**Output:**  
Following is BFS Traversal:

deque(['A'])

A deque(['B', 'C'])

B deque(['C', 'D', 'E'])

C deque(['D', 'E', 'F', 'G'])

D deque(['E', 'F', 'G', 'H', 'I'])

E deque(['F', 'G', 'H', 'I', 'J', 'K'])

F deque(['G', 'H', 'I', 'J', 'K', 'L', 'M'])

G deque(['H', 'I', 'J', 'K', 'L', 'M', 'N', 'O'])

H deque(['I', 'J', 'K', 'L', 'M', 'N', 'O'])

I deque(['J', 'K', 'L', 'M', 'N', 'O'])

J deque(['K', 'L', 'M', 'N', 'O'])

K deque(['L', 'M', 'N', 'O'])

L deque(['M', 'N', 'O'])

M deque(['N', 'O'])

N deque(['O'])

O Following is DFS Traversal:

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

**Comparison:**  
BFS uses a queue to store all the nodes at the current depth before moving to the next. This means its memory usage grows with the number of nodes at the deepest level of the graph, which can be significant for wide graphs.

DFS, on the other hand, relies on recursion (or a stack) and explores one branch fully before backtracking. Its memory usage depends on the depth of the graph, which can be more efficient for graphs with fewer levels.

**Complexities:**

 TimeComplexity: Nodes are processed all at once.

 SpaceComplexity: as the queue may contain all nodes at a given depth. For wide graphs, the queue can grow significantly.  
  
Conclusion:

BFS is more memory-intensive when the graph is wide, as the queue grows significantly to hold all child nodes.

DFS, by relying on the recursion stack, uses less memory for shallow graphs but can grow large for deep graphs.