# LAB ASSIGNMENT 11.1

**NAME : DONTHI MEGHANA**

**BATCH :05**

**Task Description #1 – Stack Implementation**

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

pass

Expected Output:

• A functional stack implementation with all required methods and

docstrings **PROMPT:-**

[?]  push(item) → To insert an element into the stack. [?]  pop()

→ To remove and return the top element of the stack. [?]

peek() → To return the top element without removing it. [?]

is_empty() → To check whether the stack is empty or not.

**CODE:-**

```
Lab_11_1.py > ...
1    class Stack:
2        def __init__(self):
3            self.items = []
4
5        def push(self, item):
6            self.items.append(item)
7
8        def pop(self):
9            if not self.is_empty():
10               return self.items.pop()
11           else:
12               raise IndexError("pop from empty stack")
13
14       def peek(self):
15           if not self.is_empty():
16               return self.items[-1]
17           else:
18               raise IndexError("peek from empty stack")
19
20       def is_empty(self):
21           return len(self.items) == 0
22
23   # Example usage:
24   if __name__ == "__main__":
25       s = Stack()
26
27       s.push(20)
28       print(s.peek())      # Output: 2
29       print(s.pop())       # Output: 2
30       print(s.is_empty())
```

**OUTPUT:**

```
[Running] python -u "c:\Users\PEDDAPELLI ANUSHA\OneDrive\Desktop\Btech.2nd yr\AI\Lab_11_1.py"
20
20
True
```

**OBSERVATION:-**

[?]  Initially the stack is empty.

[?]      After pushing 10, 20, 30, the top is 30 (LIFO order).

⬜    First pop removes 30 → next top is 20.

⬜    Next pops remove 20 and 10 sequentially.

⬜    Trying to pop from an empty stack shows a warning message.

⬜    Final check confirms stack is empty again.

**Task Description #2 – Queue Implementation Task:** Use
AI to implement a Queue using Python lists.
Sample Input Code:
class Queue: pass
Expected Output:
• FIFO-based queue class with enqueue, dequeue, peek, and size method.

**PROMPT:-**

Implement a Queue using Python lists. Test the Queue by enqueuing two elements,

peeking at the front, dequeuing one element, and checking if the queue is empty. CODE:-

```
ab_11_1.py > ...
    class Queue:
        def __init__(self):
            self.items = []

        def enqueue(self, item):
            self.items.append(item)

        def dequeue(self):
            if not self.is_empty():
                return self.items.pop(0)
            else:
                raise IndexError("dequeue from empty queue")

        def peek(self):
            if not self.is_empty():
                return self.items[0]
            else:
                raise IndexError("peek from empty queue")

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

    # Example usage:
    if __name__ == "__main__":
        q = Queue()
        q.enqueue(10)
        q.enqueue(20)
        print(q.peek())        # Output: 10
        print(q.dequeue())     # Output: 10
        print(q.is_empty())    # Output: False
```

**OBSERVATION:-**

The code defines a Queue class with methods for enqueue, dequeue, peek, and
is_empty.
**Task Description #3 – Linked List**
**Task:** Use AI to generate a Singly Linked List with insert and display methods.
Sample Input Code: class
Node:
pass class
LinkedList:
pass

Expected Output:

• A working linked list implementation with clear method documentation

**PROMPT:-**

Implement a Queue using Python lists. Test the Queue by enqueuing two elements,

peeking at the front, dequeuing one element, and checking if the queue is empty. CODE:-

```
Lab_11_1.py > ...
62  class Node:
63      def __init__(self, data):
64          self.data = data
65          self.next = None
66
67  class SinglyLinkedList:
68      def __init__(self):
69          self.head = None
70
71      def insert(self, data):
72          new_node = Node(data)
73          if not self.head:
74              self.head = new_node
75          else:
76              current = self.head
77              while current.next:
78                  current = current.next
79              current.next = new_node
80
81      def display(self):
82          current = self.head
83          while current:
84              print(current.data, end=" ")
85              current = current.next
86          print()
87
88  # Example usage:
89  if __name__ == "__main__":
90      sll = SinglyLinkedList()
91      sll.insert(5)
92      sll.insert(10)
93      sll.insert(15)
94      sll.display()   # Output: 5 10 15
```

OUTPUT:-

```
5 10 15
```

**OBSERVATION:-**
The code correctly implements a Queue with enqueue, dequeue, peek, and
is_empty methods.

**Task Description #4 – Binary Search Tree (BST)**
**Task:** Use AI to create a BST with insert and in-order traversal methods.
Sample Input Code: class
BST:
pass
**PROMPT:-**

Create a Binary Search Tree (BST) with insert and in-order traversal methods. Test the BST
by inserting several elements and displaying them in sorted order using in-order
traversal.

CODE:-

```
class BSTNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, data):
        if not self.root:
            self.root = BSTNode(data)
        else:
            self._insert(self.root, data)

    def _insert(self, node, data):
        if data < node.data:
            if node.left:
                self._insert(node.left, data)
            else:
                node.left = BSTNode(data)
        else:
            if node.right:
                self._insert(node.right, data)
            else:
                node.right = BSTNode(data)

    def inorder(self):
        self._inorder(self.root)
        print()

    def _inorder(self, node):
        if node:
            self._inorder(node.left)
            print(node.data, end=" ")
            self._inorder(node.right)
```

```
            print(node.data, end=" ")
            self._inorder(node.right)

# Example usage:
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(10)
    bst.insert(5)
    bst.insert(15)
    bst.insert(7)
    bst.inorder()   # Output: 5 7 10 15
```

OUTPUT:-

```
[Running] python -u "c:\Users\PEDDAPELLI ANUSHA\OneDrive\Desktop\Btech.2nd yr\AI\Lab_11_1.py"
5 7 10 15

[Done] exited with code=0 in 0.331 seconds
```

**OBSERVATION:-**

The code defines a BST with insert and inorder methods.

**Task Description #5 – Hash Table**

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

Sample Input Code: class HashTable:

pass

Expected Output:

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

PROMPT:-

**Prompt:**

Implement a hash table in Python with basic insert, search, and delete methods.
Demonstrate its usage by inserting key-value pairs, searching for a key, and deleting a key.

CODE:-

```python
class HashTable:
    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):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        self.table[idx].append((key, value))

    def search(self, key):
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                del self.table[idx][i]
                return True
        return False

if __name__ == "__main__":
    ht = HashTable()
    ht.insert("apple", 100)
    ht.insert("banana", 200)
    print(ht.search("apple"))
    ht.delete("apple")
    print(ht.search("apple"))
```

OBSERVATION:-

The hash table stores key-value pairs and supports insert, search, and delete operations.

- After inserting "apple" and "banana", searching for "apple" returns 100.

- After deleting "apple", searching for "apple" returns None. This confirms the hash table works as expected.

–

**Task Description #6          Graph Representation**

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

Sample Input Code:

class Graph: pass

Expected Output:

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

**PROMPT:-**

Implement a graph using an adjacency list in Python. Add methods to add edges and display the adjacency list. Demonstrate by creating a graph, adding edges, and displaying its structure.

CODE;-

```python
class HashTable:
    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):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        self.table[idx].append((key, value))

    def search(self, key):
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                del self.table[idx][i]
                return True
        return False
if __name__ == "__main__":
    ht = HashTable()
    ht.insert("apple", 100)
    ht.insert("banana", 200)
    print(ht.search("apple"))
    ht.delete("apple")
    print(ht.search("apple"))
```

**OBSERVATION:-**

The graph is represented using a dictionary where each key is a node and its value is a list of adjacent nodes.
After adding edges, the display method prints the adjacency list, showing the structure of the graph as expected.

**Task Description #7          Priority Queue**

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

–

Sample Input Code: class
PriorityQueue:

pass

Expected Output:

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

**PROMPT:-**

Implement a priority queue using Python's heapq module. Demonstrate by inserting

elements with priorities and removing them in priority order. **CODE:-**

```python
import heapq

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

    def insert(self, priority, item):
        heapq.heappush(self.heap, (priority, item))

    def remove(self):
        if self.heap:
            return heapq.heappop(self.heap)[1]
        else:
            raise IndexError("remove from empty priority queue")

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

# Example usage:
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.insert(2, "task2")
    pq.insert(1, "task1")
    pq.insert(3, "task3")
    while not pq.is_empty():
        print(pq.remove())
```

**OUTPUT:-**

```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b.11.1.7.py
Peek: 5
Pop: 5
Peek after pop: 10
Pop: 10
Pop: 15
Pop: 20
Is empty: True
Error: pop from empty priority queue
PS C:\Users\Administrator\OneDrive\ai> []
```

**OBSERVATION:-**

The priority queue stores items with their priorities.
When removing, items are returned in order of increasing priority (task1, task2, task3),
confirming correct behavior.

**Task Description #8        Deque**

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

–
Sample Input Code: class
DequeDS:
pass
Expected Output:
• Insert and remove from both ends with docstrings.

**PROMPT:-**

Implement a double-ended queue (deque) using Python's collections.deque. Demonstrate by adding and removing elements from both ends and displaying the result.

**CODE:-**

```python
from collections import deque

class DoubleEndedQueue:
    def __init__(self):
        self.deque = deque()

    def add_front(self, item):
        self.deque.appendleft(item)

    def add_rear(self, item):
        self.deque.append(item)

    def remove_front(self):
        if self.deque:
            return self.deque.popleft()
        else:
            raise IndexError("remove from empty deque (front)")

    def remove_rear(self):
        if self.deque:
            return self.deque.pop()
        else:
            raise IndexError("remove from empty deque (rear)")

    def display(self):
        print(list(self.deque))

# Example usage:
if __name__ == "__main__":
    dq = DoubleEndedQueue()
    dq.add_rear(10)
    dq.add_front(20)
    dq.add_rear(30)
    dq.display()            # Output: [20, 10, 30]
    print(dq.remove_front()) # Output: 20
    print(dq.remove_rear())  # Output: 30
    dq.display()            # Output: [10]
```

**OUTPUT:-**

**OBSERVATION:-**

The double-ended queue allows insertion and removal from both ends.
After adding and removing elements, the display shows the correct order, confirming the deque works as expected.

**Task Description #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 Expected
Output:
• A markdown table with structure names, operations, and complexities. **PROMPT:-**

generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.
**CODE:-**

```python
def add_numbers(a: int, b: int) -> int:

    return a + b


def is_even(number: int) -> bool:

    return number % 2 == 0


def greet_user(name: str, greeting: str = "Hello") -> str:

    return f"{greeting}, {name}!"

def calculate_area(length: float, width: float) -> float:
    return length * width

def factorial(n: int) -> int:
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

**OUTPUT:-**



```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b.11.1.9.py
+--------------------------+----------+----------+----------+----------+----------+
| Data Structure           | Insert   | Delete   | Search   | Access   | Update   |
+--------------------------+----------+----------+----------+----------+----------+
| Stack                    | O(1)     | O(1)     | O(n)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
| Queue                    | O(1)     | O(1)     | O(n)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
| Singly Linked List       | O(1)     | O(1)     | O(n)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
| Doubly Linked List       | O(1)     | O(1)     | O(n)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
| Hash Table               | O(1)     | O(1)     | O(1)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
| Binary Search Tree (BST) | O(log n) | O(log n) | O(n)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
| Heap (Priority Queue)    | O(log n) | O(log n) | O(n)     | O(n)     | O(n)     |
| Doubly Linked List       | O(1)     | O(1)     | O(n)     | O(n)     | O(n)     |
+--------------------------+----------+----------+----------+----------+----------+
```

**OBSERVATION:-**

This table summarizes the time complexities for common operations across various data structures.

- Stacks and queues (using lists) are fast for push/pop but slow for dequeue from the front.

- Deques and linked lists offer O(1) operations at the ends.

- Hash tables provide average O(1) search, insert, and delete.

- Balanced BSTs and heaps offer logarithmic time for insert and delete.

- Graph adjacency lists are efficient for edge operations. This helps in choosing the right data structure for specific tasks based on performance needs.