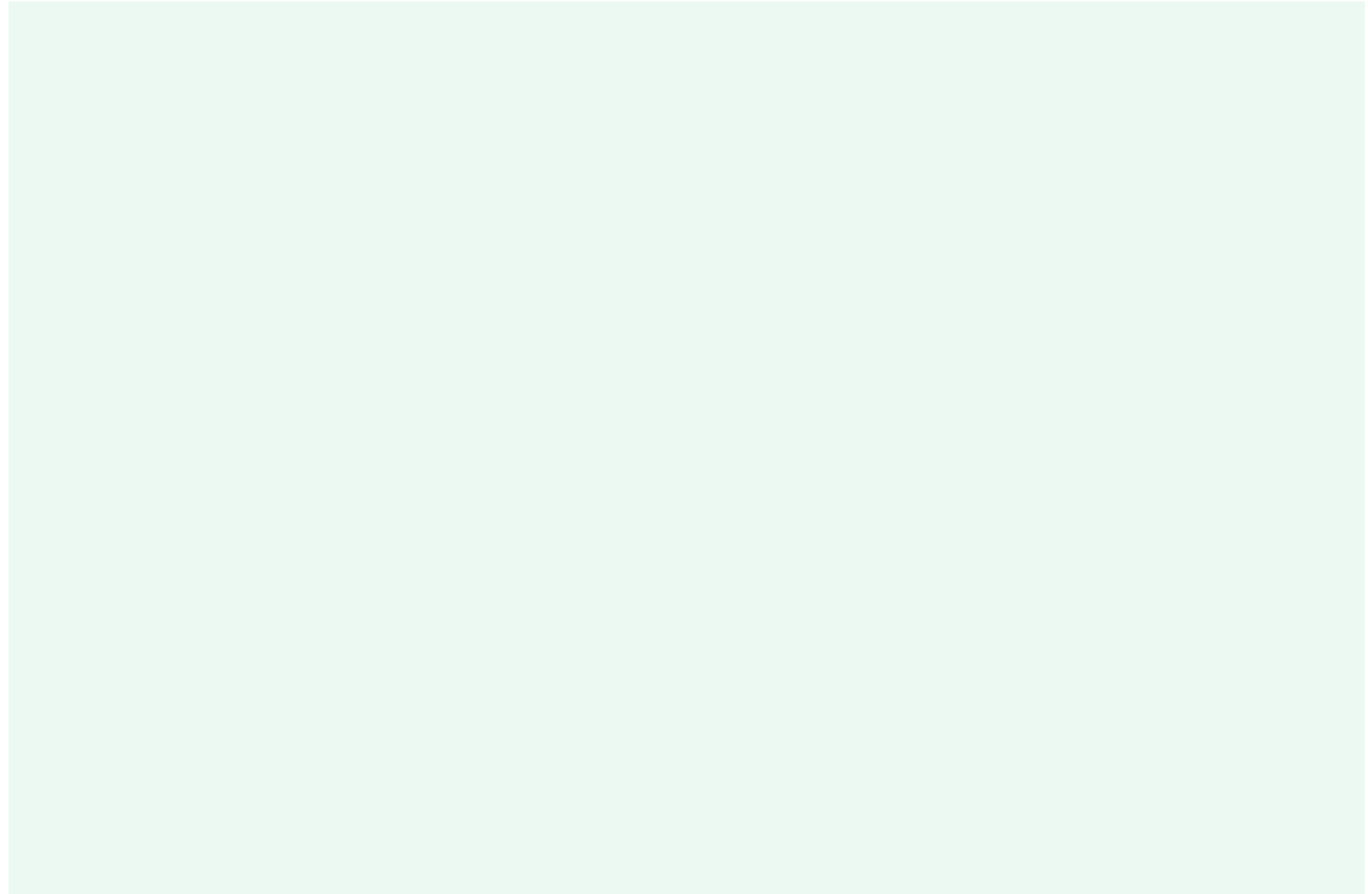


BFS Graph Traversal Explained





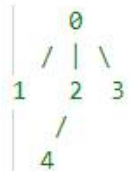
```

1  /*
2  Q. Breadth First Search (BFS) In Graph.
3  Given a directed graph. The task is to do Breadth First Traversal of this graph starting from 0.
4  Note: One can move from node u to node v only if there's an edge from u to v and find the BFS
5  traversal of the graph starting from the 0th vertex, from left to right according to the graph.
6  | Also, you should only take nodes directly or indirectly connected from Node 0 in consideration.

```

Example 1:

Input:



Output: 0 1 2 3 4

Explanation:

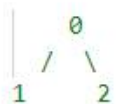
0 is connected to 1 , 2 , 3.

2 is connected to 4.

so starting from 0, it will go to 1 then 2 then 3. After this 2 to 4, thus bfs will be 0 1 2 3 4.

Example 2:

Input:



Output: 0 1 2

Explanation:

0 is connected to 1 , 2.

so starting from 0, it will go to 1 then 2, thus bfs will be 0 1 2.

```

45 public class BFS_of_graph {
46
47     // Time Complexity: O(N) + (2E -->total Degree) & Space Complexity: O(3N  $\cong$  N);
48     // Adding Neighbour Nodes
49     public static void addNeighbours(ArrayList<Integer> nums, Queue<Integer> q, boolean[] vis){
50         for(int val : nums){
51             if(vis[val] == false){
52                 // marking true the val in vis array before adding into the queue.
53                 vis[val] = true;
54                 q.add(val);
55             }
56         }
57     }
58     public static ArrayList<Integer> bfsOfGraph(int n, ArrayList<ArrayList<Integer>> adj) {
59         ArrayList<Integer> bfs = new ArrayList<>();
60         // 0-Based indexing - we will create a visited array of size n.
61         boolean[] vis = new boolean[n];
62         Queue<Integer> q = new LinkedList<>();
63         // adding 0 because it is 0-based indexing
64         q.add(0);
65         // marking 0 as true in visited array
66         vis[0] = true;
67         while(!q.isEmpty()){
68             int curr = q.remove();
69             // adding into the bfs array.
70             bfs.add(curr);
71
72             // Adding Neighbours of curr Node.
73             addNeighbours(adj.get(curr), q, vis);
74         }
75         return bfs;
76     }

```

```

77 public static void main(String[] args) {
78     ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
79     /*
80      // Adjacency List of Directed Graph
81      0-> 1, 2, 4
82      1
83      2 -> 4
84      3
85      4 ->
86     */
87
88     int n = 5;
89     for(int i = 0; i < n; i++){
90         adj.add(new ArrayList<>());
91     }
92     adj.get(0).add(1);
93     adj.get(0).add(2);
94     adj.get(0).add(3);
95     adj.get(2).add(4);
96     System.out.println(bfsOfGraph(n, adj));
97 }
98 }
99

```

Kindly explain the above code in details along with output



The provided code performs a Breadth-First Search (BFS) traversal on a directed graph starting from node 0. Below is a detailed explanation of the code and the outputs:

Code Explanation

1. `addNeighbours` Method

This method adds all the unvisited neighbors of a node to the BFS queue.

- **Input Parameters:**
 - ``nums``: The list of neighbors for the current node.
 - ``q``: The BFS queue to store nodes to be visited.
 - ``vis``: A boolean array indicating whether a node has been visited or not.
 - **Logic:**
 - For each neighbor ``val`` in ``nums``, if it is not visited (``vis[val] == false``):
 - Mark ``val`` as visited (``vis[val] = true``).
 - Add it to the BFS queue.
-

2. ``bfsOfGraph`` Method

This method performs BFS traversal of the graph.

- **Input Parameters:**
 - ``n``: Number of nodes in the graph.
 - ``adj``: Adjacency list representing the graph.
 - **Logic:**
 - Create a ``bfs`` list to store the BFS traversal result.
 - Use a ``vis`` array to keep track of visited nodes.
 - Initialize a queue ``q`` and start from node 0:
 - Add ``0`` to the queue and mark it as visited.
 - While the queue is not empty:
 - Remove the front node (``curr``) from the queue.
 - Add ``curr`` to the BFS result.
 - Add all unvisited neighbors of ``curr`` to the queue using the ``addNeighbours`` method.
 - Return the BFS result.
-

3. ``main`` Method

This method creates the graph and calls the BFS function.

- **Graph Representation:** The graph is represented using an adjacency list (``ArrayList<ArrayList<Integer>>``).
 - Example 1 Graph:

```
rust
```

```
0 -> 1, 2, 3
2 -> 4
```

- Example 2 Graph:

```
rust
```

```
0 -> 1, 2
```

- **Steps:**

- Initialize the adjacency list.
- Add directed edges between nodes as per the input graph.
- Call `bfsOfGraph(n, adj)` to perform BFS traversal and print the result.

Output Explanation

Example 1 Input

Graph:

```
rust
```

```
0 -> 1, 2, 3
2 -> 4
```

- **BFS Traversal:**

1. Start from node `0`: Add `1`, `2`, and `3` to the queue.
2. Visit node `1`: No neighbors to add.
3. Visit node `2`: Add `4` to the queue.
4. Visit node `3`: No neighbors to add.
5. Visit node `4`: No neighbors to add.

- **Output:** `0 1 2 3 4`

Example 2 Input

Graph:

```
rust

0 -> 1, 2
```

- **BFS Traversal:**
 1. Start from node `0`: Add `1` and `2` to the queue.
 2. Visit node `1`: No neighbors to add.
 3. Visit node `2`: No neighbors to add.
- **Output:** `0 1 2`

Time and Space Complexity

- **Time Complexity:**
 - $O(V + E)$, where V is the number of vertices and E is the number of edges. Traversing each vertex and its edges takes linear time.
- **Space Complexity:**
 - $O(V)$: For the `visited` array and the BFS queue.

This code efficiently computes the BFS traversal for any given graph.



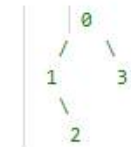

```

1  /*
2  Depth First Search (DFS) of Graph
3  Save Today, Earn More Tomorrow. Get 25% off on Complete Interview Preparation
4
5  You are given a connected undirected graph. Perform a Depth First Traversal of the graph.
6  Note: Use a recursive approach to find the DFS traversal of the graph starting
7  from the 0th vertex from left to right according to the graph.
8
9
10 Example 1:
11
12 Input: V = 5 , adj = [[2,3,1] , [0], [0,4], [0], [2]]
13
14      |
15      | 0
16      / | \
17      2 3  1
18      |
19      4
20
21 Output: 0 2 4 3 1
22 Explanation:
23 0 is connected to 2, 3, 1.
24 1 is connected to 0.
25 2 is connected to 0 and 4.
26 3 is connected to 0.
27 4 is connected to 2.
28 so starting from 0, it will go to 2 then 4,
29 and then 3 and 1.
30 Thus dfs will be 0 2 4 3 1.

```

Example 2:

Input: V = 4, adj = [[1,3], [2,0], [1], [0]]



Output: 0 1 2 3

Explanation:

0 is connected to 1 , 3.

1 is connected to 0, 2.

2 is connected to 1.

3 is connected to 0.

so starting from 0, it will go to 1 then 2

then back to 0 then 0 to 3

thus dfs will be 0 1 2 3.

*/

```
import java.util.ArrayList;
```

Codeium: Refactor | Explain

```
public class DFS_of_Graph {
```

```
    // Time Complexity: O(N) + (2E --> total Degree) & Space Complexity: O(3N ≅ N)
```

```
    static ArrayList<Integer> ls = new ArrayList<>();
```

```
    static boolean[] vis;
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
    public static void dfs(int node, ArrayList<ArrayList<Integer>> adj){
```

```
        vis[node] = true;
```

```
        ls.add(node);
```

```
        for(int it : adj.get(node)){
```

```
            if(!vis[it]){
```

```
                dfs(it, adj);
```

```
            }
```

```
        }
```

```
    }
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
    public static ArrayList<Integer> dfsOfGraph(int n, ArrayList<ArrayList<Integer>> adj) {
```

```
        vis = new boolean[n];
```

```
        dfs(0, adj);
```

```
        return ls;
```

```
    }
```

Codeium: Refactor | Explain | Generate Javadoc | X

```
    public static void dfsOfGraph(int n, ArrayList<ArrayList<Integer>> adj) {
```

```
Codeium: Refactor | Explain | Generate Javadoc | X
70 public static void main(String[] args) {
71     ArrayList < ArrayList < Integer >> adj = new ArrayList < > ();
72     for (int i = 0; i < 5; i++) {
73         adj.add(new ArrayList < > ());
74     }
75     adj.get(0).add(2);
76     adj.get(2).add(0);
77     adj.get(0).add(1);
78     adj.get(1).add(0);
79     adj.get(0).add(3);
80     adj.get(3).add(0);
81     adj.get(2).add(4);
82     adj.get(4).add(2);
83     System.out.println(dfsOfGraph(5, adj));
84 }
85 }
86
```

Kindly explain the above code in details along with output



The provided code performs a **Depth-First Search (DFS)** traversal on a graph using recursion. Below is a detailed explanation of the code and the outputs.

Code Explanation

1. DFS Functionality

Purpose: Traverse the graph using DFS starting from node 0.

2. `dfs` Method

This is the recursive function that performs DFS on the graph.

- **Input Parameters:**

- ``node``: The current node being visited.
 - ``adj``: Adjacency list representing the graph.
 - **Logic:**
 1. Mark the current node as visited (``vis[node] = true``).
 2. Add the current node to the result list ``ls``.
 3. For each neighbor of the current node:
 - If the neighbor is not visited, recursively call ``dfs`` for that neighbor.
-

3. ``dfsOfGraph`` Method

This is the entry point for DFS traversal.

- **Input Parameters:**
 - ``n``: Number of nodes in the graph.
 - ``adj``: Adjacency list representing the graph.
 - **Logic:**
 - Initialize a ``vis`` array to track visited nodes.
 - Call the recursive ``dfs`` method starting from node 0.
 - Return the result list ``ls`` after traversal.
-

4. ``main`` Method

This method initializes the graph and calls the DFS function.

- **Graph Representation:** The graph is represented using an adjacency list (``ArrayList<ArrayList<Integer>>``).
 - **Steps:**
 1. Create an adjacency list with edges between nodes as per the input graph.
 2. Call ``dfsOfGraph(n, adj)`` to perform DFS traversal and print the result.
-

Output Explanation

Example 1 Input

Graph:

```
rust

0 -> 2, 3, 1
1 -> 0
2 -> 0, 4
3 -> 0
4 -> 2
```

DFS Traversal:

1. Start at node `0`: Visit `0` and add it to the result (`ls` becomes `[0]`).
2. Visit neighbor `2`: Add it to the result (`ls` becomes `[0, 2]`).
3. Visit neighbor `4`: Add it to the result (`ls` becomes `[0, 2, 4]`).
4. Backtrack to `2`, then backtrack to `0`.
5. Visit neighbor `3`: Add it to the result (`ls` becomes `[0, 2, 4, 3]`).
6. Visit neighbor `1`: Add it to the result (`ls` becomes `[0, 2, 4, 3, 1]`).

Output: `0 2 4 3 1`

Example 2 Input

Graph:

```
rust

0 -> 1, 3
1 -> 0, 2
2 -> 1
3 -> 0
```

DFS Traversal:

1. Start at node `0`: Visit `0` and add it to the result (`res` becomes `[0]`).
2. Visit neighbor `1`: Add it to the result (`res` becomes `[0, 1]`).
3. Visit neighbor `2`: Add it to the result (`res` becomes `[0, 1, 2]`).
4. Backtrack to `1`, then backtrack to `0`.
5. Visit neighbor `3`: Add it to the result (`res` becomes `[0, 1, 2, 3]`).

Output: `0 1 2 3`

Time and Space Complexity

Time Complexity

- $O(V + E)$: Traverses all vertices and edges once.

Space Complexity

- $O(V)$: For the recursion stack and the visited array.

This code correctly implements DFS traversal using recursion, ensuring all reachable nodes from node 0 are visited.



```

3  /*
4  Graph Representation Using Adjacency List.
5
6  ①-----②
7  |       | \
8  |       |  ⑤
9  |       | /
10 ③-----④

11
12 Input:
13 n = 5;
14 m = 6;
15 m - lines:
16 [1, 2] -----> means (1 <----> 2) & (2 <-----> 1)
17 [1, 3]
18 [3, 4]
19 [2, 4]
20 [2, 5]
21 [4, 5]
22
23 Representation in List:
24
25 Step 1: Initialize N + 1 ArrayList (if 1-Based Indexing)
26
27 0 |
28 1 | -> ② -> ③
29 2 | -> ① -> ⑤ -> ⑤
30 3 | -> ① -> ⑤
31 4 | -> ③ -> ② -> ⑤
32 5 | -> ② -> ③
33
34 */
35

```

Codeium: Refactor | Explain

```

36 public class Graph_Representation_Using_Adjacency_List {
    Codeium: Refactor | Explain | Generate Javadoc | X
37     public static void representGraph(int n, int[][] input){
38         // Declaring ArrayList of ArrayList
39         ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
40         for(int i = 0; i < n + 1; i++){
41             adj.add(new ArrayList<>());
42         }
43         // Storing into list;
44         for(int i = 0; i < input.length; i++){
45             adj.get(input[i][0]).add(input[i][1]);
46             adj.get(input[i][1]).add(input[i][0]);
47         }
48         // Printing list
49         System.out.println(adj);
50     }
    Codeium: Refactor | Explain | Generate Javadoc | X
51     public static void main(String[] args) {
52         int n = 5;
53         int[][] input = {{1, 2}, {1, 3}, {3, 4}, {2, 4}, {2, 5}, {4, 5}};
54         representGraph(n, input);
55     }
56 }
57

```

Kindly explain the above code in details along with output



The code provided demonstrates how to represent an **undirected graph** using an adjacency list. Below is the detailed explanation of the code and the resulting output.

Code Explanation

1. Graph Representation

- **Input:**

- `n`: The number of vertices in the graph.
- `m`: The number of edges in the graph.
- An array of edges `input` where each edge is represented as `{u, v}`, meaning there is an edge between `u` and `v`.

- **Graph Structure:**

- The graph is **undirected**, so each edge `{u, v}` means:
 - `u` is connected to `v`.
 - `v` is connected to `u`.

- **Example Graph:**

```
makefile
```

```
Input:
```

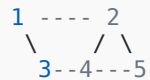
```
n = 5
```

```
Edges:
```

```
{1, 2}, {1, 3}, {3, 4}, {2, 4}, {2, 5}, {4, 5}
```

This creates the following graph:

```
lua
```



2. `representGraph` Method

This method creates the adjacency list for the graph.

- **Steps:**

1. **Initialization:**

- Create an `ArrayList` of `ArrayList<Integer>`. Each list represents the neighbors of a vertex.
- Initialize `n+1` lists because the vertices are 1-based indexed.

2. Add Edges:

- For each edge `{u, v}` in the input:
 - Add `v` to the list of `u`.
 - Add `u` to the list of `v` (because the graph is undirected).

3. Output:

- Print the adjacency list.

3. `main` Method

• Steps:

1. Define the number of vertices `n`.
2. Define the list of edges in `input` (a 2D array of pairs `{u, v}`).
3. Call `representGraph(n, input)` to create and print the adjacency list.

Output Explanation

Adjacency List Construction

The adjacency list representation of the graph looks like this:

- For each vertex `i`, its neighbors are stored in `adj.get(i)`.

For the input graph:

```
makefile

n = 5
Edges: {1, 2}, {1, 3}, {3, 4}, {2, 4}, {2, 5}, {4, 5}
```

The adjacency list becomes:

CSS

```
1 -> [2, 3]
2 -> [1, 4, 5]
3 -> [1, 4]
4 -> [3, 2, 5]
5 -> [2, 4]
```

Output:

CSS

```
[[], [2, 3], [1, 4, 5], [1, 4], [3, 2, 5], [2, 4]]
```

Here, the `0`th index is empty because the vertices are 1-based indexed.

Complexity Analysis

Time Complexity:

- **Initialization:** $O(n)$, to initialize $n+1$ empty lists.
- **Adding Edges:** $O(m)$, where m is the number of edges.
- Total: $O(n + m)$.


Space Complexity:

- $O(n + m)$, for storing the adjacency list.

This code efficiently constructs the adjacency list for an undirected graph based on the given edges, making it ready for graph algorithms like BFS, DFS, etc.



```

1  /*
2  Graph Representation Using Adjacency Matrix.
3
4  
5
6
7
8
9
10 Input:
11 n = 5;
12 m = 6;
13 m - lines:
14 [1, 2] -----> means (1 <----> 2) & (2 <-----> 1)
15 [1, 3]
16 [3, 4]
17 [2, 4]
18 [2, 5]
19 [4, 5]
20
21 Representation in Matrix:
22   0 1 2 3 4 5
23   ---
24 0 | 0 0 0 0 0 0
25 1 | 0 0 1 1 0 0
26 2 | 0 1 0 0 1 1
27 3 | 0 1 0 0 1 0
28 4 | 0 0 1 1 0 1
29 5 | 0 0 1 0 1 0
30
31 0 --> Represents no Edge
32 1 --> Represents There is edge between ith row node and jth col node;
33
34 */
35

```

```

36 public class Graph_Representation_Using_Adjacency_Matrix {
    Codeium: Refactor | Explain | Generate Javadoc | X
37     public static void representGraph(int n, int[][] input){
38         int[][] adj = new int[n + 1][n + 1];
39         for(int i = 0; i < input.length; i++){
40             adj[input[i][0]][input[i][1]] = 1;
41             adj[input[i][1]][input[i][0]] = 1;
42         }
43
44         // Printing Matrix
45         for(int[] nums : adj){
46             for(int i : nums){
47                 System.out.print(i + " ");
48             }
49             System.out.println();
50         }
51     }
    Codeium: Refactor | Explain | Generate Javadoc | X
52     public static void main(String[] args) {
53         int n = 5;
54         int[][] input = {{1, 2}, {1, 3}, {3, 4}, {2, 4}, {2, 5}, {4, 5}};
55         representGraph(n, input);
56     }
57 }
58

```

Kindly explain the above code in details along with output



The provided code demonstrates how to represent an **undirected graph** using an **adjacency matrix**. Below is a detailed explanation of the code and the resulting output.

Graph Representation Using Adjacency Matrix

Input Details

- n : Number of vertices in the graph.
- m : Number of edges in the graph.
- **input**: A list of edges, where each edge is represented as $\{u, v\}$:
 - An edge $\{u, v\}$ implies:
 - There is a connection between node u and node v .
 - Since the graph is **undirected**, the adjacency is mutual ($u \rightarrow v$ and $v \rightarrow u$).

Example Graph

Input:

```
makefile

n = 5 (vertices)
Edges: {1, 2}, {1, 3}, {3, 4}, {2, 4}, {2, 5}, {4, 5}
```

Graph Representation:

```
lua

1 ---- 2
 \    / \
  3--4---5
```

Adjacency Matrix

The adjacency matrix for this graph is a 2D array of size $(n+1) \times (n+1)$ (1-based indexing). The matrix has:

- 1 at position $[i][j]$ if there is an edge between vertex i and vertex j .
- 0 otherwise.

For the above graph, the matrix is:

```
lua
```

```

      0 1 2 3 4 5
    -----
0 | 0 0 0 0 0 0
1 | 0 0 1 1 0 0
2 | 0 1 0 0 1 1
3 | 0 1 0 0 1 0
4 | 0 0 1 1 0 1
5 | 0 0 1 0 1 0

```

Code Explanation

1. `representGraph` Method

This method constructs and prints the adjacency matrix.

- **Steps:**

1. **Matrix Initialization:**

- Create a 2D array `adj` of size `(n+1) x (n+1)` initialized to `0`.
- The extra row and column accommodate 1-based indexing.

2. **Populate Matrix:**

- For each edge `{u, v}` in the `input`:
 - Set `adj[u][v] = 1` and `adj[v][u] = 1` (because the graph is undirected).

3. **Print Matrix:**

- Iterate through the rows and print each row of the matrix.

2. `main` Method

- **Steps:**

1. Define the number of vertices `n`.
2. Define the list of edges `input` (2D array).
3. Call `representGraph(n, input)` to construct and print the adjacency matrix.

Output Explanation

For the given input:

```
makefile

n = 5
Edges: {1, 2}, {1, 3}, {3, 4}, {2, 4}, {2, 5}, {4, 5}
```

The output is the adjacency matrix:

```
0 0 0 0 0 0
0 0 1 1 0 0
0 1 0 0 1 1
0 1 0 0 1 0
0 0 1 1 0 1
0 0 1 0 1 0
```

Here:

- Row `1` corresponds to node `1`, showing edges to `2` and `3`.
- Row `2` corresponds to node `2`, showing edges to `1`, `4`, and `5`.
- Row `3` corresponds to node `3`, showing edges to `1` and `4`.
- Row `4` corresponds to node `4`, showing edges to `2`, `3`, and `5`.
- Row `5` corresponds to node `5`, showing edges to `2` and `4`.

Complexity Analysis

Time Complexity

- **Matrix Initialization:** $O(n^2)$, to initialize the matrix.
- **Adding Edges:** $O(m)$, where `m` is the number of edges.
- Total: $O(n^2 + m)$.

Space Complexity

- $O(n^2)$, for storing the adjacency matrix.

This code effectively represents the graph using an adjacency matrix and outputs it, which is useful for algorithms that require direct edge lookups between nodes.