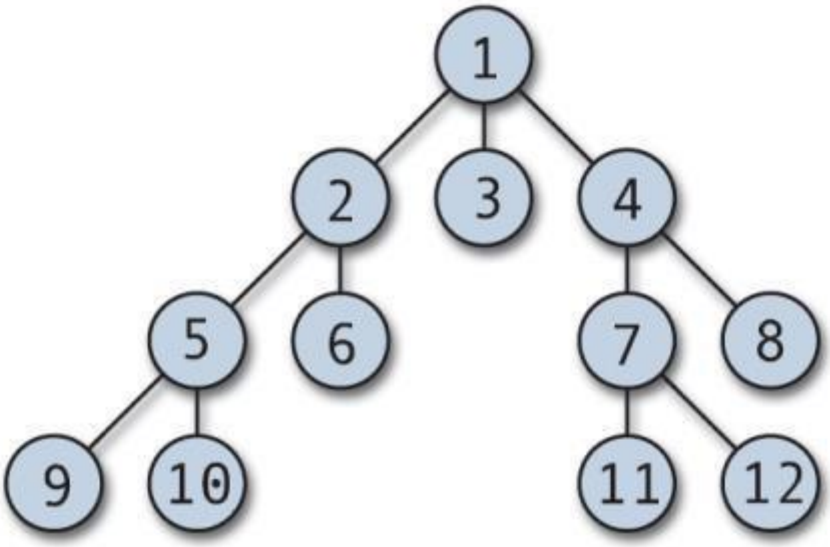
190031187 NERELLA VENKATA RADHAKRISHNA

Artificial Intelligence

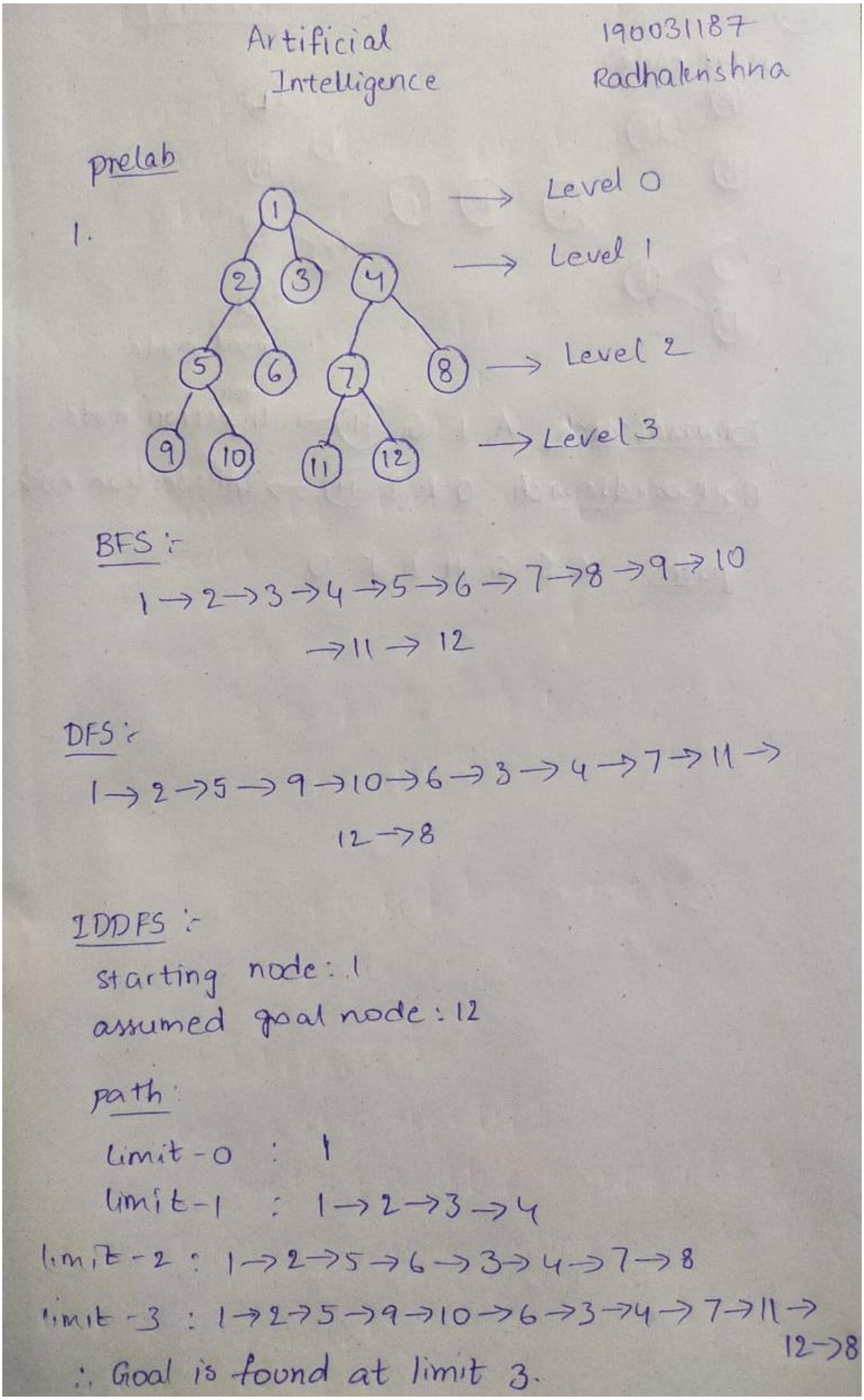
LAB-3

PRELAB:

1. Trace out the path using BFS, DFS and IDDFS for the following tree.



***implement iteration path for each and every level.



Artificial Intelligence

190031187
Radhakrishna

prelab

1.

1
2
3
4
5
6
7
8
9
10
11
12
→ Level 0
→ Level 1
→ Level 2
→ Level 3

BFS :-

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12

DFS :-

1 → 2 → 5 → 9 → 10 → 6 → 3 → 4 → 7 → 11 → 12 → 8

IDDFS :-

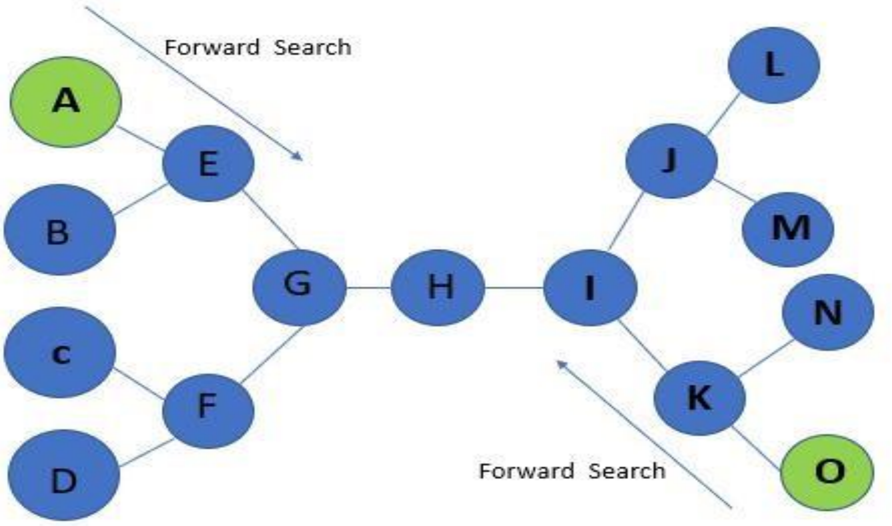
starting node : 1
assumed goal node : 12

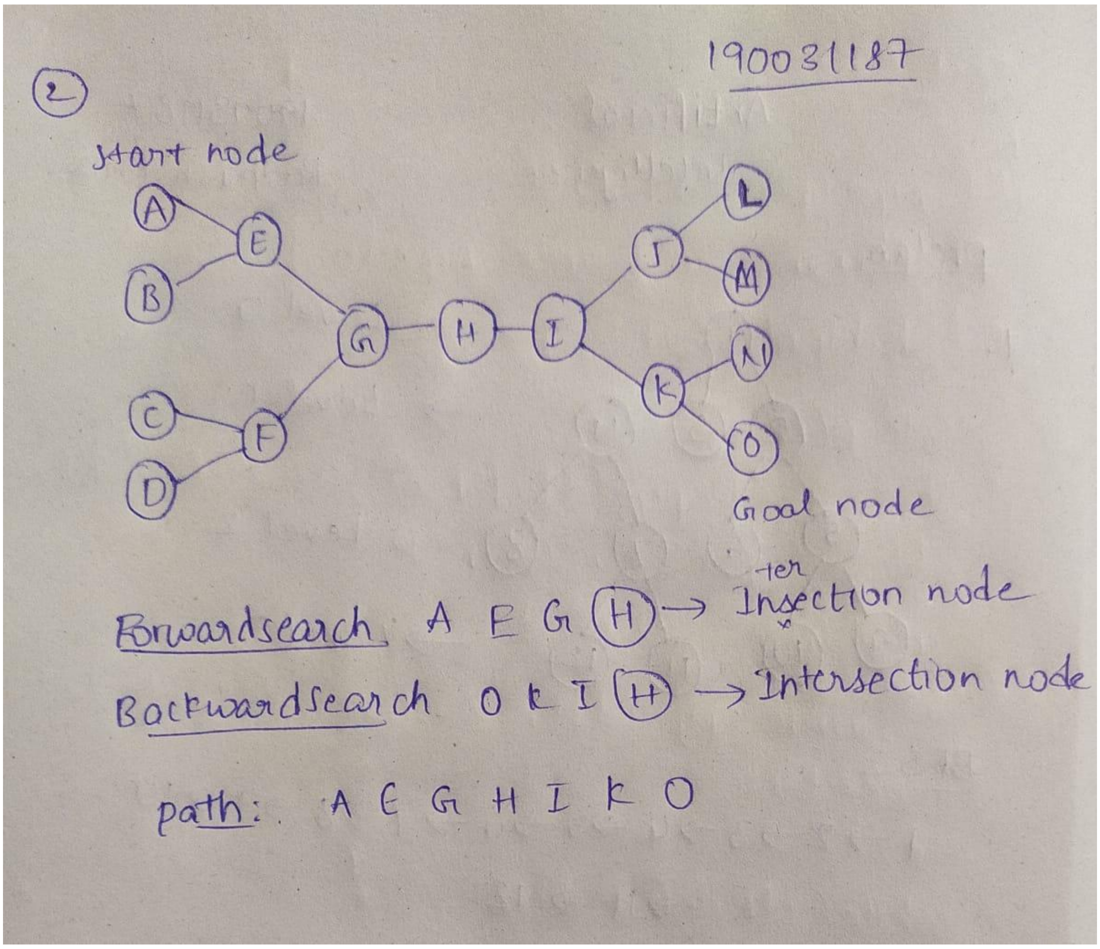
path :

limit-0 : 1
limit-1 : 1 → 2 → 3 → 4
limit-2 : 1 → 2 → 5 → 6 → 3 → 4 → 7 → 8
limit-3 : 1 → 2 → 5 → 9 → 10 → 6 → 3 → 4 → 7 → 11 → 12 → 8

∴ Goal is found at limit 3.

2. Trace out the path using Bidirectional search for the following tree with the given starting node and the goal node.




A
E
B
G
H
I
C
F
D
J
L
M
N
K
O

190031187

2

Start node

Goal node

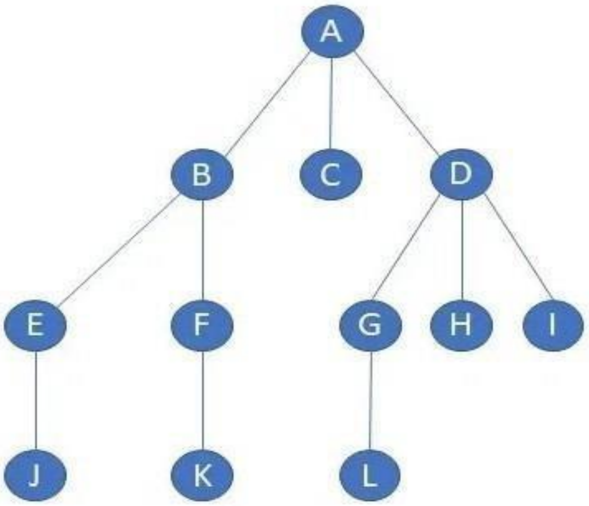
Forwardsearch A E G H → Intersection node

Backwardsearch O K I H → Intersection node

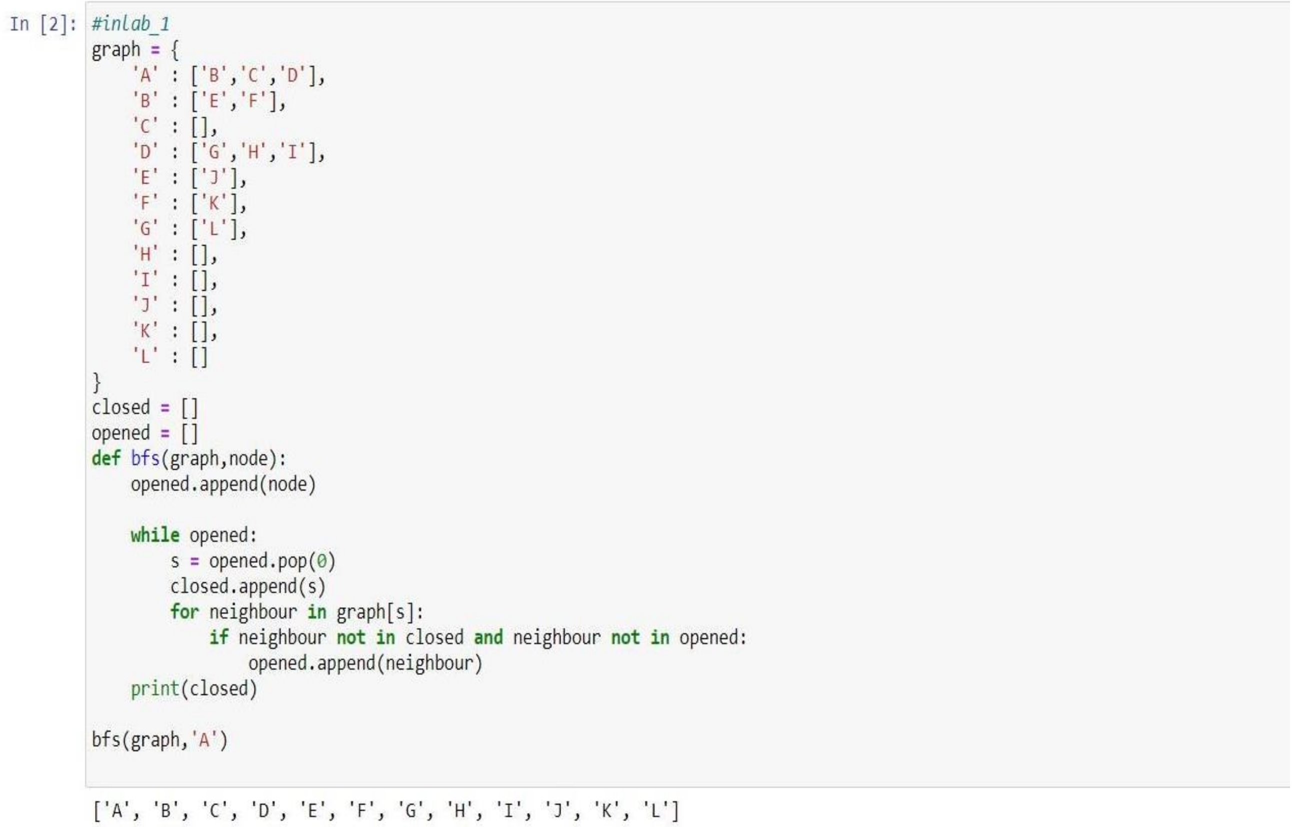
path: A E G H I K O

INLAB:

1. Write an efficient python program to implement Breadth-First Search by considering the following tree.

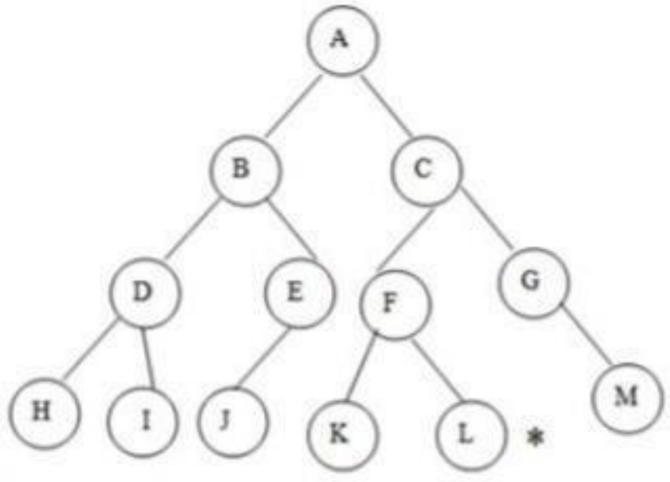


Output:
A B C D E F G H I

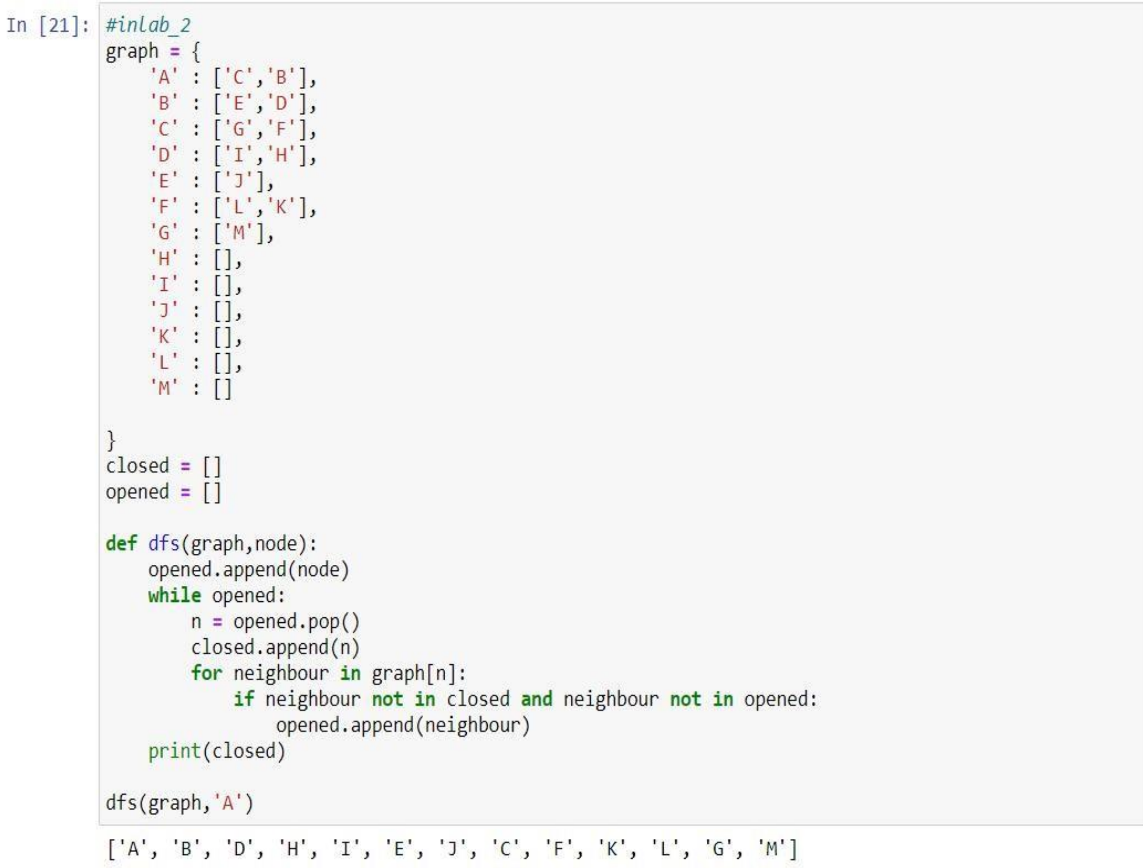


```
In [2]: #inlab_1
        graph = {
            'A' : ['B','C','D'],
            'B' : ['E','F'],
            'C' : [],
            'D' : ['G','H','I'],
            'E' : ['J'],
            'F' : ['K'],
            'G' : ['L'],
            'H' : [],
            'I' : [],
            'J' : [],
            'K' : [],
            'L' : []
        }
        closed = []
        opened = []
        def bfs(graph,node):
            opened.append(node)

            while opened:
                s = opened.pop(0)
                closed.append(s)
                for neighbour in graph[s]:
                    if neighbour not in closed and neighbour not in opened:
                        opened.append(neighbour)
            print(closed)

        bfs(graph,'A')

        ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L']
```

2. Write a python code to implement Depth-First Search by considering the following tree. Your code must satisfy the tree to find the best and shortest path.



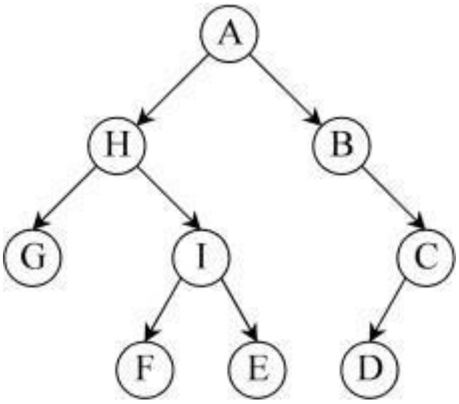
Output:
A B D H I J C F K L G M



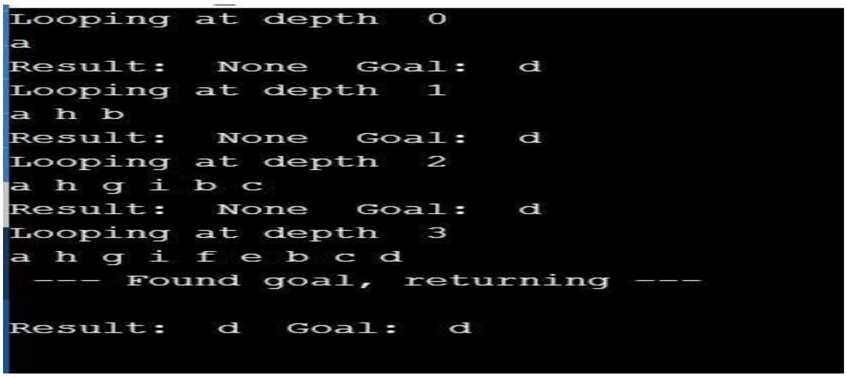
```
In [21]: #inlab_2
         graph = {
             'A' : ['C','B'],
             'B' : ['E','D'],
             'C' : ['G','F'],
             'D' : ['I','H'],
             'E' : ['J'],
             'F' : ['L','K'],
             'G' : ['M'],
             'H' : [],
             'I' : [],
             'J' : [],
             'K' : [],
             'L' : [],
             'M' : []

         }
         closed = []
         opened = []

         def dfs(graph,node):
             opened.append(node)
             while opened:
                 n = opened.pop()
                 closed.append(n)
                 for neighbour in graph[n]:
                     if neighbour not in closed and neighbour not in opened:
                         opened.append(neighbour)
             print(closed)

         dfs(graph,'A')

         ['A', 'B', 'D', 'H', 'I', 'E', 'J', 'C', 'F', 'K', 'L', 'G', 'M']
```

POSTLAB:

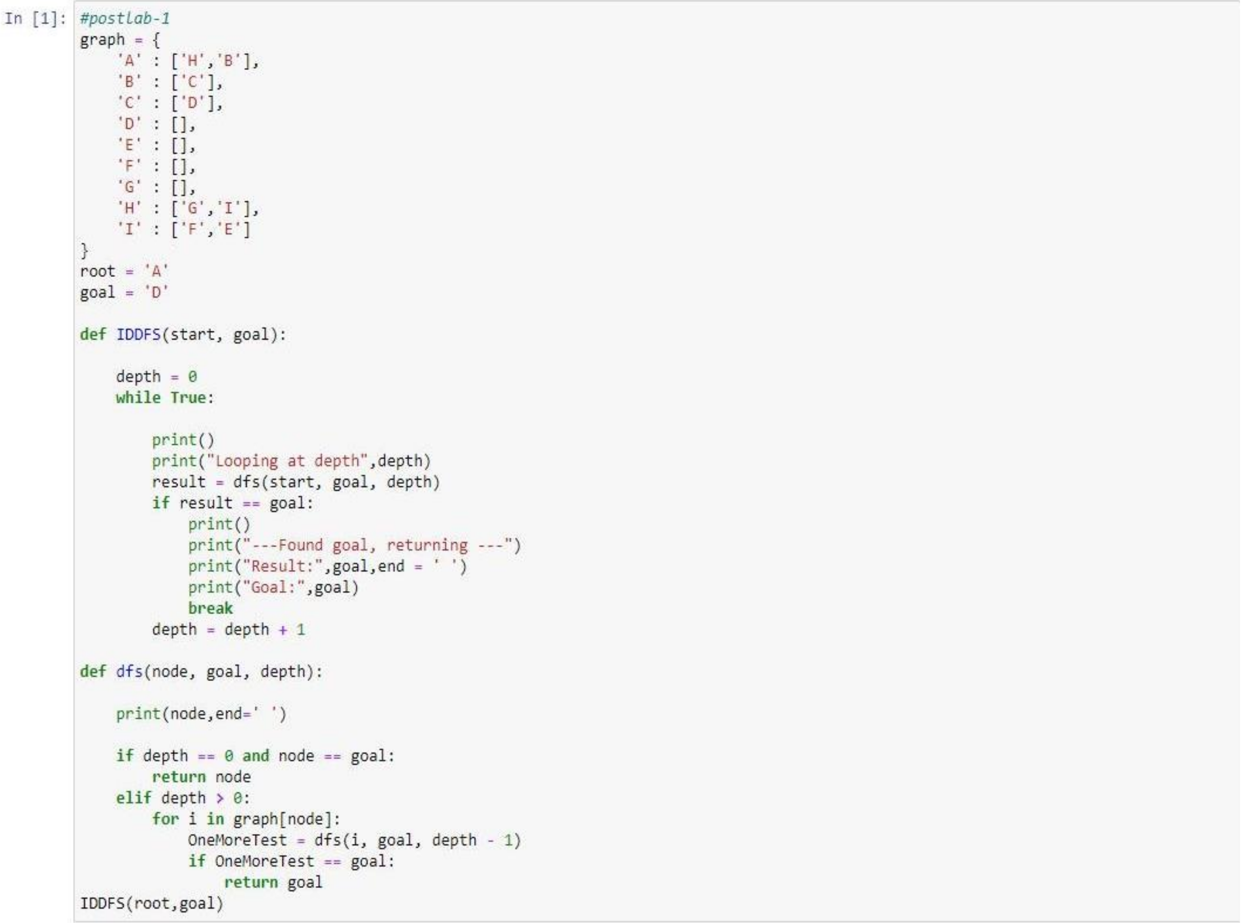
1. Write an efficient python program to implement Iterative deepening search by considering following tree.



Output:

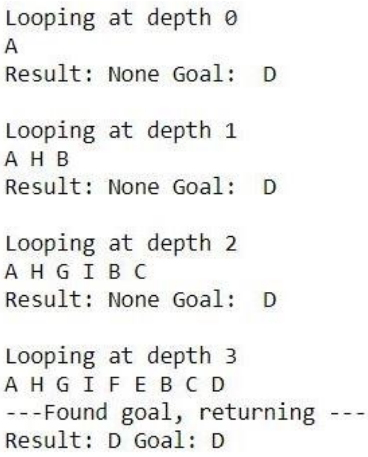


```
Looping at depth  0
a
Result:  None  Goal:  d
Looping at depth  1
a h b
Result:  None  Goal:  d
Looping at depth  2
a h g i b c
Result:  None  Goal:  d
Looping at depth  3
a h g i f e b c d
 --- Found goal, returning ---

Result:  d  Goal:  d
```



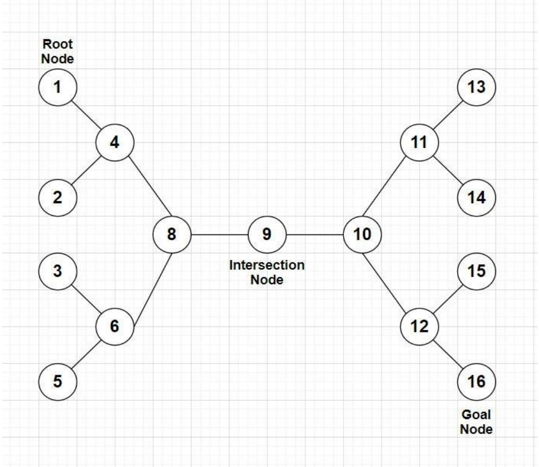
```
In [1]: #postlab-1
graph = {
    'A' : ['H','B'],
    'B' : ['C'],
    'C' : ['D'],
    'D' : [],
    'E' : [],
    'F' : [],
    'G' : [],
    'H' : ['G','I'],
    'I' : ['F','E']
}
root = 'A'
goal = 'D'

def IDDFS(start, goal):

    depth = 0
    while True:

        print()
        print("Looping at depth",depth)
        result = dfs(start, goal, depth)
        if result == goal:
            print()
            print("---Found goal, returning ---")
            print("Result:",goal,end = ' ')
            print("Goal:",goal)
            break
        depth = depth + 1

def dfs(node, goal, depth):

    print(node,end=' ')

    if depth == 0 and node == goal:
        return node
    elif depth > 0:
        for i in graph[node]:
            OneMoreTest = dfs(i, goal, depth - 1)
            if OneMoreTest == goal:
                return goal
IDDFS(root,goal)
```

OUTPUT

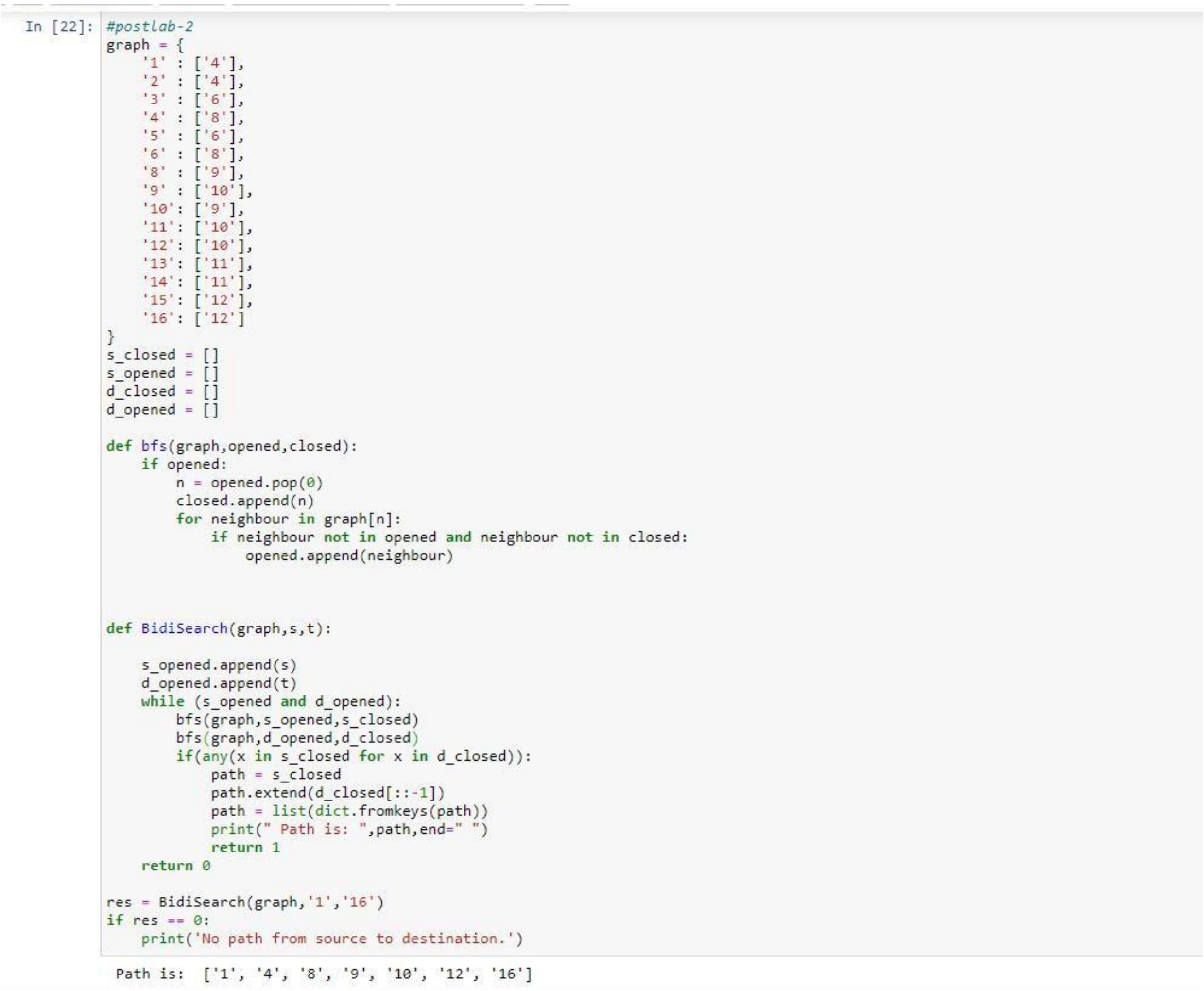


```
Looping at depth 0
A
Result: None Goal:  D

Looping at depth 1
A H B
Result: None Goal:  D

Looping at depth 2
A H G I B C
Result: None Goal:  D

Looping at depth 3
A H G I F E B C D
---Found goal, returning ---
Result: D Goal: D
```

2.Write an efficient python program to implement Bidirectional search by considering following tree and print the path.



Output:
[1, 4, 8, 9, 10, 12, 16]



```
In [22]: #postlab-2
graph = {
    '1' : ['4'],
    '2' : ['4'],
    '3' : ['6'],
    '4' : ['8'],
    '5' : ['6'],
    '6' : ['8'],
    '8' : ['9'],
    '9' : ['10'],
    '10': ['9'],
    '11': ['10'],
    '12': ['10'],
    '13': ['11'],
    '14': ['11'],
    '15': ['12'],
    '16': ['12']
}
s_closed = []
s_opened = []
d_closed = []
d_opened = []

def bfs(graph,opened,closed):
    if opened:
        n = opened.pop(0)
        closed.append(n)
        for neighbour in graph[n]:
            if neighbour not in opened and neighbour not in closed:
                opened.append(neighbour)



def BidiSearch(graph,s,t):

    s_opened.append(s)
    d_opened.append(t)
    while (s_opened and d_opened):
        bfs(graph,s_opened,s_closed)
        bfs(graph,d_opened,d_closed)
        if(any(x in s_closed for x in d_closed)):
            path = s_closed
            path.extend(d_closed[::-1])
            path = list(dict.fromkeys(path))
            print(" Path is: ",path,end=" ")
            return 1
    return 0

res = BidiSearch(graph,'1','16')
if res == 0:
    print('No path from source to destination.')
```

```
 Path is:  ['1', '4', '8', '9', '10', '12', '16']
```