Assignment -11.2

Hallticket :2303A51128

Batch:03

Task Description -1 – (Stack Using AI Guidance)

• Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output:

• A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.

Prompt:

Create a Python Stack class with push, pop, peek, and is_empty methods. Include proper documentation and example usage

Code:

```
AIAC11_2.py > ...
class Stack:
    """Stack (LIFO) - Last In First Out"""
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop() if not self.is_empty() else None
    def peek(self):
        return self.items[-1] if not self.is_empty() else None
    def is_empty(self):
        return len(self.items) == 0

# Test
if __name__ == "__main__":
    s = Stack()
    s.push(10)
    s.push(20)
    s.push(30)
    print(f"Peek: {s.peek()}")
    print(f"Pop: {s.pop()}")
    print(f"Pop: {s.pop()}")
    print(f"Pop: {s.pop()}")
    print(f"Is empty: {s.is_empty()}")

```

Output:

```
Peek: 30
Pop: 30
Pop: 20
Pop: 10
Is empty: True
```

Justification:

LIFO (Last In First Out) Python list is used because append() and pop() make push and pop easy and fast.
Methods like peek() and is_empty() are added to avoid errors and improve usability.

Task Description -2 – (Queue Design)

• Task:Use AI assistance to create a Queue data structure following FIFO principles

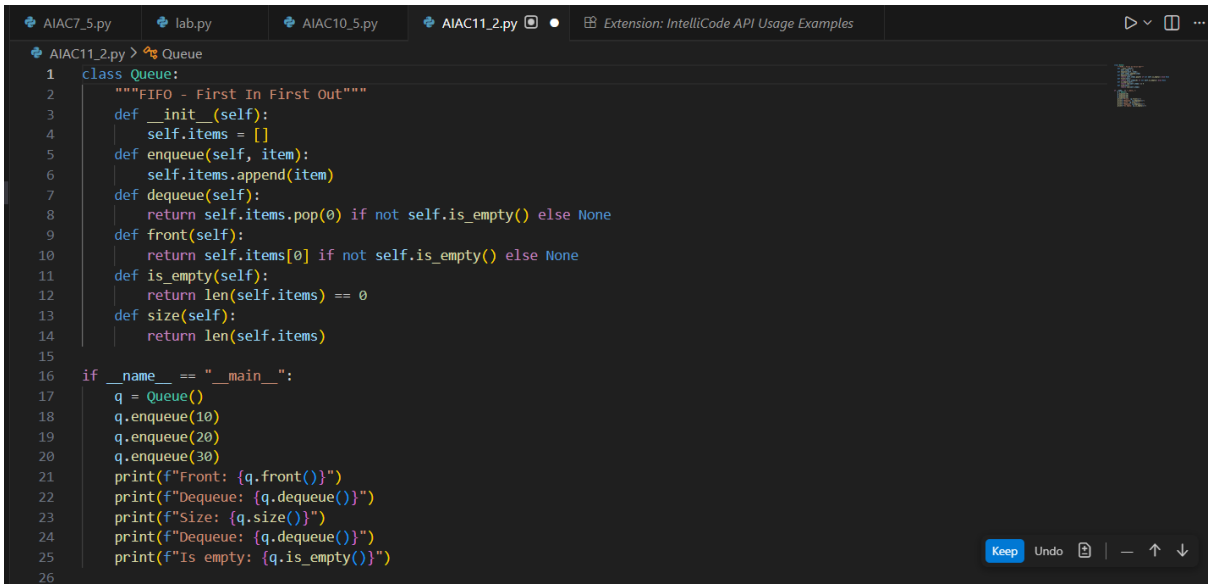
Expected Output:

• A complete Queue implementation including enqueue, dequeue, front element access, and size calculation

Prompt:

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

Code:



Output:

Justification:

Queue follows FIFO (First In First Out).
List is used to insert elements at rear and remove from front.
Functions like enqueue(), dequeue(), and front() maintain proper queue order

Task Description -3 –(Singly Linked List Construction)

• Task: Utilize AI to build a singly linked list supporting insertion and traversal.

Expected Output:

• Correctly functioning linked list with node creation, insertion logic, and display functionality.

Prompt:

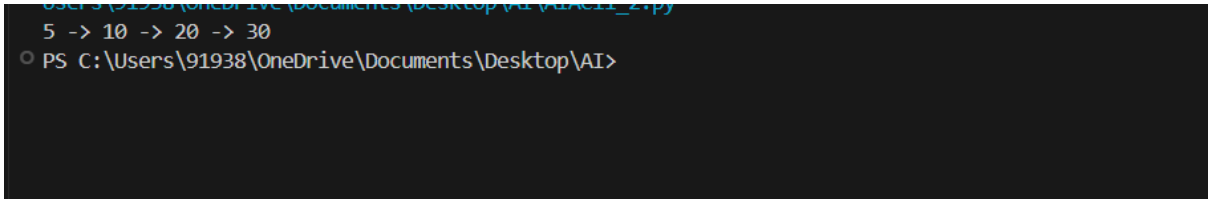
Build a singly linked list in Python with node creation, insertion at end, and traversal display

Code:

AIAC11_2.py > LinkedList > insert_beg

```python
# Node class - stores data and reference to next node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
# LinkedList class - manages nodes
class LinkedList:
    def __init__(self):
        self.head = None
    # Insert at end
    def insert_end(self, data):
        n = Node(data)
        if not self.head:
            self.head = n
            return
        c = self.head
        while c.next:
            c = c.next
        c.next = n
    # Insert at beginning
    def insert_beg(self, data):
        n = Node(data)
        n.next = self.head
        self.head = n
    # Traverse and display all nodes
    def show(self):
        c = self.head
        while c:
            print(c.data, end=" -> " if c.next else "\n")
            c = c.next

# Test: Create list, insert values, display
ll = LinkedList()
ll.insert_end(10)
ll.insert_end(20)
ll.insert_end(30)
ll.insert_beg(5)
ll.show()
```

Output:



Justification:

Linked list uses nodes connected by pointers. Separate Node class stores data and next reference. Insertion and traversal show how elements are linked dynamically.

Task Description -4 – (Binary Search Tree Operations)

• Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.

Expected Output:

• BST program with correct node insertion and in-order traversal output.

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

Code:

```python
#Implement a Binary Search Tree in Python with insert and in-order traversal methods.Provide comments and example execution.
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursively(self.root, key)
    def _insert_recursively(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert_recursively(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert_recursively(node.right, key)
    def in_order_traversal(self):
        return self._in_order_recursively(self.root)
    def _in_order_recursively(self, node):
        res = []
        if node:
            res = self._in_order_recursively(node.left)
            res.append(node.val)
            res = res + self._in_order_recursively(node.right)
        return res
# Example execution
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(50)
    bst.insert(30)
    bst.insert(20)
    bst.insert(40)
    bst.insert(70)
    bst.insert(60)
    bst.insert(80)

    print("In-order traversal of the binary search tree:")
    print(bst.in_order_traversal())
    # Output should be: [20, 30, 40, 50, 60, 70, 80]
```

Output:

```
PS C:\Users\shyam_3jvctb4\OneDrive\Documents\AI Assistant Coding> & C:/Users/shyam_3jvctb4/AppData/Local
Documents/AI Assistant Coding/ass11_2.py"
In-order traversal of the binary search tree:
[20, 30, 40, 50, 60, 70, 80]
```

Justification:

This program implements a Binary Search Tree (BST) using a Node class and a BinarySearchTree class. The insert() method adds elements following the BST rule (left < root < right). The in_order_traversal() method prints elements in sorted order. The example shows values printed in ascending order, proving the BST works correctly.

Task Description -5 – (Hash Table Implementation)

• Task: Create a hash table using AI with collision handling

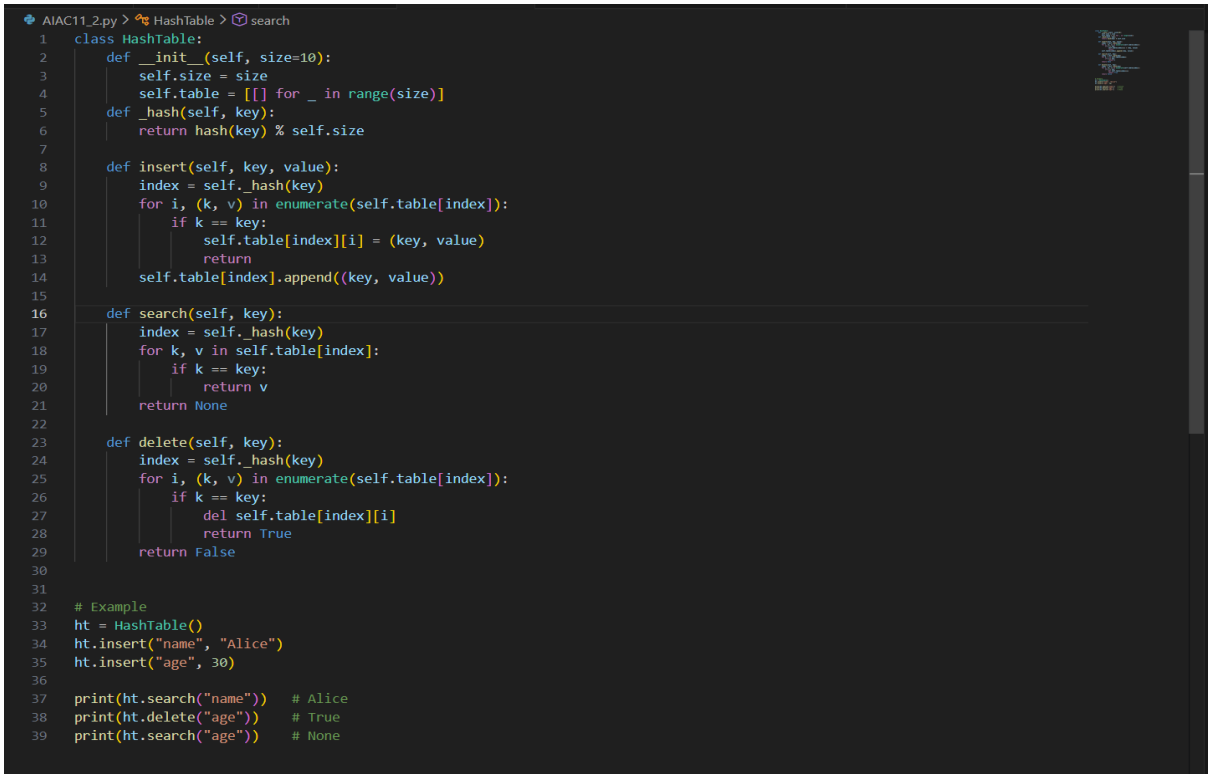
Expected Output:

• Hash table supporting insert, search, and delete using chaining or open

Prompt:

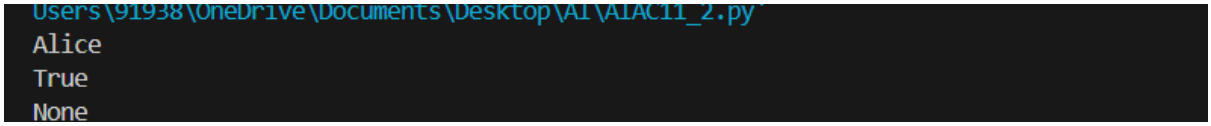
create a Hash Table in Python with insert, search, delete operations using chaining collision handling

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):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        self.table[index].append((key, value))

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

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


# Example
ht = HashTable()
ht.insert("name", "Alice")
ht.insert("age", 30)

print(ht.search("name"))   # Alice
print(ht.delete("age"))    # True
print(ht.search("age"))    # None
```

Output:



```
Users\91938\OneDrive\Documents\Desktop\AI\AIAC11_2.py
Alice
True
None
```

Justification:

Hash table allows fast data access. Modulo hash function is used to find index. Chaining method handles collisions by storing multiple elements in one bucket.

Task Description -6 :

prompt : "With the help of AI, implement the Inorder Traversal of a Binary Tree... give short and simpler code"

Code:

cp.py

cp.py > main

```python
"""
Binary Tree Inorder Traversal (Left → Root → Right)
Simple, short implementation with sample tree and printed output.
"""

class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def inorder(root, res=None):
    """Return list of values in inorder (Left, Root, Right)."""
    if res is None:
        res = []
    if root:
        inorder(root.left, res)
        res.append(root.val)
        inorder(root.right, res)
    return res

def main():
    # Build sample tree:
    #     1
```

```python
cp.py > ...
    def inorder(root, res=None):
        return res

    def main():
        # Build sample tree:
        #     1
        #    / \
        #   2   3
        #  / \
        # 4   5
        n4 = Node(4)
        n5 = Node(5)
        n2 = Node(2, n4, n5)
        n3 = Node(3)
        root = Node(1, n2, n3)

        print("Inorder Traversal of the sample tree (Left → Root → Right):")
        result = inorder(root)
        print(result)

    if __name__ == '__main__':
        main()
```

Output:

```
Inorder Traversal of the sample tree (Left → Root → Right):
[4, 2, 5, 1, 3]
```

Justification:

Node: simple binary node with value, left and right children

Inorder(root) : recursive traversal visiting left, then root, then right; returns a list of values

Sample tree built in main:

root 1 with left child 2 and right child 3; node 2 has children 4 and 5

main: builds sample tree, calls inorder, prints result and short notes

Task Description -7 :

Prompt:implement preorder traversal, show output for a sample tree.

Code:

cp.py > main

```python
"""
Binary Tree Preorder Traversal (Root → Left → Right)
Simple, short implementation with sample tree and printed output.
"""

class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def preorder(root, res=None):
    """Return list of values in preorder (Root, Left, Right)."""
    if res is None:
        res = []
    if root:
        res.append(root.val)
        preorder(root.left, res)
        preorder(root.right, res)
    return res


def main():
    # Build sample tree:
    #     1
    #    / \
    #   2   3
    #  / \
    # 4   5
    n4 = Node(4)
    n5 = Node(5)
    n2 = Node(2, n4, n5)
    n3 = Node(3)
    root = Node(1, n2, n3)

    print("Preorder Traversal of the sample tree (Root → Left → Right):")
    result = preorder(root)
    print(result)

if __name__ == '__main__':
    main()
```

Output:

```
di/OneDrive/Documents/aicoding_3-2/lab/cp.py
Preorder Traversal of the sample tree (Root → Left → Right):
[1, 2, 4, 5, 3]
```

Justification:

Comments & Code Explanation

Node: simple binary tree node with fields val, left, right.

preorder(root): recursive function — append root.val, then recurse preorder(root.left), then preorder(root.right); returns a list of visited values.

main(): builds sample tree (1 root, left 2, right 3, and 2 has children 4 and 5), calls preorder(root), prints the result and brief notes.

Traversal order: Root → Left → Right.

Sample output: [1, 2, 4, 5, 3]