Question 1(a) [3 marks]

Define best case, worst case and average case for time complexity.

Answer:

Table: Time Complexity Cases

Case Type	Definition	Example
Best Case	Minimum time needed for algorithm execution	Linear search finds element at first position
Worst Case	Maximum time needed for algorithm execution	Linear search finds element at last position
Average Case	Expected time for typical input scenarios	Linear search finds element in middle

- **Best Case**: Algorithm performs optimally with ideal input conditions
- **Worst Case**: Algorithm takes maximum possible time with unfavorable input
- **Average Case**: Mathematical expectation of execution time across all possible inputs

Mnemonic: "BWA - Best, Worst, Average"

Question 1(b) [4 marks]

What is Class and Object in OOP? Give suitable example.

Answer:

Table: Class vs Object

Aspect	Class	Object
Definition	Blueprint/template for creating objects	Instance of a class
Memory	No memory allocated	Memory allocated when created
Example	Car (template)	my_car = Car()

```python
# Class definition
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Object creation
student1 = Student("John", 20)
student1.display()
```

- **Class**: Template defining attributes and methods
- **Object**: Real instance with actual values

Mnemonic: "Class = Cookie Cutter, Object = Actual Cookie"

Question 1(c) [7 marks]

Write a program for two matrix multiplication using simple nested loop and numpy module.

Answer:

```python
# Method 1: Using Simple Nested Loop
def matrix_multiply_nested(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])

    # Initialize result matrix
    result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]

    # Matrix multiplication
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result

# Method 2: Using NumPy
import numpy as np

def matrix_multiply_numpy(A, B):
    A_np = np.array(A)
    B_np = np.array(B)
    return np.dot(A_np, B_np)

# Example usage
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]
```

```
print("Nested Loop Result:", matrix_multiply_nested(A, B))
print("NumPy Result:", matrix_multiply_numpy(A, B))
```

- **Nested Loop**: Three loops for row, column, and multiplication
- **NumPy**: Built-in dot() function for efficient multiplication

Mnemonic: "Row × Column = Result"

Question 1(c) OR [7 marks]

Write a program to implement basic operations on arrays.

Answer:

```
import array

# Create array
arr = array.array('i', [1, 2, 3, 4, 5])

def array_operations():
    print("Original array:", arr)

    # Insert element
    arr.insert(2, 10)
    print("After insert(2, 10):", arr)

    # Append element
    arr.append(6)
    print("After append(6):", arr)

    # Remove element
    arr.remove(10)
    print("After remove(10):", arr)

    # Pop element
    popped = arr.pop()
    print(f"Popped element: {popped}, Array: {arr}")

    # Search element
    index = arr.index(3)
    print(f"Index of 3: {index}")

    # Count occurrences
    count = arr.count(2)
    print(f"Count of 2: {count}")

array_operations()
```

Table: Array Operations

Operation	Method	Description
Insert	insert(index, value)	Add element at specific position
Append	append(value)	Add element at end
Remove	remove(value)	Remove first occurrence
Pop	pop()	Remove and return last element

Mnemonic: "IARP - Insert, Append, Remove, Pop"

Question 2(a) [3 marks]

Explain Big 'O' Notation.

Answer:

Table: Big O Complexity

Notation	Name	Example
O(1)	Constant	Array access
O(n)	Linear	Linear search
O(n²)	Quadratic	Bubble sort
O(log n)	Logarithmic	Binary search

- **Big O**: Describes upper bound of algorithm's time complexity
- **Purpose**: Compare efficiency of different algorithms
- **Focus**: Worst-case scenario analysis

Mnemonic: "Big O = Big Order of growth"

Question 2(b) [4 marks]

Differentiate between class method and static method.

Answer:

Table: Method Types Comparison

Aspect	Class Method	Static Method
Decorator	@classmethod	@staticmethod
First Parameter	cls (class reference)	No special parameter
Access	Can access class variables	Cannot access class/instance variables
Usage	Alternative constructors	Utility functions

```
class MyClass:
    class_var = "I am class variable"

    @classmethod
    def class_method(cls):
        return f"Class method accessing: {cls.class_var}"

    @staticmethod
    def static_method():
        return "Static method - no class access"

# Usage
print(MyClass.class_method())
print(MyClass.static_method())
```

Mnemonic: "Class method has CLS, Static method is STandalone"

Question 2(c) [7 marks]

Implement a class for single level inheritance using public and private type derivation.

Answer:

```
# Base class
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand          # Public attribute
        self._model = model         # Protected attribute
        self.__year = 2023          # Private attribute

    def start_engine(self):
        return f"{self.brand} engine started"

    def _display_model(self):       # Protected method
        return f"Model: {self._model}"

    def __private_method(self):     # Private method
        return f"Year: {self.__year}"

# Derived class (Single level inheritance)
class Car(Vehicle):
```

```
    def __init__(self, brand, model, doors):
        super().__init__(brand, model)
        self.doors = doors

    def car_info(self):
        # Can access public and protected members
        return f"Car: {self.brand}, {self._display_model()}, Doors: {self.doors}"

    def demonstrate_access(self):
        print("Public access:", self.brand)
        print("Protected access:", self._model)
        # print("Private access:", self.__year)  # This would cause error

# Usage
my_car = Car("Toyota", "Camry", 4)
print(my_car.car_info())
print(my_car.start_engine())
my_car.demonstrate_access()
```

- **Public**: Accessible everywhere (brand)
- **Protected**: Accessible in class and subclasses (_model)
- **Private**: Only accessible within same class (__year)

Mnemonic: "Public = Everyone, Protected = Family, Private = Personal"

Question 2(a) OR [3 marks]

Explain constructor with example.

Answer:

Table: Constructor Types

Type	Method	Purpose
Default	`__init__(self)`	Initialize with default values
Parameterized	`__init__(self, params)`	Initialize with custom values

```python
class Student:
    def __init__(self, name="Unknown", age=18):  # Constructor
        self.name = name
        self.age = age
        print(f"Student {name} created")

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Object creation calls constructor automatically
s1 = Student("Alice", 20)
s2 = Student()  # Uses default values
```

- **Constructor**: Special method called when object is created
- **Purpose**: Initialize object attributes
- **Automatic**: Called automatically during object creation

Mnemonic: "Constructor = Object's Birth Certificate"

Question 2(b) OR [4 marks]

Write a program to demonstrate Polymorphism.

Answer:

```python
# Base class
class Animal:
    def make_sound(self):
        pass

# Derived classes
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

class Cow(Animal):
    def make_sound(self):
        return "Moo!"

# Polymorphism demonstration
def animal_sound(animal):
    return animal.make_sound()

# Creating objects
animals = [Dog(), Cat(), Cow()]
```

```
# Same method call, different behavior
for animal in animals:
    print(f"{animal.__class__.__name__}: {animal_sound(animal)}")
```

Table: Polymorphism Benefits

Benefit	Description
Flexibility	Same interface, different implementations
Maintainability	Easy to add new types
Code Reuse	Common interface for different objects

Mnemonic: "Poly = Many, Morph = Forms"

Question 2(c) OR [7 marks]

Write a Python to implement multiple and hierarchical inheritance.

Answer:

```
# Multiple Inheritance
class Teacher:
    def __init__(self, subject):
        self.subject = subject

    def teach(self):
        return f"Teaching {self.subject}"

class Researcher:
    def __init__(self, field):
        self.field = field

    def research(self):
        return f"Researching in {self.field}"

# Multiple inheritance
class Professor(Teacher, Researcher):
    def __init__(self, name, subject, field):
        self.name = name
        Teacher.__init__(self, subject)
        Researcher.__init__(self, field)

    def profile(self):
        return f"Prof. {self.name}: {self.teach()} and {self.research()}"

# Hierarchical Inheritance
class Vehicle:
    def __init__(self, brand):
        self.brand = brand
```

```
    def start(self):
        return f"{self.brand} started"

class Car(Vehicle):
    def drive(self):
        return f"{self.brand} car driving"

class Bike(Vehicle):
    def ride(self):
        return f"{self.brand} bike riding"

# Usage
prof = Professor("Smith", "Python", "AI")
print(prof.profile())

car = Car("Honda")
bike = Bike("Yamaha")
print(car.drive())
print(bike.ride())
```

Diagram:

```
Multiple Inheritance:    Hierarchical Inheritance:
   Teacher  Researcher          Vehicle
      \        /                 /    \
       \      /               Car    Bike
      Professor
```

Mnemonic: "Multiple = Many Parents, Hierarchical = Tree Structure"

Question 3(a) [3 marks]

Explain Push and Pop operations on Stack.

Answer:

Table: Stack Operations

Operation	Description	Time Complexity
Push	Add element to top	O(1)
Pop	Remove element from top	O(1)
Peek/Top	View top element	O(1)
isEmpty	Check if stack is empty	O(1)

```python
stack = []

# Push operation
stack.append(10)  # Push 10
stack.append(20)  # Push 20
print("After push:", stack)  # [10, 20]

# Pop operation
item = stack.pop()  # Pop 20
print(f"Popped: {item}, Stack: {stack}")  # [10]
```

- **LIFO**: Last In, First Out principle
- **Top**: Only accessible element for operations

Mnemonic: "Stack = Plate Stack - Last plate In, First plate Out"

Question 3(b) [4 marks]

Explain Enqueue and Dequeue operations on Queue.

Answer:

Table: Queue Operations

Operation	Description	Position	Time Complexity
Enqueue	Add element	Rear	O(1)
Dequeue	Remove element	Front	O(1)
Front	View front element	Front	O(1)
Rear	View rear element	Rear	O(1)

```python
from collections import deque

queue = deque()

# Enqueue operation
queue.append(10)  # Enqueue 10
queue.append(20)  # Enqueue 20
print("After enqueue:", list(queue))  # [10, 20]

# Dequeue operation
item = queue.popleft()  # Dequeue 10
print(f"Dequeued: {item}, Queue: {list(queue)}")  # [20]
```

- **FIFO**: First In, First Out principle
- **Two ends**: Front for removal, Rear for insertion

Mnemonic: "Queue = Line at Store - First person In, First person Out"

Question 3(c) [7 marks]

Explain various applications of Stack.

Answer:

Table: Stack Applications

Application	Description	Example
Expression Evaluation	Convert infix to postfix	(a+b)*c* → *ab*+*c*
Function Calls	Manage function call sequence	Recursion handling
Undo Operations	Reverse recent actions	Text editor undo
Browser History	Navigate back through pages	Back button
Parentheses Matching	Check balanced brackets	{[()]} validation

```
# Example: Parentheses matching
def is_balanced(expression):
    stack = []
    pairs = {'(': ')', '[': ']', '{': '}'}

    for char in expression:
        if char in pairs:  # Opening bracket
            stack.append(char)
        elif char in pairs.values():  # Closing bracket
            if not stack:
                return False
            if pairs[stack.pop()] != char:
                return False

    return len(stack) == 0

# Test
print(is_balanced("({[]})"))  # True
print(is_balanced("({[}])"))  # False
```

- **Memory Management**: Function call stack in programming
- **Backtracking**: Maze solving, game algorithms
- **Compiler Design**: Syntax analysis and parsing

Mnemonic: "Stack Applications = UFPB (Undo, Function, Parentheses, Browser)"

Question 3(a) OR [3 marks]

List out limitations of Single Queue.

Answer:

Table: Single Queue Limitations

Limitation	Description	Problem
Memory Wastage	Front space becomes unusable	Inefficient memory usage
Fixed Size	Cannot resize dynamically	Space constraints
False Overflow	Queue appears full when front space empty	Premature capacity limit
No Reuse	Dequeued positions not reusable	Linear space utilization

```
Single Queue Problem:
Front               Rear
  ↓                  ↓
[X][X][3][4][5][ ][ ][ ]
 ↑  ↑
Wasted space (cannot reuse)
```

- **Linear Implementation**: Cannot utilize dequeued space
- **Static Array**: Fixed size allocation

Mnemonic: "Single Queue = One-Way Street (No U-Turn)"

Question 3(b) OR [4 marks]

Differentiate circular and simple queues.

Answer:

Table: Queue Types Comparison

Aspect	Simple Queue	Circular Queue
Memory Usage	Linear, wasteful	Circular, efficient
Space Reuse	No reuse of dequeued space	Reuses all positions
Overflow	False overflow possible	True overflow only
Implementation	Front and rear pointers	Front and rear with modulo

```
Simple Queue:          Circular Queue:
[X][X][3][4][ ][ ]    [5][6][3][4]
Front→    Rear→         ↑Rear Front→
(Wasted space)          (Space reused)
```

```
# Circular Queue Implementation
```

```python
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def enqueue(self, item):
        if (self.rear + 1) % self.size == self.front:
            print("Queue Full")
            return
        if self.front == -1:
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = item

    def dequeue(self):
        if self.front == -1:
            print("Queue Empty")
            return None
        item = self.queue[self.front]
        if self.front == self.rear:
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        return item
```

Mnemonic: "Circular = Ring Road (Continuous), Simple = Dead End Street"

Question 3(c) OR [7 marks]

Convert the following infix expression into postfix: (a * b) * (c ^ (d + e) – f)

Answer:

Table: Operator Precedence

Operator	Precedence	Associativity
^	3	Right to Left
*, /	2	Left to Right
+, -	1	Left to Right

Step-by-step conversion:

```
Expression: (a * b) * (c ^ (d + e) – f)

Step 1: (a * b) → ab*
Step 2: (d + e) → de+
Step 3: c ^ (de+) → c de+ ^
Step 4: (c de+ ^) – f → c de+ ^ f –
Step 5: (ab*) * (c de+ ^ f –) → ab* c de+ ^ f – *

Final Answer: ab*cde+^f-*
```

Algorithm:

1. **Operand**: Add to output
2. **'('**: Push to stack
3. **')'**: Pop until '('
4. **Operator**: Pop higher/equal precedence, then push
5. **End**: Pop all remaining operators

```python
def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack = []
    output = []

    for char in expression:
        if char.isalnum():
            output.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # Remove '('
        elif char in precedence:
            while (stack and stack[-1] != '(' and
                   stack[-1] in precedence and
                   precedence[stack[-1]] >= precedence[char]):
                output.append(stack.pop())
            stack.append(char)

    while stack:
        output.append(stack.pop())

    return ''.join(output)

# Test
result = infix_to_postfix("(a*b)*(c^(d+e)-f)")
print("Postfix:", result)  # ab*cde+^f-*
```

Mnemonic: "PEMDAS for precedence, Stack for operators"

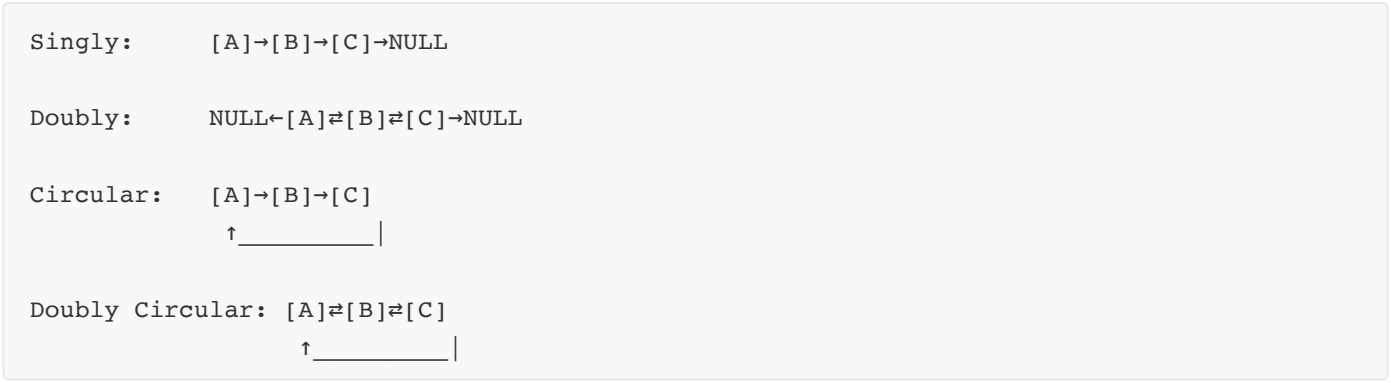
Question 4(a) [3 marks]

List types of Linked List.

Answer:

Table: Linked List Types

Type	Description	Key Feature
Singly Linked	One pointer to next node	Forward traversal only
Doubly Linked	Pointers to next and previous	Bidirectional traversal
Circular Linked	Last node points to first	No NULL pointer
Doubly Circular	Doubly + Circular features	Both directions + circular



- **Memory**: Each node contains data and pointer(s)
- **Dynamic**: Size can change during runtime

Mnemonic: "SDCD - Singly, Doubly, Circular, Doubly-Circular"

Question 4(b) [4 marks]

Differentiate between circular linked list and singly linked list.

Answer:

Table: Singly vs Circular Linked List

Aspect	Singly Linked List	Circular Linked List
Last Node	Points to NULL	Points to first node
Traversal	Ends at NULL	Continuous loop
Memory	Last node stores NULL	No NULL pointer
Detection	Check for NULL	Check for starting node

```python
# Singly Linked List Node
class SinglyNode:
    def __init__(self, data):
        self.data = data
        self.next = None

# Circular Linked List Node
class CircularNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverse_singly(head):
    current = head
    while current:  # Stops at NULL
        print(current.data)
        current = current.next

def traverse_circular(head):
    if not head:
        return
    current = head
    while True:
        print(current.data)
        current = current.next
        if current == head:  # Back to start
            break
```

Diagram:

```
Singly:    [1]→[2]→[3]→NULL

Circular:  [1]→[2]→[3]
            ↑_______|
```

Mnemonic: "Singly = Dead End, Circular = Race Track"

Question 4(c) [7 marks]

Implement a program to perform following operation on singly linked list:
a. Insert a node at the beginning of a singly linked list.
b. Insert a node at the end of a singly linked list.

Answer:

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

class SinglyLinkedList:
```

```
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        """Insert node at the beginning"""
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        print(f"Inserted {data} at beginning")

    def insert_at_end(self, data):
        """Insert node at the end"""
        new_node = Node(data)

        if not self.head:  # Empty list
            self.head = new_node
            print(f"Inserted {data} at end (first node)")
            return

        # Traverse to last node
        current = self.head
        while current.next:
            current = current.next

        current.next = new_node
        print(f"Inserted {data} at end")

    def display(self):
        """Display the linked list"""
        if not self.head:
            print("List is empty")
            return

        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next

        print(" → ".join(elements) + " → NULL")

# Usage example
sll = SinglyLinkedList()

# Insert at beginning
sll.insert_at_beginning(10)
sll.insert_at_beginning(20)
sll.display()  # 20 → 10 → NULL

# Insert at end
sll.insert_at_end(30)
sll.insert_at_end(40)
```

```
sll.display()  # 20 → 10 → 30 → 40 → NULL
```

Table: Insertion Operations

Operation	Time Complexity	Steps
Beginning	O(1)	1. Create node 2. Point to head 3. Update head
End	O(n)	1. Create node 2. Traverse to end 3. Link last node

Mnemonic: "Beginning = Quick (O(1)), End = Journey (O(n))"

Question 4(a) OR [3 marks]

Explain doubly linked list.

Answer:

Table: Doubly Linked List Features

Feature	Description
Two Pointers	prev and next in each node
Bidirectional	Can traverse forward and backward
Memory	Extra space for prev pointer
Flexibility	Easy insertion/deletion anywhere

```
class DoublyNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

# Structure visualization
# NULL ← [prev|data|next] ⇄ [prev|data|next] → NULL
```

```
Doubly Linked List Structure:
NULL←[A]⇄[B]⇄[C]→NULL
     prev next
```

- **Advantages**: Bidirectional traversal, easier deletion
- **Disadvantages**: Extra memory for prev pointer

Mnemonic: "Doubly = Two-Way Street"

Question 4(b) OR [4 marks]

Describe applications of Linked List.

Answer:

Table: Linked List Applications

Application	Use Case	Benefit
Dynamic Arrays	When size varies	Efficient memory usage
Stack/Queue	LIFO/FIFO operations	Dynamic size
Graphs	Adjacency list representation	Space efficient
Music Playlist	Previous/Next songs	Easy navigation
Browser History	Back/Forward navigation	Dynamic history
Undo Operations	Text editors	Efficient undo/redo

```
# Example: Browser History using Doubly Linked List
class Page:
    def __init__(self, url):
        self.url = url
        self.prev = None
        self.next = None

class BrowserHistory:
    def __init__(self):
        self.current = None

    def visit(self, url):
        page = Page(url)
        if self.current:
            self.current.next = page
            page.prev = self.current
        self.current = page

    def back(self):
        if self.current and self.current.prev:
            self.current = self.current.prev
            return self.current.url
        return "No previous page"

    def forward(self):
        if self.current and self.current.next:
            self.current = self.current.next
            return self.current.url
        return "No next page"
```

Mnemonic: "Linked Lists = Dynamic, Flexible, Connected"

Question 4(c) OR [7 marks]

Implement Merge Sort algorithm.

Answer:

```python
def merge_sort(arr):
    """Merge Sort implementation"""
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort both halves
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # Merge the sorted halves
    return merge(left_sorted, right_sorted)

def merge(left, right):
    """Merge two sorted arrays"""
    result = []
    i = j = 0

    # Compare elements and merge
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])

    return result

# Example usage
def demonstrate_merge_sort():
    arr = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", arr)

    sorted_arr = merge_sort(arr)
    print("Sorted array:", sorted_arr)
```

```
demonstrate_merge_sort()
```

Table: Merge Sort Analysis

Aspect	Value
Time Complexity	O(n log n)
Space Complexity	O(n)
Stability	Stable
Type	Divide and Conquer

Algorithm Steps:

1. **Divide**: Split array into two halves
2. **Conquer**: Recursively sort both halves
3. **Combine**: Merge sorted halves

Mnemonic: "Merge Sort = Divide, Sort, Merge"

Question 5(a) [3 marks]

Describe applications of binary tree.

Answer:

Table: Binary Tree Applications

Application	Description	Example
Expression Trees	Mathematical expression representation	(a+b)*c
Decision Trees	Decision making in AI/ML	Classification algorithms
File Systems	Directory structure organization	Folder hierarchy
Database Indexing	B-trees for efficient searching	Database indices
Huffman Coding	Data compression technique	File compression
Heap Operations	Priority queues implementation	Task scheduling

- **Hierarchical Data**: Naturally represents tree-like structures
- **Efficient Search**: Binary search trees provide O(log n) operations
- **Memory Management**: Used in compiler design for syntax trees

Mnemonic: "Binary Trees = EDFDHH (Expression, Decision, File, Database, Huffman, Heap)"

Question 5(b) [4 marks]

Explain Indegree and Outdegree of Binary Tree with example.

Answer:

Table: Degree Definitions

Term	Definition	Binary Tree Value
Indegree	Number of edges coming into a node	0 (root) or 1 (others)
Outdegree	Number of edges going out of a node	0, 1, or 2
Degree	Total edges connected to node	Indegree + Outdegree

```
Binary Tree Example:
        A (indegree=0, outdegree=2)
       / \
      B   C (indegree=1, outdegree=1)
     /   /
    D   E (indegree=1, outdegree=0)
```

- **Root Node**: Always has indegree = 0
- **Leaf Nodes**: Always have outdegree = 0
- **Internal Nodes**: Have outdegree = 1 or 2

Table: Example Analysis

Node	Indegree	Outdegree	Node Type
A	0	2	Root
B	1	1	Internal
C	1	1	Internal
D	1	0	Leaf
E	1	0	Leaf

Mnemonic: "In = Coming In, Out = Going Out"

Question 5(c) [7 marks]

Write a program to implement construction of binary search trees.

Answer:

```python
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
```

```
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, data):
        """Insert a node in BST"""
        if self.root is None:
            self.root = TreeNode(data)
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
            else:
                self._insert_recursive(node.left, data)
        elif data > node.data:
            if node.right is None:
                node.right = TreeNode(data)
            else:
                self._insert_recursive(node.right, data)

    def search(self, data):
        """Search for a node in BST"""
        return self._search_recursive(self.root, data)

    def _search_recursive(self, node, data):
        if node is None or node.data == data:
            return node

        if data < node.data:
            return self._search_recursive(node.left, data)
        else:
            return self._search_recursive(node.right, data)

    def inorder_traversal(self):
        """Inorder traversal (Left, Root, Right)"""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.data)
            self._inorder_recursive(node.right, result)

    def display_tree(self):
        """Simple tree display"""
```

```
        if self.root:
            self._display_recursive(self.root, 0)

    def _display_recursive(self, node, level):
        if node:
            self._display_recursive(node.right, level + 1)
            print("    " * level + str(node.data))
            self._display_recursive(node.left, level + 1)

# Example usage
bst = BinarySearchTree()
values = [50, 30, 70, 20, 40, 60, 80]

print("Inserting values:", values)
for value in values:
    bst.insert(value)

print("\nTree structure:")
bst.display_tree()

print("\nInorder traversal:", bst.inorder_traversal())

# Search examples
print(f"\nSearch 40: {'Found' if bst.search(40) else 'Not Found'}")
print(f"Search 90: {'Found' if bst.search(90) else 'Not Found'}")
```

Table: BST Operations

Operation	Time Complexity	Description
Insert	O(log n) average, O(n) worst	Add new node
Search	O(log n) average, O(n) worst	Find specific node
Delete	O(log n) average, O(n) worst	Remove node
Traversal	O(n)	Visit all nodes

Mnemonic: "BST Rule = Left < Root < Right"

Question 5(a) OR [3 marks]

Define level, degree and leaf node in binary tree.

Answer:

Table: Binary Tree Terms

Term	Definition	Example
Level	Distance from root (root = level 0)	Root=0, Children=1, etc.
Degree	Number of children a node has	0, 1, or 2
Leaf Node	Node with no children (degree = 0)	Terminal nodes

```
Binary Tree with Levels:
Level 0:        A
               / \
Level 1:      B   C
             /   / \
Level 2:    D   E   F
```

Table: Example Analysis

Node	Level	Degree	Type
A	0	2	Root
B	1	1	Internal
C	1	2	Internal
D	2	0	Leaf
E	2	0	Leaf
F	2	0	Leaf

- **Height**: Maximum level in tree
- **Depth**: Same as level for a node

Mnemonic: "Level = Floor number, Degree = Children count, Leaf = No children"

Question 5(b) OR [4 marks]

Explain complete binary tree with example.

Answer:

Table: Binary Tree Types

Type	Description	Property
Complete	All levels filled except last, left-filled	Efficient array representation
Full	Every node has 0 or 2 children	No single-child nodes
Perfect	All levels completely filled	2^h - 1 nodes

```
Complete Binary Tree:
        A
       / \
      B   C
     / \ /
    D  E F

NOT Complete:
        A
       / \
      B   C
       \ / \
        E F G
```

```python
class CompleteBinaryTree:
    def __init__(self):
        self.tree = []

    def insert(self, data):
        """Insert in complete binary tree manner"""
        self.tree.append(data)

    def get_parent_index(self, i):
        return (i - 1) // 2

    def get_left_child_index(self, i):
        return 2 * i + 1

    def get_right_child_index(self, i):
        return 2 * i + 2

    def display_level_order(self):
        """Display tree level by level"""
        if not self.tree:
            return

        level = 0
        while (2 ** level) <= len(self.tree):
            start = 2 ** level - 1
            end = min(2 ** (level + 1) - 1, len(self.tree))
            print(f"Level {level}: {self.tree[start:end]}")
            level += 1

# Example
cbt = CompleteBinaryTree()
for i in [1, 2, 3, 4, 5, 6]:
    cbt.insert(i)

cbt.display_level_order()
```

Properties:

- **Array Representation**: Parent at i, children at 2i+1 and 2i+2
- **Heap Property**: Forms foundation for heap data structure

Mnemonic: "Complete = All floors full except last, filled left to right"

Question 5(c) OR [7 marks]

Construct a Binary Search Tree (BST) for the following sequence of numbers: 50, 70, 60, 20, 90, 10, 40, 100

Answer:

Step-by-step BST Construction:

```
Insert 50: (Root)
    50

Insert 70: (70 > 50, go right)
    50
     \
      70

Insert 60: (60 > 50, go right; 60 < 70, go left)
    50
     \
      70
     /
    60

Insert 20: (20 < 50, go left)
    50
   /  \
  20   70
       /
      60

Insert 90: (90 > 50, right; 90 > 70, right)
    50
   /  \
  20   70
       / \
      60  90

Insert 10: (10 < 50, left; 10 < 20, left)
    50
   /  \
  20   70
 /    / \
10   60  90
```

```
Insert 40: (40 < 50, left; 40 > 20, right)
    50
   /  \
  20   70
 / \   / \
10  40 60 90

Insert 100: (100 > 50, right; 100 > 70, right; 100 > 90, right)
    50
   /  \
  20   70
 / \   / \
10  40 60 90
             \
              100
```

Final BST Structure:

```
        50
       /  \
      20   70
     / \   / \
    10  40 60 90
              \
               100
```

```python
# Implementation code for the construction
class BST:
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = TreeNode(data)
            print(f"Inserted {data} as root")
        else:
            self._insert(self.root, data)

    def _insert(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
                print(f"Inserted {data} to left of {node.data}")
            else:
                self._insert(node.left, data)
        else:
            if node.right is None:
                node.right = TreeNode(data)
                print(f"Inserted {data} to right of {node.data}")
            else:
                self._insert(node.right, data)
```

```
# Construct the BST
bst = BST()
sequence = [50, 70, 60, 20, 90, 10, 40, 100]

for num in sequence:
    bst.insert(num)
```

Table: Traversal Results

Traversal	Result
Inorder	10, 20, 40, 50, 60, 70, 90, 100
Preorder	50, 20, 10, 40, 70, 60, 90, 100
Postorder	10, 40, 20, 60, 100, 90, 70, 50

Properties Verified:

- **BST Property**: Left subtree < Root < Right subtree
- **Inorder**: Gives sorted sequence

Mnemonic: "BST Construction = Compare, Choose direction, Insert"