# BFS Code

**Task 1**

## Original code

def bfs(graph, start):  
 visited = set()   
 queue = [start]   
  
 while queue:   
 vertex = queue.pop(0)   
  
 if vertex not in visited:  
 print(vertex,end = "," )  
 visited.add(vertex)  
  
 for neighbor in graph[vertex]:  
 if neighbor not in visited:  
 queue.append(neighbor)  
  
  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
print("BFS Traversal:")  
bfs(graph, 'A')

## Line-by-line explanation

def bfs(graph, start):  
 Define a function named `bfs` that takes two arguments: `graph` (an adjacency-list representation) and `start` (the node to begin traversal from).

visited = set()  
 Create an empty set named `visited`. This will store nodes that have already been processed so they are not printed or expanded again.

queue = [start]  
 Initialize `queue` as a Python list containing the `start` node. This list is used with FIFO behavior (first-in, first-out).

while queue:  
 Loop as long as `queue` is not empty. Each loop processes one node taken from the front of the queue.

vertex = queue.pop(0)  
 Remove and return the first element of `queue`. `pop(0)` both returns the element and deletes it from the list. Note: this is O(n) on lists because elements must shift left.

if vertex not in visited:  
 Only process this `vertex` if it hasn't been visited yet. This prevents printing/expanding a node more than once.

print(vertex, end = ",")  
 Print the current `vertex` followed by a comma with no newline. This produces output like `A,B,C,` (there will be a trailing comma).

visited.add(vertex)  
 Add `vertex` to the `visited` set so future occurrences are skipped.

for neighbor in graph[vertex]:  
 Iterate over every neighbor (adjacent node) of `vertex` in the adjacency list.

if neighbor not in visited:  
 Before enqueuing a neighbor, check it's not already in `visited` (but note: this does NOT check whether the neighbor is already in the queue).

queue.append(neighbor)  
 Add the neighbor to the back of the queue so it will be processed in FIFO order later.

## Task 2

def bfs(graph, start\_node):  
 visited\_nodes = set()   
 queue = [start\_node]   
 while len(queue) > 0:   
 current\_node = queue[0]   
 queue = queue[1:]   
  
 if current\_node not in visited\_nodes:  
 print(current\_node, end=",")  
 visited\_nodes.add(current\_node)  
  
 for neighbor in graph[current\_node]:  
 if neighbor not in visited\_nodes:  
 queue.append(neighbor)  
  
  
graph = {  
 'A': ['B', 'C'],  
 'B': ['D', 'E'],  
 'C': ['F'],  
 'D': [],  
 'E': ['F'],  
 'F': []  
}  
  
print("BFS Result:")  
bfs(graph, 'A')

# Line By Line Explanantion:

* The function bfs(graph, start\_node) is defined with two parameters:
* Graph represents the adjacency list of the graph.
* start\_node is the node where traversal begins.
* A set named visited\_nodes is created to store all the nodes that have already been visited. This ensures that no node is visited more than once.
* A list named queue is initialized with the starting node. This queue will hold nodes waiting to be explored.
* A while loop runs as long as the queue is not empty. This guarantees that all reachable nodes will eventually be processed.
* The variable current\_node takes the first element of the queue, representing the next node to be processed.
* The queue is updated to remove the first element, leaving the remaining nodes to be processed later. This simulates the process of dequeuing.
* A check is performed to see if current\_node has already been visited. If it has not, the node will be processed.
* The current node is printed on the screen, followed by a comma, to display the order of traversal.
* The node is then added to the set of visited nodes to mark it as processed.
* The program loops through all the neighbors (connected nodes) of the current node from the graph.
* For each neighbor, a check is done to see if it has already been visited.
* If the neighbor has not been visited, it is added to the end of the queue so that it will be processed later in the correct BFS order.
* The graph is defined as a dictionary where each node points to a list of its neighbors.
* A print statement displays the heading "BFS Result:" before showing the actual traversal.
* Finally, the bfs function is called with the graph and starting node A, beginning the breadth-first search traversal.