

Lab 11.5 – Data Structures with AI

Student Name: seetharam reddy

Roll No: 2303A52440

Program Name: B. Tech

Course Code: 23CS002PC304

Course Title: AI Assisted Coding

Year/Sem: III/II

Regulation: R23

Date and Day: Week 6 – Friday

Task 1 – Stack Implementation

```
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 None  
  
    def peek(self):  
        if not self.is_empty():  
            return self.items[-1]  
        return None  
  
    def is_empty(self):  
        return len(self.items) == 0  
  
s = Stack()  
s.push(10)  
s.push(20)  
print(s.peek())  
print(s.pop())
```

Screenshot – Stack Output:

The screenshot shows a code editor window with a file named `main.py`. The code defines a `Stack` class with methods for pushing, popping, peeking, and checking if it's empty. It includes a demo section where a stack is created, two items are pushed, and then popped and peeked. The output pane shows the results of these operations.

```
main.py + 44e5mfgzp Ø
Output:
Top Element: 20
Popped: 20
Is Empty: False

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         return None
12
13     def peek(self):
14         if not self.is_empty():
15             return self.items[-1]
16         return None
17
18     def is_empty(self):
19         return len(self.items) == 0
20
21
22 # Demo
23 s = Stack()
24 s.push(10)
25 s.push(20)
26
27 print("Top Element:", s.peek())
28 print("Popped:", s.pop())
29 print("Is Empty:", s.is_empty())
30
31
```

Task 2 – Queue Implementation

```
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 None

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        return None
```

```

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

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

q = Queue()
q.enqueue(1)
q.enqueue(2)
print(q.dequeue())

```

Screenshot – Queue Output:

main.py + 44e5mfgzp Ø

```

1+ class Queue:
2+     def __init__(self):
3+         self.items = []
4+
5+     def enqueue(self, item):
6+         self.items.append(item)
7+
8+     def dequeue(self):
9+         if not self.is_empty():
10+             return self.items.pop(0)
11+         return None
12+
13+     def peek(self):
14+         if not self.is_empty():
15+             return self.items[0]
16+         return None
17+
18+     def size(self):
19+         return len(self.items)
20+
21+     def is_empty(self):
22+         return len(self.items) == 0
23+
24+
25+ # Demo
26+ q = Queue()
27+ q.enqueue(1)
28+ q.enqueue(2)
29+
30+ print("Dequeued:", q.dequeue())
31+ print("Front:", q.peek())
32+ print("Size:", q.size())
33+
34+
35+

```

Output:
Dequeued: 1
Front: 2
Size: 1

Task 3 – Linked List

```

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")

ll = LinkedList()
ll.insert(5)
ll.insert(10)
ll.insert(15)
ll.display()
```

Screenshot – Linked List Output:

The screenshot shows a code editor window with a file named `main.py`. The code defines a `Node` class and a `LinkedList` class. The `LinkedList` class has methods for inserting nodes and displaying the list. A demo section at the bottom shows the creation of a `LinkedList` object, insertion of three nodes (5, 10, 15), and then calling the `display` method which prints the nodes separated by arrows and ends with `None`.

```
main.py + 44e5mfgzp Ø
1 ~ class Node:
2 ~     def __init__(self, data):
3 ~         self.data = data
4 ~         self.next = None
5 ~
6 ~ class LinkedList:
7 ~     def __init__(self):
8 ~         self.head = None
9 ~
10 ~     def insert(self, data):
11 ~         new_node = Node(data)
12 ~         if not self.head:
13 ~             self.head = new_node
14 ~             return
15 ~
16 ~         temp = self.head
17 ~         while temp.next:
18 ~             temp = temp.next
19 ~         temp.next = new_node
20 ~
21 ~     def display(self):
22 ~         temp = self.head
23 ~         while temp:
24 ~             print(temp.data, end=" -> ")
25 ~             temp = temp.next
26 ~         print("None")
27 ~
28 ~
29 ~ # Demo
30 ~ ll = LinkedList()
31 ~ ll.insert(5)
32 ~ ll.insert(10)
33 ~ ll.insert(15)
34 ~
35 ~ ll.display()
```

Task 4 – Hash Table

```
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        return key % self.size

    def insert(self, key):
        index = self.hash_function(key)
        if key not in self.table[index]:
            self.table[index].append(key)

    def search(self, key):
        index = self.hash_function(key)
        return key in self.table[index]
```

```

def delete(self, key):
    index = self.hash_function(key)
    if key in self.table[index]:
        self.table[index].remove(key)

h = HashTable()
h.insert(10)
h.insert(20)
print(h.search(10))

```

Screenshot – Hash Table Output:

```

main.py + 44e5mfgzp ⚡
1 class HashTable:
2     def __init__(self, size=10):
3         self.size = size
4         self.table = [[] for _ in range(size)]
5
6     def hash_function(self, key):
7         return key % self.size
8
9     def insert(self, key):
10        index = self.hash_function(key)
11        if key not in self.table[index]:
12            self.table[index].append(key)
13
14     def search(self, key):
15        index = self.hash_function(key)
16        return key in self.table[index]
17
18     def delete(self, key):
19        index = self.hash_function(key)
20        if key in self.table[index]:
21            self.table[index].remove(key)
22
23
24 # Demo
25 h = HashTable()
26 h.insert(10)
27 h.insert(20)
28
29 print("Search 10:", h.search(10))
30 h.delete(10)
31 print("Search 10 after delete:", h.search(10))
32
33

```

Output:

Search 10: True
Search 10 after delete: False

Task 5 – Graph

```

class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, v):
        if v not in self.graph:
            self.graph[v] = []

    def add_edge(self, u, v):
        self.add_vertex(u)
        self.add_vertex(v)

```

```

        self.graph[u].append(v)
        self.graph[v].append(u)

    def display(self):
        for vertex in self.graph:
            print(vertex, "->", self.graph[vertex])

g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.display()

```

Screenshot – Graph Output :

The screenshot shows a code editor interface with a file named `main.py`. The code defines a `Graph` class with methods for adding vertices and edges, and displaying the graph structure. A demo section at the bottom shows the creation of a graph with 4 vertices and 3 edges, resulting in the following output:

```

main.py      +
44e5mfgzp Ø

1  class Graph:
2      def __init__(self):
3          self.graph = {}
4
5      def add_vertex(self, v):
6          if v not in self.graph:
7              self.graph[v] = []
8
9      def add_edge(self, u, v):
10         self.add_vertex(u)
11         self.add_vertex(v)
12         self.graph[u].append(v)
13         self.graph[v].append(u)
14
15     def display(self):
16         for vertex in self.graph:
17             print(vertex, "->", self.graph[vertex])
18
19
20 # Demo
21 g = Graph()
22 g.add_edge(1, 2)
23 g.add_edge(1, 3)
24 g.add_edge(2, 4)
25
26 g.display()
27

```

Output:

```

1 -> [2, 3]
2 -> [1, 4]
3 -> [1]
4 -> [2]

```

Result

All fundamental data structures were successfully implemented.