

Data Structure Using Python (313306)

Model Question Paper — Set B · Model Answers

Q.1

a) Walrus operator.

2 M

The **walrus operator** `:=` (assignment expression, Python 3.8+) assigns a value to a variable **inside an expression**, so the value is computed once and used immediately.

```
nums = [4, 9, 2, 7]
if (n := len(nums)) > 3:
    print("List has", n, "items")
```

Output:

```
List has 4 items
```

b) Dictionary.

2 M

A **dictionary** is a mutable collection of **key : value** pairs enclosed in `{ }`. Keys are unique and immutable; values are accessed by key, not by index.

```
student = {"roll": 12, "name": "Asha", "branch": "AN"}
print(student["name"])      # Asha
```

c) Class and object.

2 M

i) Class: A class is a user-defined blueprint (template) that defines data members (attributes) and methods common to all its objects. Created with the **class** keyword.

ii) Object: An object is an instance of a class; it has actual values for the attributes. e.g. `s1 = Student()`

d) Tree and graph.

2 M

i) Tree: A non-linear, hierarchical data structure consisting of a finite set of nodes with one special node called the **root**; the remaining nodes are divided into disjoint sub-trees. It has no cycles.

ii) Graph: A non-linear data structure $G = (V, E)$ consisting of a set of vertices V and a set of edges E connecting pairs of vertices. It may contain cycles.

e) Module vs package.

2 M

Module	Package
A single .py file containing functions, classes, variables.	A folder (directory) containing many modules and an <code>__init__.py</code> file.
Imported as: <code>import mymodule</code>	Imported as: <code>import mypkg.mymodule</code>
Example: <code>math</code> , <code>random</code>	Example: <code>numpy</code> , <code>mypkg</code>

(Any two, 1 M each)

f) Algorithm for preorder traversal.

2 M

PREORDER(ROOT) — Root, Left, Right

1. If ROOT is None, return.
2. Visit (print) ROOT.data.
3. PREORDER(ROOT.left).
4. PREORDER(ROOT.right).

g) Sorting techniques.

2 M

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Merge sort / Quick sort

(Any two, 1 M each)

Q.2

a) Data types in Python.

4 M

Data types: Numeric (int, float, complex), Boolean (bool), Sequence (str, list, tuple, range), Set (set, frozenset), Mapping (dict), None.

- **int:** whole numbers of any size, e.g. a = 25, b = -7.
- **list:** ordered, mutable collection in [], may hold mixed types, e.g. L = [1, "a", 2.5]; L[0] → 1.

```
a = 25           # int
b = 3.75        # float
c = 2 + 3j      # complex
s = "Python"    # str
L = [1, 2, 3]   # list
T = (4, 5)      # tuple
S = {6, 7}      # set
D = {"id": 1}   # dict
f = True        # bool
for v in (a, b, c, s, L, T, S, D, f):
    print(v, "->", type(v))
```

Output:

```
25 -> <class 'int'>
3.75 -> <class 'float'>
(2+3j) -> <class 'complex'>
Python -> <class 'str'>
[1, 2, 3] -> <class 'list'>
(4, 5) -> <class 'tuple'>
{6, 7} -> <class 'set'>
{'id': 1} -> <class 'dict'>
True -> <class 'bool'>
```

(List of data types 2 M, any two described 2 M)

b) User-defined arithmetic functions.

4 M

A user-defined function is created with **def**, takes parameters and returns a value with **return**.

```
# User-defined functions for basic arithmetic operations
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Cannot divide by zero"
    return a / b

x = int(input("Enter first number : "))
y = int(input("Enter second number: "))
print("Addition      =", add(x, y))
print("Subtraction   =", subtract(x, y))
print("Multiplication =", multiply(x, y))
print("Division      =", divide(x, y))
```

Output:

```
Enter first number : 45
Enter second number: 9
Addition      = 54
Subtraction   = 36
Multiplication = 405
Division      = 5.0
```

(Four functions 3 M, output 1 M)

c) Data hiding.

4 M

Data hiding means restricting direct access to the data members of a class from outside, so that data can be changed only through the methods of the class. It gives security and supports encapsulation.

- In Python, a member whose name starts with **double underscore** (e.g. `__balance`) is private.
- Python performs **name mangling**: `__balance` is stored as `_Account__balance`, so `obj.__balance` gives `AttributeError`.
- A single underscore (`_x`) means "protected" only by convention.

```
# Data hiding using private member (double underscore)
class Account:
    def __init__(self, owner, balance):
        self.owner = owner          # public
        self.__balance = balance    # private (hidden)

    def deposit(self, amt):          # controlled access
        if amt > 0:
            self.__balance += amt
```

```
def get_balance(self):
    return self.__balance

acc = Account("Om", 5000)
acc.deposit(1500)
print("Owner   :", acc.owner)
print("Balance :", acc.get_balance())
try:
    print(acc.__balance)          # direct access fails
except AttributeError as e:
    print("Error   :", e)
print("Name mangled:", acc._Account__balance)
```

Output:

```
Owner   : Om
Balance : 6500
Error   : 'Account' object has no attribute '__balance'
Name mangled: 6500
```

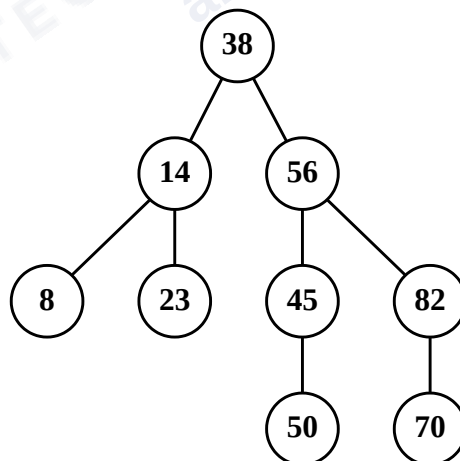
(Concept 2 M, example 2 M)

d) BST for 38, 14, 56, 8, 23, 45, 82, 70, 50.

4 M

A **binary search tree** is a binary tree in which every node's left sub-tree has smaller values and right sub-tree has greater values.

1. **38**: root
2. **14**: $14 < 38 \rightarrow$ left of 38
3. **56**: $56 > 38 \rightarrow$ right of 38
4. **8**: $8 < 38, 8 < 14 \rightarrow$ left of 14
5. **23**: $23 < 38, 23 > 14 \rightarrow$ right of 14
6. **45**: $45 > 38, 45 < 56 \rightarrow$ left of 56
7. **82**: $82 > 38, 82 > 56 \rightarrow$ right of 56
8. **70**: $70 > 38, 70 > 56, 70 < 82 \rightarrow$ left of 82
9. **50**: $50 > 38, 50 < 56, 50 > 45 \rightarrow$ right of 45



Final binary search tree

Preorder: 38, 14, 8, 23, 56, 45, 50, 82, 70

Postorder: 8, 23, 14, 50, 45, 70, 82, 56, 38

(Tree 2 M, each traversal 1 M)

Q.3

a) Relational operators.

4 M

Relational (comparison) operators compare two values and return a Boolean result (True / False).

Operator	Meaning	Example (a = 15, b = 9)	Result
==	Equal to	a == b	False
!=	Not equal to	a != b	True
>	Greater than	a > b	True
<	Less than	a < b	False
>=	Greater than or equal to	a >= 15	True
<=	Less than or equal to	b <= 8	False

```
a, b = 15, 9
print(a == b, a != b)
print(a > b, a < b)
print(a >= 15, b <= 8)
```

Output:

```
False True
True False
True False
```

(Any four operators with example, 1 M each)

b) Number pattern program.

4 M

```
# Print number pattern
rows = 5
for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

The outer loop controls rows; the inner loop prints 1 to i in row i.

(Logic 3 M, output 1 M)

c) Tree terminologies.

4 M

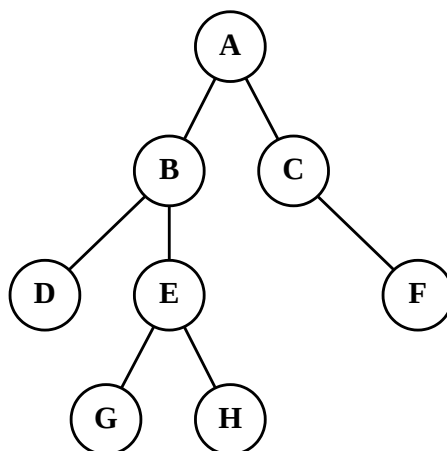


Fig.: Example tree (root A)

- **Root:** the topmost node with no parent — A.
- **Parent / child:** a node directly above another is its parent; E is the parent of G and H, which are its children.
- **Leaf node:** a node with no child — D, F, G, H.
- **Siblings:** nodes with the same parent — D and E; G and H.
- **Degree of node:** number of children — degree(B) = 2, degree(C) = 1.
- **Level:** root at level 0, children at level 1 ... — A = 0, B = 1, D = 2, E = 2, G = 3, H = 3, C = 1, F = 2.
- **Height / depth of tree:** maximum level of any node = 3.

(Any four terms with example, 1 M each)

d) Recursive binary search.

4 M

Binary search works on a **sorted** list: compare the key with the middle element; if smaller, search the left half, else the right half, until found or low > high.

```

# Recursive binary search
def binary_search(arr, low, high, key):
    if low > high:
        return -1                    # not found
    mid = (low + high) // 2
    print("low =", low, " high =", high, " mid =", mid, " arr[mid] =", arr[mid])
    if arr[mid] == key:
        return mid
    elif key < arr[mid]:
        return binary_search(arr, low, mid - 1, key)    # left half
    else:
        return binary_search(arr, mid + 1, high, key)   # right half

arr = [6, 14, 21, 33, 40, 52, 67, 75, 88]
key = 67
pos = binary_search(arr, 0, len(arr) - 1, key)
if pos != -1:
    print(key, "found at index", pos)
else:
    print(key, "not found")
  
```

Output:

```
low = 0  high = 8  mid = 4  arr[mid] = 40
low = 5  high = 8  mid = 6  arr[mid] = 67
67 found at index 6
```

(Program 3 M, output 1 M)

Q.4

a) Binary search vs linear search.

4 M

Linear search	Binary search
List may be sorted or unsorted.	List must be sorted.
Compares key with every element one by one.	Compares key with middle element and halves the list each time.
Time complexity $O(n)$.	Time complexity $O(\log n)$.
Simple; suitable for small lists.	Faster for large lists.
Works on linked list also.	Needs direct (index) access, e.g. array / list.

(Any four points, 1 M each)

b) Output of tuple code.

4 M

Indexes: 0 red, 1 Green, 2 BLUE, 3 yellow, 4 Pink (negative: -1 Pink ... -5 red).

Output:

```
Green
Pink
('Green', 'BLUE', 'yellow')
('Pink', 'BLUE', 'red')
5 2
['red', 'Green']
```

- $T[1] \rightarrow$ 2nd item; $T[-1] \rightarrow$ last item.
- $T[1:4] \rightarrow$ items at index 1, 2, 3.
- $T[::-2] \rightarrow$ from last item backwards, every second item (4, 2, 0).
- $\text{len}() = 5$; $\text{index}(\text{"BLUE"}) = 2$; $\text{list}()$ converts the slice to a list.

(1 M for each 1½ lines of correct output)

c) Module — create, import, import object.

4 M

i) **Create module:** Write functions/variables in a file and save it with .py extension; the file name is the module name.

mathops.py

```
# mathops.py -- our module
PI = 3.14159

def square(n):
    return n * n
```

```
def cube(n):  
    return n * n * n
```

ii) Import module: import mathops (use as mathops.square()) or with alias import mathops as mo.

iii) Import object from module: from mathops import PI, cube — the names are used directly without the module name (from mathops import * imports all).

main program

```
import mathops                                # i) import whole module  
print("square(6) =", mathops.square(6))  
  
import mathops as mo                          # ii) import with new name (alias)  
print("cube(3)   =", mo.cube(3))  
  
from mathops import PI, cube                  # iii) import selected objects  
print("PI       =", PI)  
print("cube(4)  =", cube(4))
```

Output:

```
square(6) = 36  
cube(3)   = 27  
PI        = 3.14159  
cube(4)   = 64
```

(Each point 1 M, example 1 M)

d) Method overloading.

4 M

Method overloading means one method name that works with different numbers or types of arguments. Python does **not** support it in the usual way: if a method is defined twice, the last definition replaces the first.

```
class Demo:  
    def show(self, a):  
        print("one argument:", a)  
    def show(self, a, b): # replaces the earlier show()  
        print("two arguments:", a, b)  
  
d = Demo()  
d.show(1, 2)  
try:  
    d.show(1)  
except TypeError as e:  
    print("TypeError:", e)
```

Output:

```
two arguments: 1 2  
TypeError: Demo.show() missing 1 required positional argument: 'b'
```

So overloading is achieved using **default arguments** or **variable-length arguments (*args)**:

```
# Method overloading using *args (variable number of arguments)
class Calculator:
    def add(self, *nums):          # one method, many forms
        total = 0
        for n in nums:
            total += n
        return total

c = Calculator()
print("add(5, 10)          =", c.add(5, 10))
print("add(5, 10, 15)     =", c.add(5, 10, 15))
print("add(1.5, 2.5, 3, 4)=", c.add(1.5, 2.5, 3, 4))
```

Output:

```
add(5, 10)          = 15
add(5, 10, 15)     = 30
add(1.5, 2.5, 3, 4)= 11.0
```

(Explanation 2 M, example 2 M)

e) Stack using list.

4 M

A stack is a LIFO (Last In First Out) list. With a Python list, **append()** does PUSH and **pop()** does POP at the same end (TOP).

```
# Stack using Python list (LIFO)
stack = []
MAX = 4

def push(x):
    if len(stack) == MAX:
        print("Stack Overflow")
    else:
        stack.append(x)
        print(x, "pushed")

def pop():
    if not stack:
        print("Stack Underflow")
    else:
        print(stack.pop(), "popped")

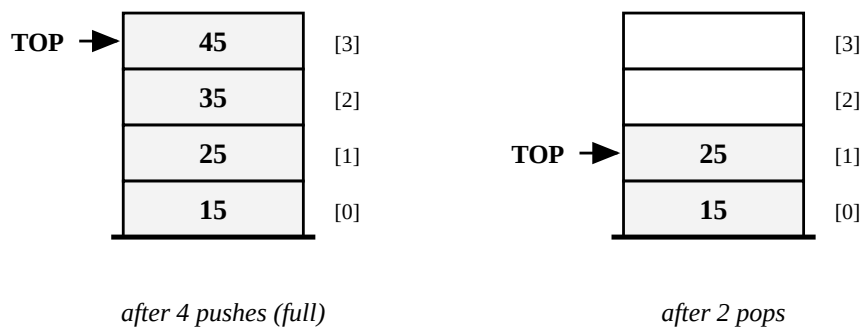
def display():
    print("Stack (top -> bottom):", stack[::-1])

for v in (15, 25, 35, 45, 55):
    push(v)
display()
pop(); pop()
display()
```

Output:

```
15 pushed
25 pushed
35 pushed
45 pushed
```

Stack Overflow
 Stack (top -> bottom): [45, 35, 25, 15]
 45 popped
 35 popped
 Stack (top -> bottom): [25, 15]



(PUSH 1 M, POP 1 M, display 1 M, output 1 M)

Q.5

a) Multiple inheritance.

6 M

In **multiple inheritance** a derived class inherits from **more than one** base class: `class Child(Father, Mother):`. The child object can use the methods of all base classes. If the same method exists in both parents, Python searches according to the **MRO** (Method Resolution Order), left to right.

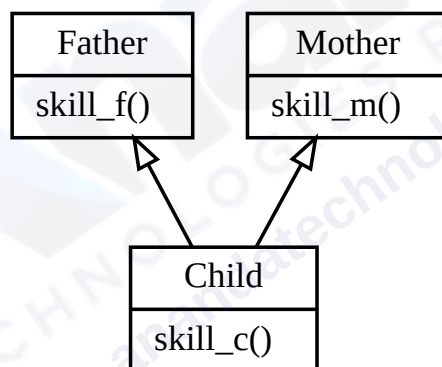


Fig.: Multiple inheritance

```
# Multiple inheritance: Child inherits from two parents
class Father:
    def skill_f(self):
        print("Father: Gardening")

class Mother:
    def skill_m(self):
        print("Mother: Painting")

class Child(Father, Mother):      # two base classes
    def skill_c(self):
        print("Child : Coding")

c = Child()
```

```
c.skill_f()
c.skill_m()
c.skill_c()
print("MRO:", [k.__name__ for k in Child.__mro__])
```

Output:

```
Father: Gardening
Mother: Painting
Child : Coding
MRO: ['Child', 'Father', 'Mother', 'object']
```

(Explanation 2 M, program 3 M, output 1 M)

b) Array and 2-D array program.

6 M

An **array** is a collection of elements of the **same data type** stored in contiguous memory locations and accessed by an index starting from 0. A **two-dimensional array** stores data in rows and columns; element $a[i][j]$ is in row i , column j .

0	1	2	3
11	12	13	14

Row 0 of the array (index 0 to 3)

```
import numpy as np
# two-dimensional array of integers (3 rows x 4 columns)
a = np.array([[11, 12, 13, 14],
              [21, 22, 23, 24],
              [31, 32, 33, 34]])
print("Array:\n", a)
print("Shape          :", a.shape)
print("a[1][2]        :", a[1][2])
print("a[2, 0]         :", a[2, 0])
print("Row 0           :", a[0])
print("Column 3        :", a[:, 3])
a[0, 0] = 99          # modify value
print("After a[0,0]=99:\n", a)
```

Output:

```
Array:
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
Shape          : (3, 4)
a[1][2]        : 23
a[2, 0]         : 31
Row 0           : [11 12 13 14]
Column 3        : [14 24 34]
After a[0,0]=99:
[[99 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

(Definition 2 M, program 3 M, output 1 M)

c) Infix to prefix.

6 M

i) $A + B * C - D / E$

Step 1 — reverse the expression (swap '(' and ')'): $E/D-C*B+A$

Step 2 — convert the reversed expression using stack (an operator pops only operators of *higher* precedence):

Symbol	Stack	Output
E	—	E
/	/	E
D	/	ED
-	-	ED/
C	-	ED/C
*	_*	ED/C
B	_*	ED/CB
+	_-+	ED/CB*
A	_-+	ED/CB*A
(end)	—	ED/CB*A+-

Step 3 — reverse the output: **Prefix = $-+A*BC/DE$**

ii) $(P - Q / R) * (S / T - U)$

Step 1 — reverse the expression (swap '(' and ')'): $(U-T/S)*(R/Q-P)$

Step 2 — convert the reversed expression using stack (an operator pops only operators of *higher* precedence):

Symbol	Stack	Output
(	(	
U	(	U
-	(-	U
T	(-	UT
/	(-/	UT
S	(-/	UTS
)	—	UTS/-
*	*	UTS/-
(	* (	UTS/-
R	* (	UTS/-R
/	* (/	UTS/-R
Q	* (/	UTS/-RQ

-	*(-	UTS/-RQ/
P	*(-	UTS/-RQ/P
)	*	UTS/-RQ/P-
(end)	—	UTS/-RQ/P-*

Step 3 — reverse the output: **Prefix = *-P/QR-/STU**

(Each expression 3 M)

Q.6

a) Basic terms.

6 M

- **Identifiers:** names given to variables, functions, classes etc. Rules: letters, digits, underscore; cannot start with a digit; case-sensitive; not a keyword. e.g. `total_marks`, `_count`.
- **Keywords:** reserved words with special meaning; cannot be used as identifiers. e.g. `if`, `else`, `for`, `while`, `def`, `class`, `True`, `None`.
- **Indentation:** leading spaces that define a block of code (Python does not use { }); usually 4 spaces. e.g.

```
if x > 0:
    print("positive")
```

- **Variables:** named memory locations that store values; created on assignment, no type declaration. e.g. `age = 19`.
- **Comments:** non-executable notes. Single-line with `#`, multi-line using triple quotes. e.g. `# find sum`.
- **Tuples:** ordered, immutable collection in `()`. e.g. `t = (10, 20, 30)`.

(Each term with example 1 M)

b) Singly linked list program.

6 M

A singly linked list is a chain of nodes; each node has **data** and **next** (reference to next node). **head** refers to the first node; next of the last node is `None`.

```
# Singly linked list: create, append and display
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def append(self, data):          # insert at end
        new = Node(data)
        if self.head is None:
            self.head = new
            return
        temp = self.head
```

```

while temp.next:
    temp = temp.next
temp.next = new

def display(self):
    temp = self.head
    while temp:
        print(temp.data, end=" -> ")
        temp = temp.next
    print("None")

ll = LinkedList()
for v in (18, 27, 36, 45):
    ll.append(v)
ll.display()
ll.append(54)
print("After append(54):")
ll.display()

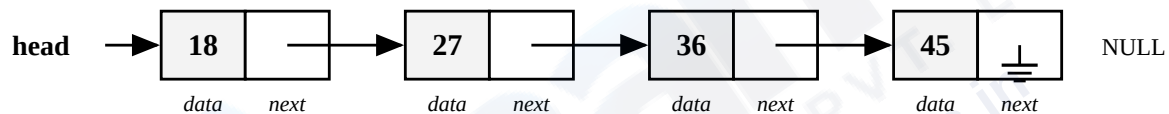
```

Output:

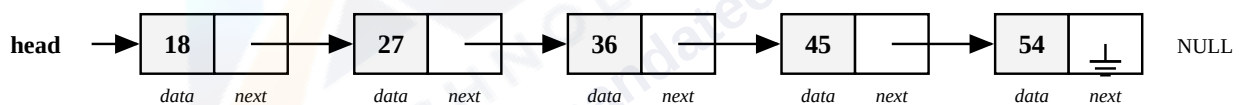
```

18 -> 27 -> 36 -> 45 -> None
After append(54):
18 -> 27 -> 36 -> 45 -> 54 -> None

```



List after appending 18, 27, 36, 45



After append(54): new node linked at the end

(Node class 1 M, append 2 M, display 1 M, output and diagram 2 M)

c) BFS and DFS.

6 M

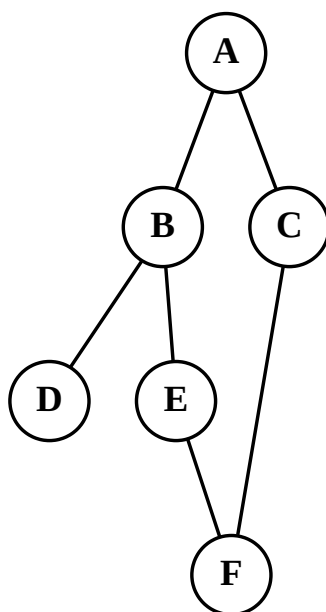


Fig. 1

BFS (Breadth First Search) visits the start vertex, then all its adjacent vertices, then their unvisited neighbours — level by level. It uses a **queue**.

1. Insert start vertex in queue, mark visited.
2. Repeat while queue is not empty: delete vertex V from queue and visit it.
3. Insert all unvisited neighbours of V in the queue and mark them visited.

DFS (Depth First Search) goes as deep as possible from the start vertex along one path, then backtracks. It uses a **stack** (or recursion).

1. Visit start vertex, mark visited.
2. For each unvisited neighbour W of V, call DFS(W).
3. When no unvisited neighbour remains, backtrack.

```

# BFS and DFS on a graph (adjacency list)
from collections import deque
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E'],
}

def bfs(start):
    visited, q, order = {start}, deque([start]), []
    while q:
        v = q.popleft()
        order.append(v)
        for w in graph[v]:
            if w not in visited:
                visited.add(w)
                q.append(w)
  
```

```
return order

def dfs(v, visited=None):
    if visited is None:
        visited = []
    visited.append(v)
    for w in graph[v]:
        if w not in visited:
            dfs(w, visited)
    return visited

print("BFS from A:", bfs('A'))
print("DFS from A:", dfs('A'))
```

Output:

```
BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']
DFS from A: ['A', 'B', 'D', 'E', 'F', 'C']
```

BFS from A: A, B, C, D, E, F **DFS from A:** A, B, D, E, F, C (neighbours in alphabetical order).

(BFS 3 M, DFS 3 M)

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in