

# **2303a51449**

Batch:-03

## **Task 1 – Stack Implementation**

### **AI Prompt Used:**

Design and implement a Stack class in Python with push, pop, peek and is\_empty methods including proper documentation.

### **Python Implementation:**

```
class Stack:  
    """Stack implementation using Python list"""\n  
  
    def __init__(self):  
        self.items = []\n  
  
    def push(self, item):  
        """Add item to the top of the stack"""  
        self.items.append(item)\n  
  
    def pop(self):  
        """Remove and return the top item"""  
        if not self.is_empty():  
            return self.items.pop()  
        return "Stack is empty"\n  
  
    def peek(self):  
        """Return top item without removing it"""  
        if not self.is_empty():  
            return self.items[-1]  
        return "Stack is empty"\n  
  
    def is_empty(self):  
        """Check if stack is empty"""  
        return len(self.items) == 0\n  
  
# Example Usage  
s = Stack()
```

```
s.push(10)
s.push(20)
print(s.peek())
print(s.pop())
print(s.is_empty())
```

### **Code Explanation:**

The Stack class uses a Python list to store elements. Push adds elements, pop removes the last element, peek checks the top element, and is\_empty verifies whether the stack has elements.

### **Sample Output:**

```
20
20
False
```

## **Task 2 – Queue Implementation**

### **AI Prompt Used:**

Create a Queue data structure in Python following FIFO principle with enqueue, dequeue, front and size methods.

### **Python Implementation:**

```
class Queue:
    """Queue implementation using list"""

    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)
        return "Queue is empty"
```

```
def front(self):
    if not self.is_empty():
        return self.items[0]
    return "Queue is empty"

def size(self):
    return len(self.items)

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

# Example Usage
q = Queue()
q.enqueue(5)
q.enqueue(15)
print(q.front())
print(q.dequeue())
print(q.size())
```

### **Code Explanation:**

Queue follows FIFO principle. Enqueue adds to rear, dequeue removes from front. Front returns first element, and size returns number of elements.

### **Sample Output:**

```
5
5
1
```

## **Task 3 – Singly Linked List**

### **AI Prompt Used:**

Build a singly linked list in Python supporting node creation, insertion at end, and traversal.

### **Python Implementation:**

```
class Node:
    def __init__(self, data):
        self.data = data
```

```

        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

# Example Usage
ll = LinkedList()
ll.insert(1)
ll.insert(2)
ll.insert(3)
ll.display()

```

### **Code Explanation:**

Each node contains data and a next pointer. Insert adds node at end. Display traverses from head until None.

### **Sample Output:**

1 -> 2 -> 3 -> None

## **Task 4 – Binary Search Tree**

### **AI Prompt Used:**

Implement a Binary Search Tree in Python with insertion and in-order traversal.

## **Python Implementation:**

```
class BSTNode:  
    def __init__(self, key):  
        self.left = None  
        self.right = None  
        self.val = key  
  
class BST:  
    def insert(self, root, key):  
        if root is None:  
            return BSTNode(key)  
        if key < root.val:  
            root.left = self.insert(root.left, key)  
        else:  
            root.right = self.insert(root.right, key)  
        return root  
  
    def inorder(self, root):  
        if root:  
            self.inorder(root.left)  
            print(root.val, end=" ")  
            self.inorder(root.right)  
  
# Example Usage  
bst = BST()  
root = None  
root = bst.insert(root, 50)  
root = bst.insert(root, 30)  
root = bst.insert(root, 70)  
bst.inorder(root)
```

## **Code Explanation:**

BST maintains sorted property. Left subtree contains smaller values and right subtree contains larger values. In-order traversal prints sorted output.

## **Sample Output:**

30 50 70

## **Task 5 – Hash Table Implementation**

### **AI Prompt Used:**

Create a hash table in Python with collision handling using chaining method.

### Python Implementation:

```
class HashTable:  
    def __init__(self, size):  
        self.size = size  
        self.table = [[] for _ in range(size)]  
  
    def hash_function(self, key):  
        return hash(key) % self.size  
  
    def insert(self, key, value):  
        index = self.hash_function(key)  
        for pair in self.table[index]:  
            if pair[0] == key:  
                pair[1] = value  
                return  
        self.table[index].append([key, value])  
  
    def search(self, key):  
        index = self.hash_function(key)  
        for pair in self.table[index]:  
            if pair[0] == key:  
                return pair[1]  
        return "Not Found"  
  
    def delete(self, key):  
        index = self.hash_function(key)  
        for pair in self.table[index]:  
            if pair[0] == key:  
                self.table[index].remove(pair)  
                return "Deleted"  
        return "Not Found"  
  
# Example Usage  
ht = HashTable(10)  
ht.insert("a", 1)  
ht.insert("b", 2)  
print(ht.search("a"))  
ht.delete("a")  
print(ht.search("a"))
```

### Code Explanation:

Hash table uses chaining for collision handling. Keys are hashed into index positions. Insert adds key-value pairs, search retrieves value, and delete removes pair.

### **Sample Output:**

```
1
Not Found
```