# Question

Given a reference of a node in a [**connected**](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)#Connected_graph) undirected graph.

Return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) (clone) of the graph.

Each node in the graph contains a val (int) and a list (List[Node]) of its neighbors.

class Node {

public int val;

public List<Node> neighbors;

}

**Test case format:**

For simplicity sake, each node's value is the same as the node's index (1-indexed). For example, the first node with val = 1, the second node with val = 2, and so on. The graph is represented in the test case using an adjacency list.

**Adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

**Example 1:**



**Input:** adjList = [[2,4],[1,3],[2,4],[1,3]]

**Output:** [[2,4],[1,3],[2,4],[1,3]]

**Explanation:** There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Example 2:**



**Input:** adjList = [[]]

**Output:** [[]]

**Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

**Example 3:**

**Input:** adjList = []

**Output:** []

**Explanation:** This an empty graph, it does not have any nodes.

**Example 4:**



**Input:** adjList = [[2],[1]]

**Output:** [[2],[1]]

**Constraints:**

* 1 <= Node.val <= 100
* Node.val is unique for each node.
* Number of Nodes will not exceed 100.
* There is no repeated edges and no self-loops in the graph.
* The Graph is connected and all nodes can be visited starting from the given node.

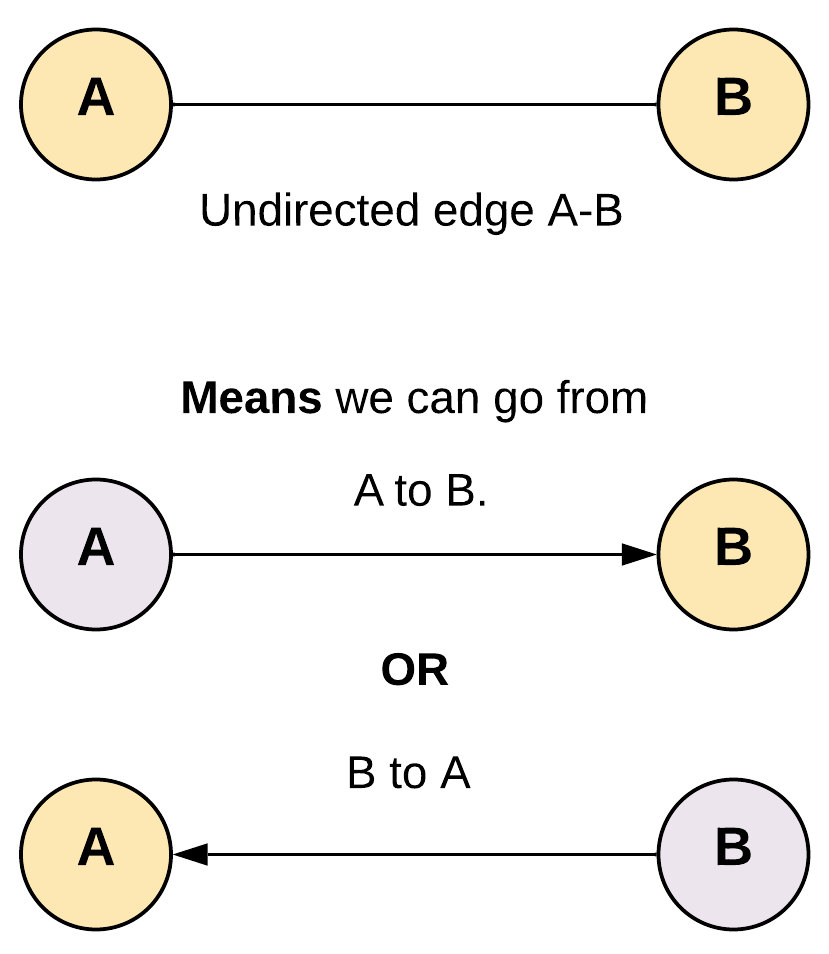
# Solution

#### **Approach 1: Depth First Search**

**Intuition**

Note: As we can see this question has garnered a lot of negative reviews. It has a lot more dislikes than the likes. We have tried to improve the problem statement to make it more understandable. However, these are the kinds of situations you might get into in an interview when the problem statement might look a little absurd. What is important then is to ask the interviewer to clarify the problem. This problem statement was confusing to me as well initially and that's why I decided to write the solution hoping to clarify most of the doubts that the readers might have had.

The basic intuition for this problem is to just copy as we go. We need to understand that we are dealing with a graph and this means a node could have any number of neighbors. This is why neighbors is a list. What is also crucial to understand is that we don't want to get stuck in a cycle while we are traversing the graph. According to the problem statement, any given undirected edge could be represented as two directional edges. So, if there is an undirected edge between node A and node B, the graph representation for it would have a directed edge from A to B and another from B to A. After all, an undirected graph is a set of nodes that are connected together, where all the edges are bidirectional. How else would you say that A could be reached from B and B could be reached from A?

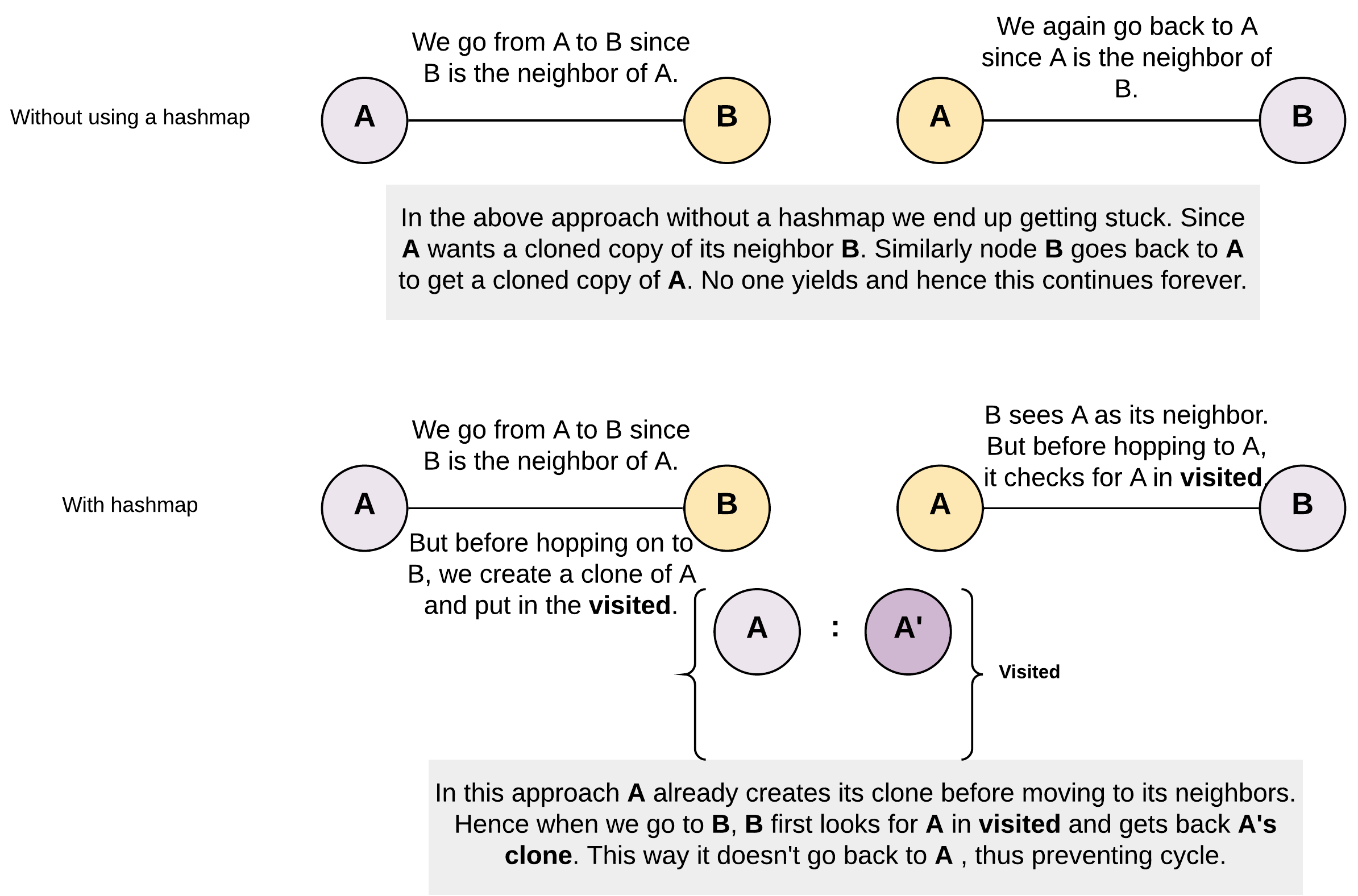


To avoid getting stuck in a loop we would need some way to keep track of the nodes which have already been copied. By doing this we don't end up traversing them again.

**Algorithm**

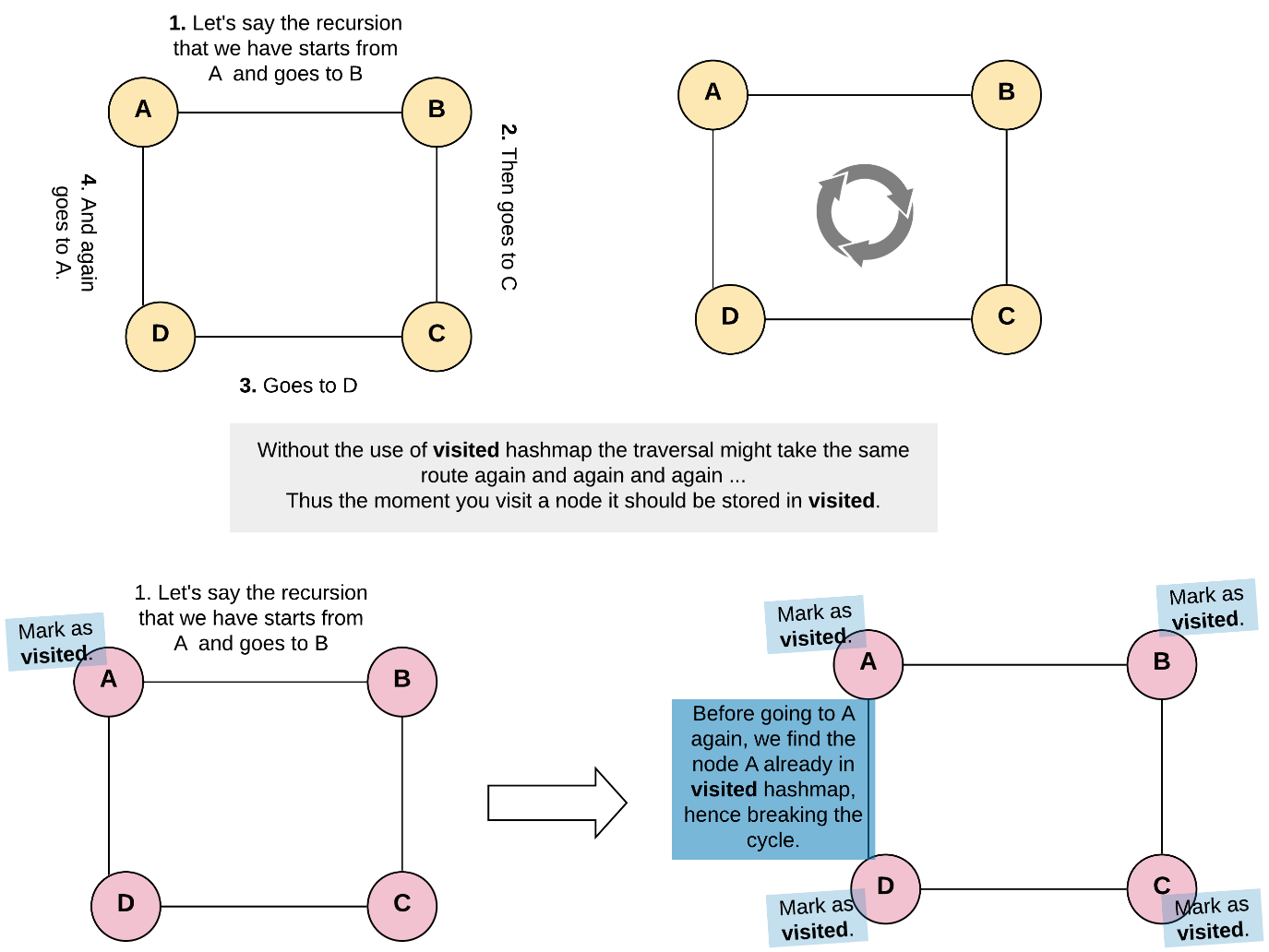
1. Start traversing the graph from the given node.
2. We would take a hash map to store the reference of the copy of all the nodes that have already been visited and cloned. The key for the hash map would be the node of the original graph and corresponding value would be the corresponding cloned node of the cloned graph. If the node already exists in the visited we return corresponding stored reference of the cloned node.

For a given edge A - B, since A is connected to B and B is also connected to A if we don't use visited we will get stuck in a cycle.



1. If we don't find the node in the visited hash map, we create a copy of it and put it in the hash map. Note, how it's important to create a copy of the node and add to the hash map before entering recursion.
2. clone\_node = Node(node.val, [])
3. visited[node] = clone\_node

In the absence of such an ordering, we would be caught in the recursion because on encountering the node again in somewhere down the recursion again, we will be traversing it again thus getting into cycles.



1. Now make the recursive call for the neighbors of the node. Pay attention to how many recursion calls we will be making for any given node. For a given node the number of recursive calls would be equal to the number of its neighbors. Each recursive call made would return the clone of a neighbor. We will prepare the list of these clones returned and put into neighbors of clone node which we had created earlier. This way we will have cloned the given node and it's neighbors.

Tip: Recursion could get a bit cumbersome to grasp, if you try to get into every call yourself and try to see what's happening. And why look at every call when every call does the same thing with different inputs. So, you just worry about ONE such call and let the recursion do the rest. And of course always handle the base case or the termination condition of the recursion. Otherwise how would it end?

|  |
| --- |
| **/\***  **// Definition for a Node.**  **class Node {**  **public int val;**  **public List<Node> neighbors;**  **public Node() {}**  **public Node(int \_val,List<Node> \_neighbors) {**  **val = \_val;**  **neighbors = \_neighbors;**  **}**  **};**  **\*/**  **class Solution {**  **private HashMap <Node, Node> visited = new HashMap <> ();**  **public Node cloneGraph(Node node) {**  **if (node == null) {**  **return node;**  **}**  **// If the node was already visited before.**  **// Return the clone from the visited dictionary.**  **if (visited.containsKey(node)) {**  **return visited.get(node);**  **}**  **// Create a clone for the given node.**  **// Note that we don't have cloned neighbors as of now, hence [].**  **Node cloneNode = new Node(node.val, new ArrayList());**  **// The key is original node and value being the clone node.**  **visited.put(node, cloneNode);**  **// Iterate through the neighbors to generate their clones**  **// and prepare a list of cloned neighbors to be added to the cloned node.**  **for (Node neighbor: node.neighbors) {**  **cloneNode.neighbors.add(cloneGraph(neighbor));**  **}**  **return cloneNode;**  **}**  **}** |

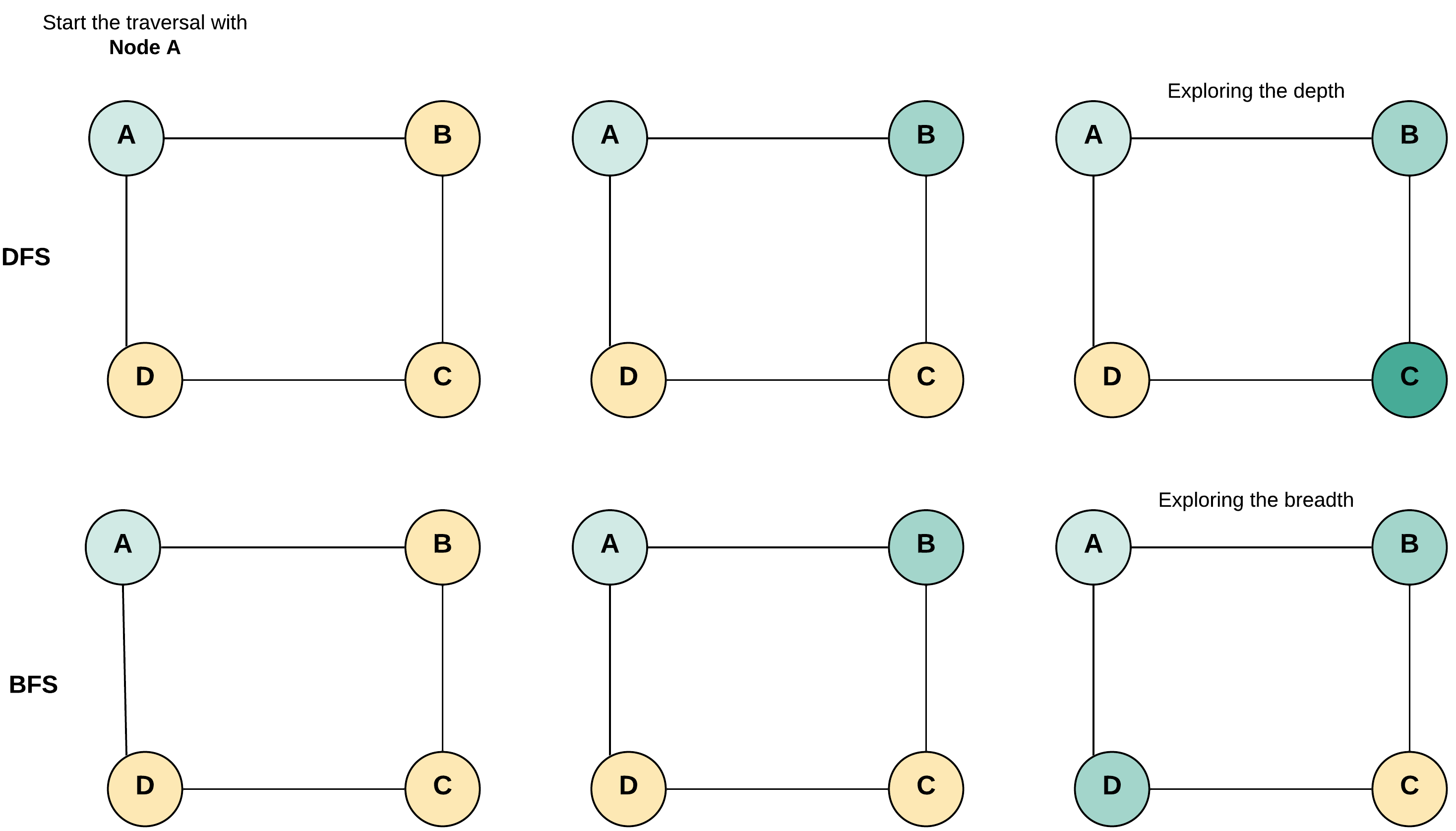
**Complexity Analysis**

* Time Complexity: O(N + M)*O*(*N*+*M*), where N*N* is a number of nodes (vertices) and M*M* is a number of edges.
* Space Complexity: O(N)*O*(*N*). This space is occupied by the visited hash map and in addition to that, space would also be occupied by the recursion stack since we are adopting a recursive approach here. The space occupied by the recursion stack would be equal to O(H)*O*(*H*) where H*H* is the height of the graph. Overall, the space complexity would be O(N)*O*(*N*).

#### **Approach 2: Breadth First Search**

**Intuition**

We could agree DFS is a good enough solution for this problem. However, if the recursion stack is what we are worried about then DFS is not our best bet. Sure, we can write an iterative version of depth first search by using our own stack. However, we also have the BFS way of doing iterative traversal of the graph and we'll be exploring that solution as well.



The difference is only in the traversal of DFS and BFS. As the name says it all, DFS explores the depths of the graph first and BFS explores the breadth. Based on the kind of graph we are expecting we can chose one over the other. We would need the visited hash map in both the approaches to avoid cycles.

**Algorithm**

1. We will use a hash map to store the reference of the copy of all the nodes that have already been visited and copied. The key for the hash map would be the node of the original graph and corresponding value would be the corresponding cloned node of the cloned graph. The visited is used to prevent cycles and get the cloned copy of a node.
2. Add the first node to the queue. Clone the first node and add it to visited hash map.
3. Do the BFS traversal.
   * Pop a node from the front of the queue.
   * Visit all the neighbors of this node.
   * If any of the neighbors was already visited then it must be present in the visited dictionary. Get the clone of this neighbor from visited in that case.
   * Otherwise, create a clone and store in the visited.
   * Add the clones of the neighbors to the corresponding list of the clone node.

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public List<Node> neighbors;  public Node() {}  public Node(int \_val,List<Node> \_neighbors) {  val = \_val;  neighbors = \_neighbors;  }  };  \*/  class Solution {  public Node cloneGraph(Node node) {  if (node == null) {  return node;  }  // Hash map to save the visited node and it's respective clone  // as key and value respectively. This helps to avoid cycles.  HashMap<Node, Node> visited = new HashMap();  // Put the first node in the queue  LinkedList<Node> queue = new LinkedList<Node> ();  queue.add(node);  // Clone the node and put it in the visited dictionary.  visited.put(node, new Node(node.val, new ArrayList()));  // Start BFS traversal  while (!queue.isEmpty()) {  // Pop a node say "n" from the from the front of the queue.  Node n = queue.remove();  // Iterate through all the neighbors of the node "n"  for (Node neighbor: n.neighbors) {  if (!visited.containsKey(neighbor)) {  // Clone the neighbor and put in the visited, if not present already  visited.put(neighbor, new Node(neighbor.val, new ArrayList()));  // Add the newly encountered node to the queue.  queue.add(neighbor);  }  // Add the clone of the neighbor to the neighbors of the clone node "n".  visited.get(n).neighbors.add(visited.get(neighbor));  }  }  // Return the clone of the node from visited.  return visited.get(node);  }  } |

**Complexity Analysis**

* Time Complexity : O(N + M)*O*(*N*+*M*), where N*N* is a number of nodes (vertices) and M*M* is a number of edges.
* Space Complexity : O(N)*O*(*N*). This space is occupied by the visited dictionary and in addition to that, space would also be occupied by the queue since we are adopting the BFS approach here. The space occupied by the queue would be equal to O(W)*O*(*W*) where W*W* is the width of the graph. Overall, the space complexity would be O(N)*O*(*N*).