def add_order(self, student_id, order_details):
        self.orders.append((student_id, order_details))

    def serve_next_order(self):
        if not self.orders:
            print("X The queue is empty. No orders to serve.")

        self.served_order = self.orders.pop(0)
        student_id = self.served_order[0]
        order_details = self.served_order[1]

        print(f"★ Serving order for Student {student_id}: {order_details}")
        print(f"Remaining orders in queue: {len(self.orders)}")
        return self.served_order

    def display_queue(self):
        """Displays all orders currently in the queue."""
        if not self.orders:
            print("The queue is currently empty.")
            return

        print("\nCurrent Orders in Queue:")
        for i, order in enumerate(self.orders):
            print(f"{i+1}. Student ID: {order[0]}, Order: {order[1]}")
        print("-" * 40)

# Main program to demonstrate the Cafeteria Queue

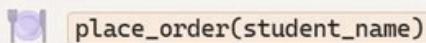
```

The screenshot shows a Microsoft Visual Studio Code (VS Code) interface with the following details:

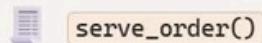
- File Explorer:** Shows a folder structure under "C:\Users\kadal\AppData\Local\Microsoft\Windows\INetCache\IE\JBOOXDIA". The file "AI LAB 11.1 T10.py" is the active editor.
- Code Editor:** Displays Python code for a cafeteria queue system. The code includes functions for serving students, displaying the queue, adding new orders, and attempting to serve from an empty queue.
- Terminal:** Shows the output of running the script. It displays messages about placing an order for student 104, serving remaining orders for students 103 and 104, and attempting to serve from an empty queue which fails because there are no orders.
- Status Bar:** Shows connection status ("No connection"), file path ("PS C:\Users\kadal\10"), and system information (Python 3.11.9, Microsoft Store).

EXPLANATION:

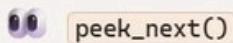
- Initializes an empty list `queue` to store student names.



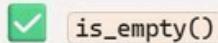
- Adds a student to the end of the queue using `append()` – O(1) time.



- Removes and returns the first student using `pop(0)` — $O(n)$ time due to shifting.
 - Raises an error if the queue is empty.



- Returns the first student without removing them – O(1) time.



- Checks if the queue is empty – O(1) time.

