AI ASSIST CODING

ASSIGNMENT-11.2

VANCHA SRAVYA

2303A51124

BATCH NO:15

Data Structures with AI: Implementing Fundamental Structures

Lab Objectives

- **Use AI to assist in designing and implementing fundamental data structures in Python.**
- **Learn how to prompt AI for structure creation, optimization, and documentation.**
- **Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables.**
- **Enhance code quality with AI-generated comments and performance suggestions.**

TASK-1: Stack Using AI Guidance

PROMPT:

Design a Stack class in Python supporting push, pop, peek, and is_empty operations.

Add proper documentation and comments.

PYTHON CODE:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
```

```
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return "Stack is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return "Stack is empty"

    def is_empty(self):
        return len(self.items) == 0
s = Stack()
s.push(10)
s.push(20)
print("Top:", s.peek())
print("Popped:", s.pop())
print("Is Empty:", s.is_empty())
```

OUTPUT:

Top: 20

Popped: 20

Is Empty: False

TASK-2: Queue Design

PROMPT:

Create a Queue class in Python following FIFO principle.

Include enqueue, dequeue, front, and size methods with documentation.

PYTHON CODE:

```
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)
        return

    def front(self):
        if not self.is_empty():
            return self.items[0]
        return

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

    def is_empty(self):
        return len(self.items) == 0
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print("Front:", q.front())
print("Dequeued:", q.dequeue())
print("Size:", q.size())
print("Is Empty:", q.is_empty())
```

OUTPUT:

Front: 5

Dequeued: 5

Size: 1

TASK-3:Singly Linked List Construction

PROMPT:

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

Add proper comments.

PYTHON CODE:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None


    def insert(self, data):
        new_node = Node(data)
        # If list is empty, make new node as head
        if self.head is None:
            self.head = new_node
            return
        # Otherwise, traverse to the last node
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
```

```
    def display(self):
        if self.head is None:
            print("Linked List is empty")
            return
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next

        print("None")

if __name__ == "__main__":
    linked_list = SinglyLinkedList()
    numbers = list(map(int, input("Enter numbers separated by space: ").split()))
    for num in numbers:
        linked_list.insert(num)
    print("\nLinked List Elements:")
    linked_list.display()
```

INPUT:

Enter numbers separated by space: 10 20 30 40

OUTPUT:

Linked List Elements:

10 -> 20 -> 30 -> 40 -> None

TASK-4: Binary Search Tree Operations

PROMPT:

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

Add documentation and comments.

PYTHON CODE:

```
class BSTNode:
```

```
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None
    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if node is None:
            return BSTNode(key)

        if key < node.key:
            node.left = self._insert(node.left, key)
        else:
            node.right = self._insert(node.right, key)

        return node

    def inorder(self):
        print("In-order Traversal:", end=" ")
        self._inorder(self.root)
        print()

    def _inorder(self, node):
        if node:
            self._inorder(node.left)
```

```
            print(node.key, end=" ")
            self._inorder(node.right)
if __name__ == "__main__":

    bst = BinarySearchTree()

    numbers = list(map(int, input("Enter numbers separated by space: ").split()))

    for num in numbers:
        bst.insert(num)

    print("\nBinary Search Tree Created Successfully.")
    bst.inorder()
```

OUTPUT:

Binary Search Tree Created Successfully.

In-order Traversal: 10 20 30

TASK-5: Hash Table Implementation

PROMPT:

Create a Hash Table in Python with collision handling using chaining.

Support insert, search, and delete operations.

PYTHON CODE:

```
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 "Key not found"
    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 "Deleted"
        return "Key not found"
    def display(self):
        for i, bucket in enumerate(self.table):
            print(f"Index {i}: {bucket}")
if __name__ == "__main__":
    ht = HashTable()
    ht.insert("name", "Kiranmai")
    ht.insert("age", 19)
    ht.insert("course", "AI")
    print("\nHash Table:")
    ht.display()
print("\nSearch 'name':", ht.search("name"))
```

```python
    print("Delete 'name':", ht.delete("name"))
    print("Search 'name':", ht.search("name"))
    print("\nUpdated Hash Table:")
    ht.display()
```

OUTPUT:

```
Hash Table:
Index 0: []
Index 1: [('age', 19)]
Index 2: []
Index 3: [('course', 'AI')]
Index 4: []
Index 5: []
Index 6: []
Index 7: []
Index 8: [('name', 'Kiranmai')]
Index 9: []

Search 'name': Kiranmai
Delete 'name': Deleted
Search 'name': Key not found

Updated Hash Table:
Index 0: []
Index 1: [('age', 19)]
Index 2: []
Index 3: [('course', 'AI')]
Index 4: []
Index 5: []
Index 6: []
```

Index 7: []

Index 8: []

Index 9: []