|  |  |
| --- | --- |
| **Find eventual safe state in C++** | |
| #include <bits/stdc++.h>  using namespace std;  class Solution {  private:      bool dfsCheck(int node, vector<int> adj[], int vis[],   int pathVis[],          int check[]) {          vis[node] = 1;          pathVis[node] = 1;          check[node] = 0;          // traverse for adjacent nodes          for (auto it : adj[node]) {              // when the node is not visited              if (!vis[it]) {              if (dfsCheck(it, adj, vis, pathVis, check) == true) {                      check[node] = 0;                      return true;                  }              }              // if the node has been previously visited              // but it has to be visited on the same path              else if (pathVis[it]) {                  check[node] = 0;                  return true;              }          }          check[node] = 1;          pathVis[node] = 0;          return false;      }  public:      vector<int> eventualSafeNodes(int V, vector<int> adj[]) {          int vis[V] = {0};          int pathVis[V] = {0};          int check[V] = {0};          vector<int> safeNodes;          for (int i = 0; i < V; i++) {              if (!vis[i]) {                  dfsCheck(i, adj, vis, pathVis, check);              }          }          for (int i = 0; i < V; i++) {              if (check[i] == 1) safeNodes.push\_back(i);          }          return safeNodes;      }  };  int main() {      //V = 12;      vector<int> adj[12] = {{1}, {2}, {3}, {4, 5}, {6}, {6}, {7}, {}, {1, 9}, {10},          {8},{9}};      int V = 12;      Solution obj;      vector<int> safeNodes = obj.eventualSafeNodes(V, adj);      for (auto node : safeNodes) {          cout << node << " ";      }      cout << endl     return 0;  } | **Dry Run:**  Let's dry-run the code with the given graph:  **Adjacency List** for the graph:  0 -> 1  1 -> 2  2 -> 3  3 -> 4, 5  4 -> 6  5 -> 6  6 -> 7  7 -> (no outgoing edges)  8 -> 1, 9  9 -> 10  10 -> 8  11 -> 9  **DFS Exploration**:   1. **Starting DFS from node 0**:    * vis[0] = 1, pathVis[0] = 1    * Go to node 1: vis[1] = 1, pathVis[1] = 1    * Go to node 2: vis[2] = 1, pathVis[2] = 1    * Go to node 3: vis[3] = 1, pathVis[3] = 1    * Go to node 4: vis[4] = 1, pathVis[4] = 1    * Go to node 6: vis[6] = 1, pathVis[6] = 1    * Go to node 7: vis[7] = 1, pathVis[7] = 1      + Node 7 has no outgoing edges, so it is safe: check[7] = 1    * Node 6 is safe as it leads to safe node 7: check[6] = 1    * Node 4 is safe as it leads to safe node 6: check[4] = 1    * Node 3 is safe as it leads to safe nodes 4 and 5: check[3] = 1    * Node 2 is safe as it leads to safe node 3: check[2] = 1    * Node 1 is safe as it leads to safe node 2: check[1] = 1    * Node 0 is safe as it leads to safe node 1: check[0] = 1 2. **DFS from node 8**:    * vis[8] = 1, pathVis[8] = 1    * Go to node 1, but node 1 is already visited and part of the current path (cycle detected).    * Hence, node 8 is unsafe. 3. **DFS from node 9**:    * vis[9] = 1, pathVis[9] = 1    * Go to node 10: vis[10] = 1, pathVis[10] = 1    * Go to node 8, and since 8 is already visited and part of the current DFS path, node 9 is unsafe. 4. **DFS from node 10**:    * Same as node 9, it leads to node 8, so it's unsafe. 5. **DFS from node 11**:    * vis[11] = 1, pathVis[11] = 1    * Go to node 9, which is unsafe.    * Therefore, node 11 is unsafe.   **Final Results:**   * The safe nodes are **[0, 1, 2, 3, 4, 5, 6, 7]**.   **Output:**  0 1 2 3 4 5 6 7 |
| **Output:-**  0 1 2 3 4 5 6 7 | |