

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 > ...
1 class Stack:
2     """Stack (LIFO) - Last In First Out"""
3     def __init__(self):
4         self.items = []
5     def push(self, item):
6         self.items.append(item)
7     def pop(self):
8         return self.items.pop() if not self.is_empty() else None
9     def peek(self):
10        return self.items[-1] if not self.is_empty() else None
11    def is_empty(self):
12        return len(self.items) == 0
13
14    # Test
15    if __name__ == "__main__":
16        s = Stack()
17        s.push(10)
18        s.push(20)
19        s.push(30)
20        print(f"Peek: {s.peek()}")
21        print(f"Pop: {s.pop()}")
22        print(f"Pop: {s.pop()}")
23        print(f"Pop: {s.pop()}")
24        print(f"Is empty: {s.is_empty()}")
25
```

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

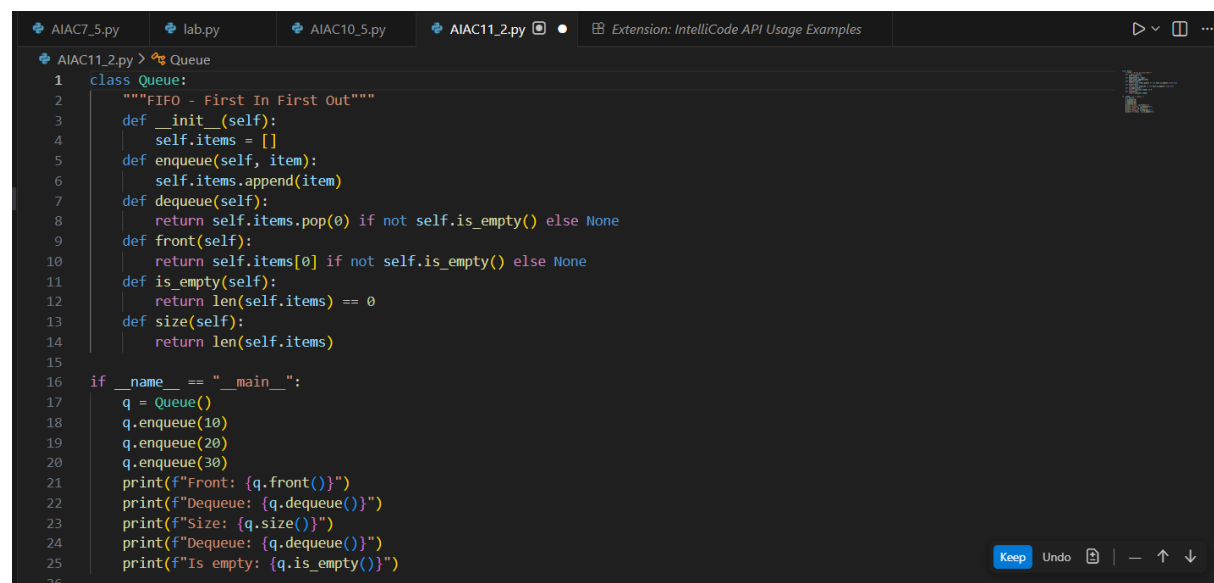
Expected Output:

- 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:

A screenshot of a code editor with a dark theme. The editor shows a Python file named 'AIAC11_2.py' with a class 'Queue' implementation. The code includes methods for enqueue, dequeue, front, is_empty, and size. A main block at the bottom demonstrates the usage of the Queue class by enqueuing 10, 20, and 30, then printing the front, dequeuing, printing the size, dequeuing again, and checking if it's empty. The code is as follows:

```
1 class Queue:
2     """FIFO - First In First Out"""
3     def __init__(self):
4         self.items = []
5     def enqueue(self, item):
6         self.items.append(item)
7     def dequeue(self):
8         return self.items.pop(0) if not self.is_empty() else None
9     def front(self):
10        return self.items[0] if not self.is_empty() else None
11    def is_empty(self):
12        return len(self.items) == 0
13    def size(self):
14        return len(self.items)
15
16 if __name__ == "__main__":
17     q = Queue()
18     q.enqueue(10)
19     q.enqueue(20)
20     q.enqueue(30)
21     print(f"Front: {q.front()}")
22     print(f"Dequeue: {q.dequeue()}")
23     print(f"Size: {q.size()}")
24     print(f"Dequeue: {q.dequeue()}")
25     print(f"Is empty: {q.is_empty()}")
26
```

Output:

A screenshot of the program's output in a terminal window. The output shows the results of the operations performed in the main block of the code: Front: 10, Dequeue: 10, Size: 2, Dequeue: 20, and Is empty: False.

```
Front: 10
Dequeue: 10
Size: 2
Dequeue: 20
Is empty: False
```

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
1  # Node class - stores data and reference to next node
2  class Node:
3      def __init__(self, data):
4          self.data = data
5          self.next = None
6  # LinkedList class - manages nodes
7  class LinkedList:
8      def __init__(self):
9          self.head = None
10     # Insert at end
11     def insert_end(self, data):
12         n = Node(data)
13         if not self.head:
14             self.head = n
15             return
16         c = self.head
17         while c.next:
18             c = c.next
19         c.next = n
20     # Insert at beginning
21     def insert_beg(self, data):
22         n = Node(data)
23         n.next = self.head
24         self.head = n
25     # Traverse and display all nodes
26     def show(self):
27         c = self.head
28         while c:
29             print(c.data, end=" -> " if c.next else "\n")
30             c = c.next
31
32     # Test: Create list, insert values, display
33     ll = LinkedList()
34     ll.insert_end(10)
35     ll.insert_end(20)
36     ll.insert_end(30)
37     ll.insert_beg(5)
38     ll.show()
39
40
```

Output:

```
Users\91938\OneDrive\Documents\Desktop\AI\AI11_2.py
5 -> 10 -> 20 -> 30
PS C:\Users\91938\OneDrive\Documents\Desktop\AI>
```

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:

```

1 #Implement a Binary Search Tree in Python with insert and in-order traversal methods.Provide comments and example execution.
2 class Node:
3     def __init__(self, key):
4         self.left = None
5         self.right = None
6         self.val = key
7 class BinarySearchTree:
8     def __init__(self):
9         self.root = None
10
11     def insert(self, key):
12         if self.root is None:
13             self.root = Node(key)
14         else:
15             self._insert_recursively(self.root, key)
16     def _insert_recursively(self, node, key):
17         if key < node.val:
18             if node.left is None:
19                 node.left = Node(key)
20             else:
21                 self._insert_recursively(node.left, key)
22         else:
23             if node.right is None:
24                 node.right = Node(key)
25             else:
26                 self._insert_recursively(node.right, key)
27     def in_order_traversal(self):
28         return self._in_order_recursively(self.root)
29     def _in_order_recursively(self, node):
30         res = []
31         if node:
32             res = self._in_order_recursively(node.left)
33             res.append(node.val)
34             res = res + self._in_order_recursively(node.right)
35         return res
36
37 # Example execution
38 if __name__ == "__main__":
39     bst = BinarySearchTree()
40     bst.insert(50)
41     bst.insert(30)
42     bst.insert(20)
43     bst.insert(40)
44     bst.insert(70)
45     bst.insert(60)
46     bst.insert(80)
47
48     print("In-order traversal of the binary search tree:")
49     print(bst.in_order_traversal())
50     # 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:

```
AIAC11_2.py > HashTable > search
1 class HashTable:
2     def __init__(self, size=10):
3         self.size = size
4         self.table = [[] for _ in range(size)]
5     def _hash(self, key):
6         return hash(key) % self.size
7
8     def insert(self, key, value):
9         index = self._hash(key)
10        for i, (k, v) in enumerate(self.table[index]):
11            if k == key:
12                self.table[index][i] = (key, value)
13                return
14        self.table[index].append((key, value))
15
16    def search(self, key):
17        index = self._hash(key)
18        for k, v in self.table[index]:
19            if k == key:
20                return v
21        return None
22
23    def delete(self, key):
24        index = self._hash(key)
25        for i, (k, v) in enumerate(self.table[index]):
26            if k == key:
27                del self.table[index][i]
28                return True
29        return False
30
31    # Example
32    ht = HashTable()
33    ht.insert("name", "Alice")
34    ht.insert("age", 30)
35
36    print(ht.search("name")) # Alice
37    print(ht.delete("age")) # True
38    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

```
1  """
2  Binary Tree Inorder Traversal (Left → Root → Right)
3  Simple, short implementation with sample tree and printed output.
4  """
5
6  class Node:
7      def __init__(self, val, left=None, right=None):
8          self.val = val
9          self.left = left
10         self.right = right
11
12     def inorder(root, res=None):
13         """Return list of values in inorder (Left, Root, Right)."""
14         if res is None:
15             res = []
16         if root:
17             inorder(root.left, res)
18             res.append(root.val)
19             inorder(root.right, res)
20         return res
21
22     def main():
23         # Build sample tree:
24         #     1
```

```

cp.py > ...
12 def inorder(root, res=None):
13     if root:
14         res.append(root.val)
15         inorder(root.left, res)
16         inorder(root.right, res)
17     return res
18
19 def main():
20     # Build sample tree:
21     #      1
22     #    / \
23     #   2  3
24     #  / \
25     # 4  5
26     n4 = Node(4)
27     n5 = Node(5)
28     n2 = Node(2, n4, n5)
29     n3 = Node(3)
30     root = Node(1, n2, n3)
31
32     print("Inorder Traversal of the sample tree (Left → Root → Right):")
33     result = inorder(root)
34     print(result)
35
36 if __name__ == '__main__':
37     main()
38
39
40
41

```

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
1  """
2  Binary Tree Preorder Traversal (Root → Left → Right)
3  Simple, short implementation with sample tree and printed output.
4  """
5
6  class Node:
7      def __init__(self, val, left=None, right=None):
8          self.val = val
9          self.left = left
10         self.right = right
11
12
13     def preorder(root, res=None):
14         """Return list of values in preorder (Root, Left, Right)."""
15         if res is None:
16             res = []
17         if root:
18             res.append(root.val)
19             preorder(root.left, res)
20             preorder(root.right, res)
21         return res
22
```

```
22
23
24     def main():
25         # Build sample tree:
26         #      1
27         #     / \
28         #    2   3
29         #   / \
30         #  4   5
31         n4 = Node(4)
32         n5 = Node(5)
33         n2 = Node(2, n4, n5)
34         n3 = Node(3)
35         root = Node(1, n2, n3)
36
37         print("Preorder Traversal of the sample tree (Root → Left → Right):")
38         result = preorder(root)
39         print(result)
40
41     if __name__ == '__main__':
42         main()
43
```

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]