**AI/ML**

**5 BCA B**

**"Practical - 2"**

***BY***

**"Tharan" (23215134)**

**SUBMITTED TO**

**MEGHAVI**

****

**SCHOOL OF SCIENCES**

**2025-2026**

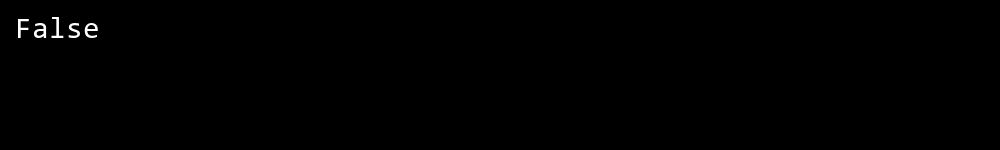
## **QUESTION 1**

1. Traverse the tree using BFS first forming a tree then traversing it

### **Code Solution**

class Node:  
 def \_\_init\_\_(self, value):  
 self.left = None  
 self.right = None  
 self.value = value  
  
def search\_element(root, target):  
 if root is None or root.value == target:  
 return root  
 if target < root.value:  
 return search\_element(root.left, target)  
 return search\_element(root.right, target)  
  
root = Node(8)  
root.left = Node(3)  
root.right = Node(10)  
root.left.left = Node(1)  
root.left.right = Node(6)  
root.right.right = Node(14)  
root.right.right.left = Node(13)  
  
target = 4  
result = search\_element(root, target)  
  
if result:  
 print(True)  
else:  
 print(False)

### **FINAL Output**



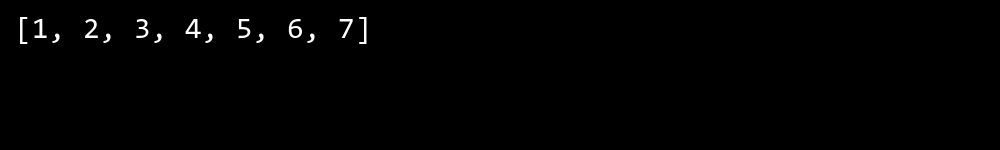
## **QUESTION 2**

2. Showcase the process of searching element in a graph.  
taking multiple number for forming a tree

### **Code Solution**

class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
def breadth\_first\_search(root):  
 if root is None:  
 return []  
   
 result = []  
 queue = []  
 queue.append(root)  
   
 while queue:  
 current = queue.pop(0)  
 result.append(current.value)  
   
 if current.left:  
 queue.append(current.left)  
 if current.right:  
 queue.append(current.right)  
   
 return result  
  
root = Node(1)  
root.left = Node(2)  
root.right = Node(3)  
root.left.left = Node(4)  
root.left.right = Node(5)  
root.right.left = Node(6)  
root.right.right = Node(7)  
  
traversal\_result = breadth\_first\_search(root)  
print(traversal\_result)

### **FINAL Output**



## **QUESTION 3**

3. Take any number as the input from the user for searching an element from a tree.

### **Code Solution**

class Graph:  
 def \_\_init\_\_(self):  
 self.graph = {}  
   
 def add\_edge(self, vertex, edge):  
 if vertex not in self.graph:  
 self.graph[vertex] = []  
 self.graph[vertex].append(edge)  
   
 def bfs(self, start, search\_value):  
 visited = set()  
 queue = [start]  
 visited.add(start)  
   
 while queue:  
 vertex = queue.pop(0)  
 if vertex == search\_value:  
 return True  
   
 for neighbor in self.graph[vertex]:  
 if neighbor not in visited:  
 visited.add(neighbor)  
 queue.append(neighbor)  
 return False  
   
 def dfs(self, start, search\_value, visited=None):  
 if visited is None:  
 visited = set()  
   
 if start == search\_value:  
 return True  
   
 visited.add(start)  
   
 for neighbor in self.graph[start]:  
 if neighbor not in visited:  
 if self.dfs(neighbor, search\_value, visited):  
 return True  
 return False  
  
g = Graph()  
g.add\_edge(1, 2)  
g.add\_edge(1, 3)  
g.add\_edge(2, 4)  
g.add\_edge(2, 5)  
g.add\_edge(3, 6)  
g.add\_edge(3, 7)  
g.add\_edge(4, 8)  
g.add\_edge(5, 9)  
g.add\_edge(6, 10)  
  
search\_value = 7  
print(g.bfs(1, search\_value))  
print(g.dfs(1, search\_value))

### **FINAL Output**

