Question 1(a) [3 marks]

Define Big - O Notation, Big Omega Notation, Big Theta Notation.

Answer:

Table: Asymptotic Notations Comparison

Notation	Symbol	Description	Usage
Big-O	O(f(n))	Upper bound	Worst case
Big Omega	Ω(f(n))	Lower bound	Best case
Big Theta	Θ(f(n))	Tight bound	Average case

- **Big-O Notation**: Describes maximum time/space complexity
- **Big Omega**: Describes minimum time/space complexity
- **Big Theta**: Describes exact time/space complexity

Mnemonic: "OWT - O for wOrst, Omega for Best, Theta for Tight"

Question 1(b) [4 marks]

Define Set. Write various operations that can be performed on Set.

Answer:

Definition: Set is a collection of unique elements with no duplicates.

Table: Set Operations

Operation	Symbol	Description	Example
Union	A ∪ B	Combines all elements	{1,2} ∪ {2,3} = {1,2,3}
Intersection	A ∩ B	Common elements	{1,2} ∩ {2,3} = {2}
Difference	A - B	Elements in A not in B	{1,2} - {2,3} = {1}
Subset	A ⊆ B	All A elements in B	{1} ⊆ {1,2} = True

- **Add/Insert**: Adding new element
- **Remove/Delete**: Removing existing element
- **Contains**: Check if element exists

Mnemonic: "UIDS - Union, Intersection, Difference, Subset"

Question 1(c) [7 marks]

Write a Python class to represent a Cricketer. The class contains the name of the cricketer, team name and run as the data members. The member functions are as follows: to initialize the data members, to set run and display run.

Answer:

```python
class Cricketer:
    def __init__(self, name="", team="", run=0):
        self.name = name
        self.team = team
        self.run = run

    def set_run(self, run):
        self.run = run

    def display_run(self):
        print(f"Player: {self.name}")
        print(f"Team: {self.team}")
        print(f"Runs: {self.run}")

# Example usage
player = Cricketer("Virat Kohli", "India", 100)
player.display_run()
```

- **Constructor**: Initializes name, team, and run
- **set_run()**: Updates run value
- **display_run()**: Shows player information

Mnemonic: "CSD - Constructor, Set, Display"

Question 1(c OR) [7 marks]

Design a student class for reading and displaying the student information, the getInfo() and displayInfo() methods will be used respectively. Where getInfo() will be a private method.

Answer:

```python
class Student:
    def __init__(self):
        self.name = ""
        self.roll_no = ""
        self.marks = 0
        self.__getInfo()  # Private method call

    def __getInfo__(self):  # Private method
        self.name = input("Enter name: ")
        self.roll_no = input("Enter roll number: ")
        self.marks = int(input("Enter marks: "))

    def displayInfo(self):
```

```python
        print(f"Name: {self.name}")
        print(f"Roll No: {self.roll_no}")
        print(f"Marks: {self.marks}")

# Example usage
student = Student()
student.displayInfo()
```

- **Private method**: Uses double underscore (__getInfo)
- **Constructor**: Automatically calls private method
- **Public method**: displayInfo() shows student data

Mnemonic: "PCP - Private, Constructor, Public"

Question 2(a) [3 marks]

Differentiate between Stack and Queue.

Answer:

Table: Stack vs Queue Comparison

Feature	Stack	Queue
Order	LIFO (Last In First Out)	FIFO (First In First Out)
Operations	Push, Pop	Enqueue, Dequeue
Access Point	One end (top)	Two ends (front & rear)
Example	Plates stack	Bank queue

- **Stack**: Like book pile - last added, first removed
- **Queue**: Like waiting line - first come, first served

Mnemonic: "SLIF QFIF - Stack LIFO, Queue FIFO"

Question 2(b) [4 marks]

Define recursion. Explain with example.

Answer:

Definition: Function calling itself with smaller problem until base condition.

```python
def factorial(n):
    # Base case
    if n <= 1:
        return 1
    # Recursive case
    return n * factorial(n-1)

# Example: factorial(3)
# 3 * factorial(2)
# 3 * 2 * factorial(1)
# 3 * 2 * 1 = 6
```

- **Base case**: Stopping condition
- **Recursive case**: Function calls itself
- **Problem reduction**: Each call handles smaller problem

Mnemonic: "BRP - Base, Recursive, Problem-reduction"

Question 2(c) [7 marks]

Consider the size of the stack as 5. Apply the following operation on stack and show status and top pointer after each operation. Push a,b,c pop

Answer:

Stack Operations Trace:

```
Initial State:
Stack: [ _ _ _ _ _ ]  Top: -1
       0 1 2 3 4

After Push 'a':
Stack: [ a _ _ _ _ ]  Top: 0
       0 1 2 3 4

After Push 'b':
Stack: [ a b _ _ _ ]  Top: 1
       0 1 2 3 4

After Push 'c':
Stack: [ a b c _ _ ]  Top: 2
       0 1 2 3 4

After Pop:
Stack: [ a b _ _ _ ]  Top: 1
       0 1 2 3 4
Popped element: c
```

- **Push operations**: Add elements from index 0 onwards

- **Top pointer**: Points to last inserted element
- **Pop operation**: Removes top element, decrements top pointer

Mnemonic: "PTD - Push Top Decrement"

Question 2(a OR) [3 marks]

List applications of Stack and Queue.

Answer:

Table: Applications of Stack and Queue

Data Structure	Applications
Stack	Function calls, Undo operations, Expression evaluation, Browser history
Queue	Process scheduling, Printer queue, BFS traversal, Handling requests

- **Stack applications**: Undo-redo, recursion, parsing
- **Queue applications**: Task scheduling, buffering, breadth-first search

Mnemonic: "Stack FUBE, Queue SPBH"

Question 2(b OR) [4 marks]

Convert following algebraic expression into postfix notation using Stack: i) (a*b)*(c^d(d+e)-f) ii) a-b/(c*d/e)

Answer:

i) (a*b)*(c^d(d+e)-f)

Symbol	Stack	Output
(	(	
a	(	a
*	(*	a
b	(*	ab
)		ab*
*	*	ab*
(	*(	ab*
c	*(	ab*c
^	*(^	ab*c
d	*(^	ab*cd
(	*(^(	ab*cd
d	*(^(	ab*cdd
+	*(^(+	ab*cdd
e	*(^(+	ab*cdde
)	*(^	ab*cdde+
)	*	ab*cdde+^
-	*-	ab*cdde+^
f	*-	ab*cdde+^f
		ab*cdde+^f-*

Result: ab*cdde+^f-*

ii) a-b/(c*d/e)

Result: abcd*e/-

Mnemonic: "PEMDAS reversed for postfix"

Question 2(c OR) [7 marks]

Develop a program to implement a queue using a list that performs following operations: enqueue, dequeue.

Answer:

```python
class Queue:
    def __init__(self):
        self.queue = []
        self.front = 0
        self.rear = -1

    def enqueue(self, item):
        self.queue.append(item)
        self.rear += 1
        print(f"Enqueued: {item}")

    def dequeue(self):
        if self.front <= self.rear:
            item = self.queue[self.front]
            self.front += 1
            print(f"Dequeued: {item}")
            return item
        else:
            print("Queue is empty")
            return None

    def display(self):
        if self.front <= self.rear:
            print("Queue:", self.queue[self.front:self.rear+1])
        else:
            print("Queue is empty")

# Example usage
q = Queue()
q.enqueue('A')
q.enqueue('B')
q.dequeue()
q.display()
```

- **Enqueue**: Add element at rear
- **Dequeue**: Remove element from front
- **FIFO principle**: First In, First Out

Mnemonic: "ERF - Enqueue Rear, Front"

Question 3(a) [3 marks]

List types of linked lists. Give graphical representation of each type.

Answer:

Table: Types of Linked Lists

Type	Description	Diagram
Singly	One direction pointer	A→B→C→NULL
Doubly	Two direction pointers	NULL←A⇄B⇄C→NULL
Circular	Last points to first	A→B→C→A

```
Singly Linked List:
[Data|Next] -> [Data|Next] -> [Data|NULL]

Doubly Linked List:
[Prev|Data|Next] <-> [Prev|Data|Next] <-> [Prev|Data|Next]

Circular Linked List:
[Data|Next] -> [Data|Next] -> [Data|Next]
    ^                              |
    |______________________________|
```

Mnemonic: "SDC - Singly, Doubly, Circular"

Question 3(b) [4 marks]

Write an algorithm to search a given node in a singly link list.

Answer:

```python
def search_node(head, key):
    current = head
    position = 0

    while current is not None:
        if current.data == key:
            return position
        current = current.next
        position += 1

    return -1  # Not found

# Algorithm steps:
# 1. Start from head
# 2. Compare current data with key
# 3. If found, return position
# 4. Move to next node
# 5. Repeat until end
```

- **Linear search**: Traverse from head to tail
- **Time complexity**: O(n)
- **Return**: Position if found, -1 if not found

Mnemonic: "SCMR - Start, Compare, Move, Return"

Question 3(c) [7 marks]

Implement program to perform following operation on singly linked list: 1)Insert a node at the beginning of a singly linked list. 2)Delete a node from the beginning 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):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        print(f"Inserted {data} at beginning")

    def delete_from_beginning(self):
        if self.head is None:
            print("List is empty")
            return None

        deleted_data = self.head.data
        self.head = self.head.next
        print(f"Deleted {deleted_data} from beginning")
        return deleted_data

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("NULL")

# Example usage
ll = SinglyLinkedList()
ll.insert_at_beginning(10)
ll.insert_at_beginning(20)
ll.delete_from_beginning()
ll.display()
```

- **Insert**: Create node, link to head, update head
- **Delete**: Store data, move head to next, return data

Mnemonic: "CLU - Create, Link, Update"

Question 3(a OR) [3 marks]

Differentiate between circular linked list and singly linked list.

Answer:

Table: Circular vs Singly Linked List

Feature	Singly Linked List	Circular Linked List
Last node points to	NULL	First node (head)
Traversal	Linear (one direction)	Circular (continuous)
End detection	next == NULL	next == head
Memory	Less (no extra pointer)	Same structure

- **Circular advantage**: No NULL pointers, continuous traversal
- **Singly advantage**: Simple implementation, clear end point

Mnemonic: "CNTE - Circular No Termination End"

Question 3(b OR) [4 marks]

Explain three applications of linked list in brief.

Answer:

Table: Linked List Applications

Application	Description	Advantage
Dynamic memory allocation	Manage memory blocks	Efficient memory usage
Implementation of stacks/queues	Using linked structure	Dynamic size
Polynomial representation	Store coefficients and powers	Easy arithmetic operations

- **Music playlist**: Add/remove songs dynamically
- **Browser history**: Navigate back/forward
- **Image viewer**: Previous/next image navigation

Mnemonic: "DIP - Dynamic, Implementation, Polynomial"

Question 3(c OR) [7 marks]

Implement a program to create and display circular linked lists.

Answer:

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

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            current = self.head
            while current.next != self.head:
                current = current.next
            current.next = new_node
            new_node.next = self.head

    def display(self):
        if self.head is None:
            print("List is empty")
            return

        current = self.head
        print("Circular List:")
        while True:
            print(current.data, end=" -> ")
            current = current.next
            if current == self.head:
                break
        print(f"{self.head.data} (back to head)")

# Example usage
cll = CircularLinkedList()
cll.insert(10)
cll.insert(20)
cll.insert(30)
cll.display()
```

- **Creation**: Link last node to head
- **Display**: Stop when reaching head again

Mnemonic: "CLH - Create, Link, Head"

Question 4(a) [3 marks]

Write a program for Selection Sort Method.

Answer:

```python
def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr

# Example usage
data = [64, 34, 25, 12, 22]
sorted_data = selection_sort(data)
print("Sorted array:", sorted_data)
```

- **Find minimum**: In unsorted portion
- **Swap**: With first unsorted element
- **Time complexity**: $O(n^2)$

Mnemonic: "FMS - Find, Minimum, Swap"

Question 4(b) [4 marks]

Apply Insertion sort to following data to arrange them in ascending order. 25 15 35 20 30 5 10

Answer:

Insertion Sort Steps:

```
Initial: [25, 15, 35, 20, 30, 5, 10]

Pass 1: [15, 25, 35, 20, 30, 5, 10]  (Insert 15)
Pass 2: [15, 25, 35, 20, 30, 5, 10]  (35 in place)
Pass 3: [15, 20, 25, 35, 30, 5, 10]  (Insert 20)
Pass 4: [15, 20, 25, 30, 35, 5, 10]  (Insert 30)
Pass 5: [5, 15, 20, 25, 30, 35, 10]  (Insert 5)
Pass 6: [5, 10, 15, 20, 25, 30, 35]  (Insert 10)

Final: [5, 10, 15, 20, 25, 30, 35]
```

- **Method**: Take element, find position in sorted part
- **Comparisons**: 15 total comparisons
- **Shifts**: Elements moved to make space

Mnemonic: "TFI - Take, Find, Insert"

Question 4(c) [7 marks]

Implement a python program to search a particular element from a list using Linear Search.

Answer:

```python
def linear_search(arr, target):
    comparisons = 0

    for i in range(len(arr)):
        comparisons += 1
        if arr[i] == target:
            print(f"Element {target} found at index {i}")
            print(f"Number of comparisons: {comparisons}")
            return i

    print(f"Element {target} not found")
    print(f"Number of comparisons: {comparisons}")
    return -1

def linear_search_all_positions(arr, target):
    positions = []
    for i in range(len(arr)):
        if arr[i] == target:
            positions.append(i)
    return positions

# Example usage
data = [10, 25, 30, 15, 20, 30, 35]
target = 30

result = linear_search(data, target)
all_positions = linear_search_all_positions(data, target)
print(f"All positions of {target}: {all_positions}")
```

- **Sequential search**: Check each element one by one
- **Time complexity**: O(n) worst case
- **Best case**: O(1) if found at first position

Mnemonic: "CEO - Check Each One"

Question 4(a OR) [3 marks]

Write a program of Insertion Sort Method.

Answer:

```python
def insertion_sort(arr):
```

```python
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

    return arr

# Example usage
data = [12, 11, 13, 5, 6]
print("Original:", data)
sorted_data = insertion_sort(data.copy())
print("Sorted:", sorted_data)
```

- **Key element**: Current element to be inserted
- **Shift right**: Larger elements move right
- **Insert**: Key at correct position

Mnemonic: "KSI - Key, Shift, Insert"

Question 4(b OR) [4 marks]

Apply Quick Sort to the following data and arrange them in the proper manner. 5 6 1 8 2 9 10 15 7 13

Answer:

Quick Sort Steps:

```
Initial: [5, 6, 1, 8, 2, 9, 10, 15, 7, 13]
Pivot: 5 (first element)

Partition 1: [1, 2] 5 [6, 8, 9, 10, 15, 7, 13]

Left subarray [1, 2]:
Pivot: 1 → [] 1 [2]
Result: [1, 2]

Right subarray [6, 8, 9, 10, 15, 7, 13]:
Pivot: 6 → [] 6 [8, 9, 10, 15, 7, 13]

Continue partitioning...

Final: [1, 2, 5, 6, 7, 8, 9, 10, 13, 15]
```

- **Divide**: Choose pivot, partition around it
- **Conquer**: Recursively sort subarrays

- **Average time**: O(n log n)

Mnemonic: "DCC - Divide, Conquer, Combine"

Question 4(c OR) [7 marks]

Implement Merge sort algorithm.

Answer:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    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

    result.extend(left[i:])
    result.extend(right[j:])

    return result

# Example usage
data = [38, 27, 43, 3, 9, 82, 10]
sorted_data = merge_sort(data)
print("Sorted array:", sorted_data)
```

- **Divide**: Split array into halves
- **Merge**: Combine sorted subarrays
- **Time complexity**: O(n log n) always

Mnemonic: "DSM - Divide, Sort, Merge"

Question 5(a) [3 marks]

Write Short note on: Applications of Tree.

Answer:

Table: Tree Applications

Application	Description	Example
File systems	Directory structure	Folders and files
Expression parsing	Mathematical expressions	(a+b)*c
Database indexing	Fast data retrieval	B-trees in databases

- **Decision trees**: AI and machine learning
- **Huffman coding**: Data compression
- **Game trees**: Chess, tic-tac-toe

Mnemonic: "FED - File, Expression, Database"

Question 5(b) [4 marks]

Explain different Tree Traversal Methods.

Answer:

Table: Tree Traversal Methods

Method	Order	Process
Inorder	Left-Root-Right	LNR
Preorder	Root-Left-Right	NLR
Postorder	Left-Right-Root	LRN

```
Example Tree:
        A
       / \
      B   C
     / \
    D   E

Inorder: D B E A C
Preorder: A B D E C
Postorder: D E B C A
```

- **Inorder**: Gives sorted sequence for BST
- **Preorder**: Used for copying tree
- **Postorder**: Used for deleting tree

Mnemonic: "LNR PNL LRN for In-Pre-Post"

Question 5(c) [7 marks]

Write a menu driven program to perform the following operation on Binary Search Tree: Create a BST.

Answer:

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

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

    def insert(self, data):
        self.root = self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        if node is None:
            return TreeNode(data)

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

        return node

    def inorder(self, node):
        if node:
            self.inorder(node.left)
            print(node.data, end=" ")
            self.inorder(node.right)

def main():
    bst = BST()

    while True:
        print("\n1. Insert")
        print("2. Display (Inorder)")
        print("3. Exit")

        choice = int(input("Enter choice: "))

        if choice == 1:
            data = int(input("Enter data: "))
            bst.insert(data)
        elif choice == 2:
```

```
            print("BST (Inorder):", end=" ")
            bst.inorder(bst.root)
            print()
        elif choice == 3:
            break

if __name__ == "__main__":
    main()
```

- **BST property**: Left < Root < Right
- **Insertion**: Compare and go left/right
- **Menu driven**: User-friendly interface

Mnemonic: "CIM - Compare, Insert, Menu"

Question 5(a OR) [3 marks]

Define and give examples : Strict Binary Tree and Complete Binary Tree.

Answer:

Table: Binary Tree Types

Type	Definition	Example
Strict Binary Tree	Every node has 0 or 2 children	Each internal node has exactly 2 children
Complete Binary Tree	All levels filled except possibly last, filled left to right	Perfect structure till second last level

```
Strict Binary Tree:
        A
       / \
      B   C
         / \
        D   E

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

- **Strict**: No node with single child
- **Complete**: Optimal space utilization

Mnemonic: "SC - Strict Complete"

Question 5(b OR) [4 marks]

Explain basic terminology of Binary Tree : Level number, Degree, Indegree , Out-degree , Leaf Node.

Answer:

```
Binary Tree Example:
    Level 0:      A        (Root)
                 / \
   Level 1:     B   C
               / \   \
   Level 2:  D   E   F   (Leaves: D, E, F)
```

Table: Binary Tree Terminology

Term	Definition	Example
Level number	Distance from root (root = 0)	A=0, B=1, D=2
Degree	Number of children	A=2, B=2, C=1
Indegree	Number of incoming edges	All nodes = 1 (except root = 0)
Out-degree	Number of outgoing edges	Same as degree
Leaf Node	Node with no children	D, E, F

Mnemonic: "LDIOL - Level, Degree, In-Out, Leaf"

Question 5(c OR) [7 marks]

Write a menu driven program to perform the following operation on Binary Search Tree: Insert an element in BST.

Answer:

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

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

    def insert(self, data):
        if self.root is None:
            self.root = TreeNode(data)
            print(f"Root node {data} created")
        else:
```

```
            self._insert_helper(self.root, data)

    def _insert_helper(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_helper(node.left, data)
        elif data > node.data:
            if node.right is None:
                node.right = TreeNode(data)
                print(f"Inserted {data} to right of {node.data}")
            else:
                self._insert_helper(node.right, data)
        else:
            print(f"Data {data} already exists")

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

def main():
    bst = BST()

    while True:
        print("\n--- BST Operations ---")
        print("1. Insert Element")
        print("2. Display BST (Inorder)")
        print("3. Exit")

        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = int(input("Enter element to insert: "))
            bst.insert(data)
        elif choice == 2:
            result = []
            bst.display_inorder(bst.root, result)
            print("BST Elements (sorted):", result)
        elif choice == 3:
            print("Exiting...")
            break
        else:
            print("
            print("Invalid choice!")

if __name__ == "__main__":
    main()
```

- **Insert logic**: Compare with current node, go left/right
- **Recursive approach**: Clean and efficient implementation
- **Menu system**: Interactive user interface

Mnemonic: "CRL - Compare, Recursive, Left/right"