- Name : G. Likhitha Rao
- Ht No: 2303A52487
- Batch - 35

TASK 1

```
class Stack:

    Stack Data Structure implementation using Python list.
    Follows LIFO (Last In First Out) principle.


    def finite(self):

        Initializes an empty stack.

        self.items = []

    def push(self, item):

        Adds an element to the top of the stack.

        Parameters:
        item (any): Element to be pushed onto the stack

        self.items.append(item)

    def pop(self):

        Removes and returns the top element from the stack.

        Returns:
        any: Top element of the stack
        If stack is empty, returns None

        If- not self.1s_empty( ):
            return self.items.pop( )
        return None

    def peek(self):

        Returns the top element of the stack without removing it.

        Returns:
        any: Top element of the stack
        If stack is empty, returns None

        if not self.is_empty():
            return self.items[-l]
        return None

    def 1s_empty(self):

        Checks whether the stack is empty.

        Returns:
        bool: True if stack is empty, False otherwise

        return len(self.items) == 0


# Sample usage
s = Stack()
s.push(10)
s.push(20)
print("Top e1ement: ", s.peek())
print("Popped element: ", s.pop())
print("Is stack enpty?", s.1s_empty())
```

```
Top element: 20
Popped e1ement: 20
Is stack enpty? False
```

TASK 2

```
class Queue:

    Queue Data Structure implementation using Python list.
    Follows FIFO (First In First Out) principle.


    def finite(self):

        Initializes an empty queue.

        self.items = []

    def enqueue(self, item):

        Adds au element to the rear of the queue.

        Parameters:
        item (any): Element to be added to the queue

        self.items.append(item)

    def dequeue(self):

        Removes and netunns the front element of the queue.

        Returns:
        any: Front element of the queue
        If queue is empty, returns None

        if not self.is_empty():
            return self.items.pop(0)
        return None

    def peek(self):

        Returns the front element without removing it.

        Returns :
        any: Front element of the queue
        If queue is empty, returns None

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

    def size(self):

        Retunns the numben of elements in the queue.

        Returns:
        int: Size of the queue

        netunn len(self.items)

    def is_empty(self):

        Checks whether the queue is empty.

        Returns:
        bool: True if queue is empty, False otherwise

        return len(self.items) == 0


# Sample usage
q = Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

print("Front element:“, q.peek())
print("Dequeued element:“, q.dequeue())
```

```
print("queue size:", q.size())
```

```
Front element: 10
Dequeued element: 10
Queue size: 2
```

TASK 3

```
class Node:

    Node class represents a single element in a singly linked list.


    def finite(self, data):

        Initializes a node with data and a reference to the next node.

        Parameters:
        data (any): Value stored in the node

        self.data = data
        self.uext = None


class LinkedList:

    Singly Linked List implementation.


    def finite(self):

        Initializes an empty linked list.

        self.head = None

    def insert(self, data):

        Inserts a new node at the beginning of the linked list.

        Parameters:
        data (any): Value to be inserted

        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):

        Displays all elements of the linked list.

        temp = self.head
        while temp:
            print(temp.data, end=" -› ")
            temp = tenp.next
        print("None")


# Sample usage
11 = LinkedList()
11.insert(10)
11.insert(20)
11.insert(30)

11.display()
```

```
30 -> 20 -> 10 -> None
```

TASK 4

```
class Node:

    Node class represents a single node in a Binary Search Tree.
```

```
    def finite(self, key):

        Initializes a node with a key value.

        Parameters:
        key (int): Value stored in the node

        self.key = key
        self.left = None
        self.night = None


class BST:

    Binary Search Tree implementation with recursive insert
    and in-onden tnavensal methods.


    def finite(self):

        Initializes an empty BST.

        self.root = None

    def insert(self, key):

        Inserts a key into the BST.

        Parameters:
        key (int): Value to be inserted

        self.root = self._insert_recursive(self.root, key)

    def _insent_recursive(self, node, key):

        Recursively inserts a key into the BST.

        Parameters:
        node (Node): Current node
        key (int): Value to be inserted

        Returns :
        Node : Updated node

        lf- node 1s None:
            return Node(key)

        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        elif key > node.key:
            node.right = self._insert_recursive(node.right, key)

        return node

    def inorder(self):

        Performs in-order traversal of the BST.

        self._inorder_recursive(self.root)

    def _inorder_recursive(self, node):

        Recursive helper for in-order traversal.

        Parameters:
        node (Node): Current node

        if node:
            self._inorder_recursive(node.left)
            pnint(node.key, end=" ")
            self._inorder_recursive(node.right)


# Sample usage
bst = BST()
bst.insert(50)
```

```
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)

print("In-order Traversal:")
bst.inorder()
```

```
In-order Traversal:
20 30 40 50 60 70 80
```

TASK 5

```
class HashTable:

    Hash Table implementation using chaining for collision handling.


    def  finite(self,  size=10):

        Initializes the hash table with a fixed size.

        Parameters:
        size (iut): Number of buckets in the hash table

        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):

        Generates a hash index for a given key.

        Parameters:
        key (str): Key to be hashed

        Returns:
        Int: Hash index

        return hash(key) % self.size

    def insert(self, key, value):

        inserts a key-value pair into the hash table.

        Parameters:
        key (str): Key to insert
        value (any): Value associated with the key

        index = self._hash(key)

        # Check if key already exists and update
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return

        # Otherwise, add new key-value pair
        self.table[index].append([key, value])

    def search(self, key):

        Searches for a value by key in the hash table.

        Parameters:
        key (str): Key to search

        Returns :
        any: Value if found, otherwise None

        index = self._hash(key)

        for pair in self.table[iudex]:
            if pair[0] == key:
```

```
                return pair[1]

        return  None

    def delete(self, key):

        Deletes a key-value pair from the hash table.

        Parameters:
        key (str): Key to delete

        Returns:
        bool: True if deleted, False if key not found

        index = self._hash(key)

        fon i, pair in enumenate(self.table[index]):
            if pair[0] == key:
                del self.table[iudex][i]
                return True

        return False


# Sample usage
ht = HashTable()

ht.insert(“name“, "Harini")
ht.insert(“age“, 20)
ht.insert(“course“, "AI")

print("Search name:“, ht.search(“name"))
ht.delete(“age“)
print("Search age:“, ht.search(”age“))
```

```
Search name: Harini
Search age: None
```

TASK 6

```
class Graph:

    Graph implementation using an adjacency list.


    def finite(self):

        Initializes an empty graph.

        self.graph = {}

    def add_vertex(self, vertex):

        Adds a vertex to the graph.

        Parameters:
        vertex (any): Vertex to be added

        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertexl, vertex2):

        Adds an edge between two vertices (undirected graph).

        Parameters:
        vertexl (any): Finst vertex
        vertex2 (any): Second vertex

        # Ensure both vertices exist
        if vertex1 not in self.graph:
            self.add_vertex(vertexl)
        if vertex2 not in self.graph:
            self.add_vertex(vertex2)
```

```
            # Add connections
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertexl)

        def display(self):

            Displays the graph connections.

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


    # Sample usage
    g - Graph( )
    g.add_vertex(“A“)
    g.add_vertex(“B“)
    g.add_vertex(“C“)

    g. add_edge("A", "B")
    g.add_edge("A", "C")
    g.add_edge("B", "C")

    g.display()
```

```
A -> ['B', 'C']
B -> ['A', 'C']
C -> ['A', 'B']
```

TASK 7

```
import heapq

class PriorityQueue:

    Priority Queue implementation using heapq (min-heap).
    Lower priority value means higher priority.


    def finite(self):

        Initializes an empty priority queue.

        self.queue = []

    def euqueue(self, item, priority):

        Inserts an element into the priority queue with a given priority.

        Parameters:
        item (any): Data to be stored
        priority (int): Priority of the item (lower value = higher priority)

        heapq.heappush(self.queue, (priority, item))

    def dequeue(self):

        Removes and returns the element with the highest priority.

        Returns:
        any: Item with highest priority
        If queue is empty, returns None

        if self.is_empty():
            return  None
        return heapq.heappop(self.queue)[1]

    def display(self):

        Displays the elements in the priority queue.

        print(“Priority Queue:”, self.queue)

    def is_empty(self):

        Checks whether the priority queue is empty.
```

```
        Returns:
        bool: True if empty, False othenwise

        return len(self.queue) == 0


# Sample usage
pq = PriorityQueue()
pq.enqueue(”Task A“, 3)
pq.enqueue(“Task B“, 1)
pq.enqueue(“Task C“, 2)

pq.display()
print("Dequeued:“, pq.dequeue())
pq.display()
```

```
Priority Queue: [(1, 'Task B'), (3, 'Task A'), (2, 'Task C')]
Dequeued: Task B
Priority Queue: [(2, 'Task C'), (3, 'Task A')]
```

TASK8

```
from collections import deque

class DequeDS:

    Double-Ended Queue (Deque) implementation usiug collections.deque.
    Allows insertion and removal from both ends.


    def finite(self):

        Initializes an empty deque.

        self.deque = deque()

    def insert_front(self, item):

        Inserts an element at the front of the deque.

        Parameters:
        item (any): Element to be inserted

        self.deque.appendleft(item)

    de-F- insert_near( sel-I, item) :

        Inserts an element at the rear of the deque.

        Parameters:
        item (any): Element to be inserted

        self.deque.append(item)

    def remove_front(self):

        Removes and returns the front element of the deque.

        Returns:
        any: Front element, or None if deque is empty

        if self.is_empty():
            return None
        return self.deque.popleft()

    def remove_rear(self):

        Removes and returns the rear element of the deque.

        Returns :
        any: Rear element, or None if deque is empty

        1fse1f- .1s_empty():
            return  None
```

```
            return self.deque.pop()

        def is_empty(self):

            Checks whether the deque is empty.

            Returns :
            bool: True if empty, False otherwise

            return len(self.deque) == 0

        def display(self):

            Displays the elements of the deque.

            print( "Deque   contents : ",    11st( se1-£- . deque) )


    # Sample usage
    dq = DequeDS()
    dq.insert_front(10)
    dq.insert_rear(20)
    dq.insert_front(5)
    dq.display()

    print( "Removed  -Front: ",   dq. remove_front( ) )
    print( "Removed  rear: ",  dq. remove_rear( ) )
    dq. d1splay( )
```

```
Deque contents: [5, 10, 20]
Removed front: 5
Removed rear: 20
Deque contents: [10]
```

TASK 9

1. Student Attendance Tracking

Requirement:

Daily log of students entering and exiting

Maintain order of entry/exit

Fast insertion and removal at both ends

✅ Best Data Structure: Deque

Reason:

Entry —• insert at rear

Exit —• remove from front

Supports real-time tracking efficiently

Operations Complexity:

Insert / Remove —• 0(1)

📌 Why AI chose Deque: Deque handles both ends efficiently, unlike lists which are slower for front removals.

2. Event Registration System

Requirement:

Register participants

Ouick search, add, and remove

Avoid duplicates

✅ Best Data Structure: Hash Table (Dictionary / Set)

Reason:

Fast lookup by student ID

Efficient removal

Prevents duplicate registrations

Operations Complexity:

Insert / Search / Delete —• 0(1) average

📌 Why AI chose Hash Table: Hash tables are ideal for membership checks and fast updates.

3. Library Book Borrowing

Requirement:

Track available books

Store due dates

Ouick access by book ID

✅ Best Data Structure: Hash Table

Reason:

Book ID —• Key

Book details & due date —+ Value

Instant lookup and update

Operations Complexity:

Insert / Search / Update —• O(1)

📌 Why AI chose Hash Table: Perfect for key—value relationships like BookID —• DueDate.

4. Bus Scheduling System

Requirement:

Maintain bus routes

Store stop-to-stop connections

Support route traversal

✅ Best Data Structure: Graph (Adjacency List)

Reason:

Stops — Vertices

Routes —• Edges

Efficient for route mapping and traversal

Operations Complexity:

Traversal —+ 0(V + E)

📌 Why AI chose Graph: Graphs naturally represent transport networks and connections.

5. Cafeteria Order Oueue

Requirement:

Serve students in the order they arrive

First-Come-First-Served

✅ Best Data Structure: Oueue

Reason:

FIFO principle

Simple and efficient order handling

Operations Complexity:

Enqueue / Dequeue —• 0(1)

📌 Why AI chose Queue: Oueues are designed exactly for service-based systems.

📊 Summary Table System Module Data Structure Justification Student Attendance Deque Fast insert/remove from both ends Event Registration Hash Table Fast search, add, remove Library Borrowing Hash Table Instant book lookup Bus Scheduling Graph Represents routes & connections Cafeteria Orders Queue FIFO service order