DAY 7

Name:-Piyanshu Raj

UID: -22BCS15295

Date: -27/12/2024

Section: - 620 A

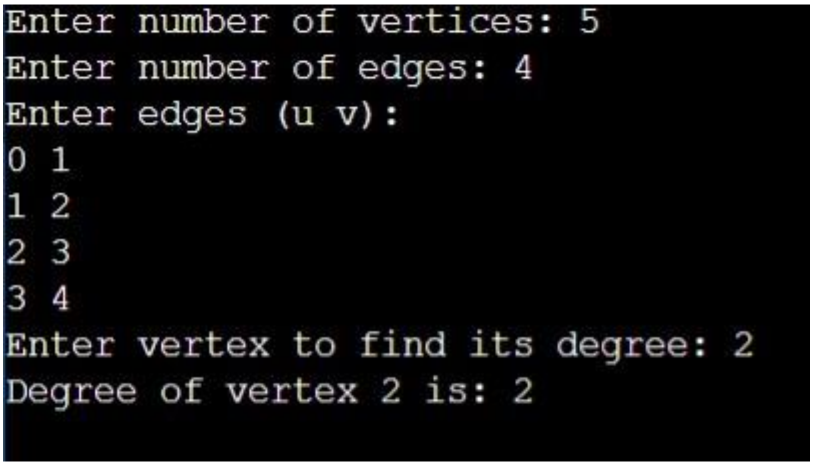
Question 1: -

WAP to find the degree of given vertex in a graph

```
#include <iostream>

#include <vector> using

namespace std;


class Graph { public:

   int V;

   vector<vector<int>> adjList;

   Graph(int vertices) {

V = vertices;

adjList.resize(V);

   }

   void addEdge(int u, int v) {

adjList[u].push_back(v);        adjList[v].push_back(u);

   }

    int getDegree(int vertex) {

       return adjList[vertex].size();
   }
};

int main() {     int vertices, edges;
cout << "Enter number of vertices: ";
cin >> vertices;
   Graph g(vertices);

   cout << "Enter number of edges: ";
cin >> edges;

   cout << "Enter edges (u v): \n";
for (int i = 0; i < edges; i++) {
int u, v;        cin >> u >> v;
       g.addEdge(u, v);
   }

   int vertex;
   cout << "Enter vertex to find its degree: ";
   cin >> vertex;

   cout << "Degree of vertex " << vertex << " is: " <<
g.getDegree(vertex) << endl;
```

```
    return 0;
}
```

OUTPUT: -



```
Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
0 1
1 2
2 3
3 4
Enter vertex to find its degree: 2
Degree of vertex 2 is: 2
```

Question 2: -

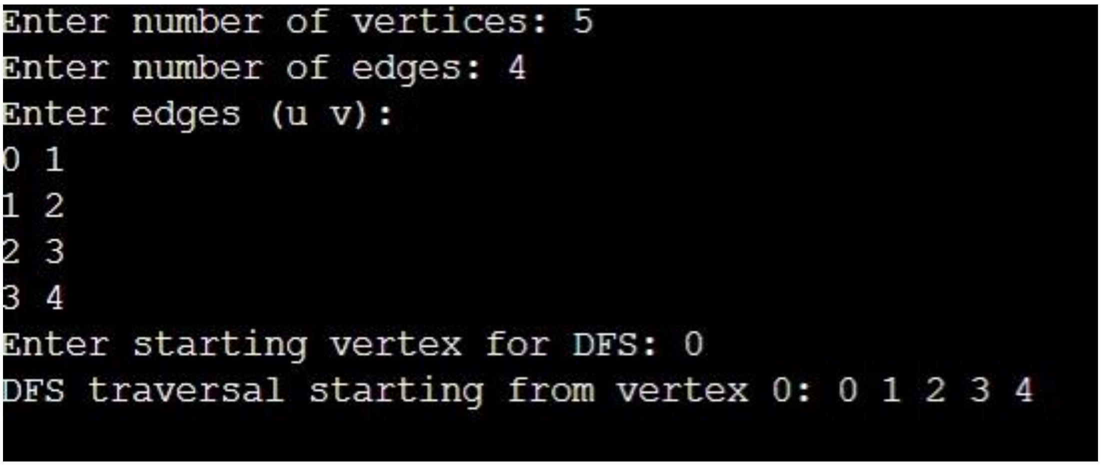
WAP for DFS

```
#include <iostream>
#include <vector>
#include <stack> using
namespace std;

class Graph { public:
    int V;
    vector<vector<int>> adjList;
    Graph(int vertices) {
V = vertices;
adjList.resize(V);
    }
```

```
   void addEdge(int u, int v) {
adjList[u].push_back(v);        adjList[v].push_back(u);
   }
   void DFS(int start) {
vector<bool> visited(V, false);
stack<int> s;
       s.push(start);
visited[start] = true;

       while (!s.empty()) {            int
node = s.top();            s.pop();
cout << node << " ";            for (int
adj : adjList[node]) {               if
(!visited[adj]) {
visited[adj] = true;
s.push(adj);
               }
           }
       }
       cout << endl;
   }
};
```

```
int main() {     int vertices, edges;
cout << "Enter number of vertices: ";
cin >> vertices;
   Graph g(vertices);

   cout << "Enter number of edges: ";
cin >> edges;

   cout << "Enter edges (u v): \n";
for (int i = 0; i < edges; i++) {
int u, v;        cin >> u >> v;
      g.addEdge(u, v);
   }

   int start;
   cout << "Enter starting vertex for DFS: ";     cin >> start;

   cout << "DFS traversal starting from vertex " << start << ": ";
g.DFS(start);

   return 0;
}
```

OUTPUT: -



```
Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
0 1
1 2
2 3
3 4
Enter starting vertex for DFS: 0
DFS traversal starting from vertex 0: 0 1 2 3 4
```

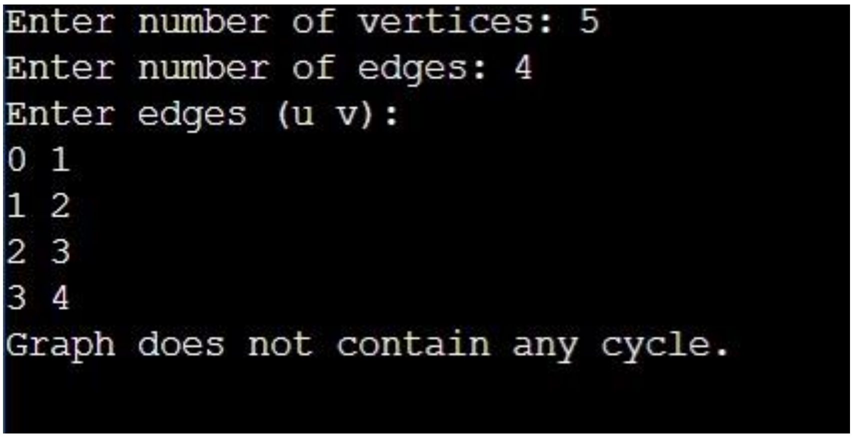
Question 3: -

WAP to detect a cycle in undirected graph

```
#include <iostream>
#include <vector> using
namespace std;


class Graph { public:
   int V;
   vector<vector<int>> adjList;
   Graph(int vertices) {
V = vertices;
adjList.resize(V);
   }
   void addEdge(int u, int v) {
adjList[u].push_back(v);        adjList[v].push_back(u);
   }
   bool DFS(int node, vector<bool>& visited, vector<int>& parent) {
visited[node] = true;        for (int adj : adjList[node]) {            if
(!visited[adj]) {                parent[adj] = node;                if (DFS(adj,
visited, parent))                    return true;
            }
            else if (parent[node] != adj) {
return true;
            }
        }
        return false;
    }
    bool detectCycle() {
vector<bool> visited(V, false);
vector<int> parent(V, -1);        for
(int i = 0; i < V; i++) {            if
(!visited[i]) {                if (DFS(i,
visited, parent))                    return
true;
            }
        }
        return false;
    }
};
```

```
int main() {     int vertices, edges;
cout << "Enter number of vertices: ";
cin >> vertices;
   Graph g(vertices);

   cout << "Enter number of edges: ";
cin >> edges;

   cout << "Enter edges (u v): \n";
for (int i = 0; i < edges; i++) {
int u, v;         cin >> u >> v;
      g.addEdge(u, v);
   }

   if (g.detectCycle()) {
      cout << "Graph contains a cycle." << endl;
   } else {
      cout << "Graph does not contain any cycle." << endl;
   }

   return 0;
}
```

OUTPUT: -



```
Enter number of vertices: 5
Enter number of edges: 4
Enter edges (u v):
0 1
1 2
2 3
3 4
Graph does not contain any cycle.
```

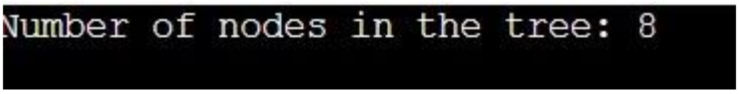
Question 4: -

Given the root of complete binary tree return the number of nodes in the tree

```
#include <iostream> using

namespace std; struct

TreeNode {

   int val;

   TreeNode *left;

   TreeNode *right;

   TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};


class Solution { public:

   int getHeight(TreeNode* root) {

int height = 0;        while (root) {

height++;           root = root->left;

       }

       return height;
```

```
    }
    int countNodes(TreeNode* root) {
        if (!root) return 0;

        int height = getHeight(root);
if (height == 0) return 0;

        int left = 1, right = (1 << height) - 1;
while (left < right) {
            int mid = (left + right + 1) / 2;
if (exists(root, height, mid)) {
left = mid;             } else {
right = mid - 1;
            }
        }

        return left + (1 << (height - 1)) - 1;
    }

private:
    bool exists(TreeNode* root, int height, int index) {
int left = 0, right = (1 << height) - 1;        for (int i = 0;
i < height - 1; i++) {            int mid = (left + right) / 2;
if (index <= mid) {                 root = root->left;
right = mid;            } else {                 root = root-
>right;                 left = mid + 1;
            }
        }
        return root != nullptr;
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);     root->right
= new TreeNode(3);     root->left->left =
new TreeNode(4);     root->left->right = new
TreeNode(5);     root->right->left = new
TreeNode(6);

    Solution solution;
    cout << "Number of nodes in the tree: " <<
solution.countNodes(root) << endl;

    return 0;
}
```

OUTPUT: -



```
Number of nodes in the tree: 8
```

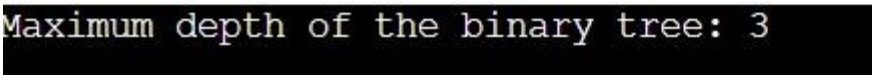
Question 5: -

A binary tree find the max depth of binary tree

```
#include <iostream> using
namespace std; struct
TreeNode {

   int val;
   TreeNode *left;
   TreeNode *right;
   TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};


class Solution { public:
   int maxDepth(TreeNode* root) {
      if (root == nullptr) {
return 0;
      }
      int leftDepth = maxDepth(root->left);
int rightDepth = maxDepth(root->right);
return 1 + max(leftDepth, rightDepth);
   }
};
```

```
int main() {
   TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);     root->right
= new TreeNode(3);     root->left->left =
new TreeNode(4);     root->left->right = new
TreeNode(5);


   Solution solution;
   cout << "Maximum depth of the binary tree: " <<
solution.maxDepth(root) << endl;


   return 0;
}
```

OUTPUT: -



```
Maximum depth of the binary tree: 3
```

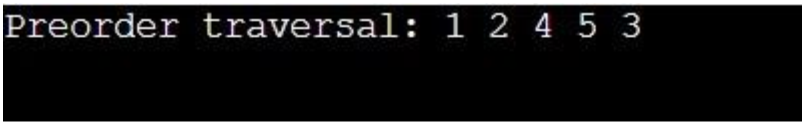
Question 6: -

Given the root of binary tree return preorder traverse of its node value

```
#include <iostream>
#include <vector> using
namespace std; struct
TreeNode {
   int val;
   TreeNode* left;
```

```
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
vector<int> result;         preorderHelper(root,
result);         return result;
    }

private:
    void preorderHelper(TreeNode* node, vector<int>& result) {
if (node == nullptr) {             return;
        }
        result.push_back(node->val);         preorderHelper(node-
>left, result);         preorderHelper(node->right, result);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);     root->right
= new TreeNode(3);     root->left->left =
new TreeNode(4);     root->left->right = new
TreeNode(5);     Solution solution;

    vector<int> result = solution.preorderTraversal(root);
cout << "Preorder traversal: ";     for (int val : result) {
cout << val << " ";
    }
    cout << endl;


    return 0;
}
```

OUTPUT: -



```
Preorder traversal: 1 2 4 5 3
```

Question 7: -

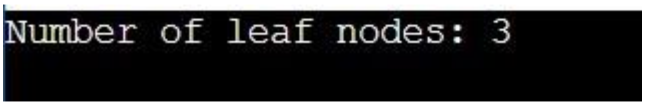
given a binary tree the task is to count the leaf node A node is a leaf node is both the leaf and right value are null

```
#include <iostream>
using namespace std;
struct TreeNode {
int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

```
class Solution { public:
    int countLeafNodes(TreeNode* root) {
        if (root == nullptr) {
return 0;
        }
        if (root->left == nullptr && root->right == nullptr) {
return 1;
        }
        return countLeafNodes(root->left) + countLeafNodes(root>right);
    }
};

int main() {
    TreeNode* root = new TreeNode(1);
root->left = new TreeNode(2);     root->right
= new TreeNode(3);     root->left->left =
new TreeNode(4);     root->left->right = new
TreeNode(5);

    Solution solution;
    int leafCount = solution.countLeafNodes(root);     cout
<< "Number of leaf nodes: " << leafCount << endl;
```

```
   return 0;
}
```

OUTPUT: -



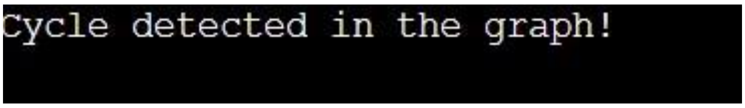
```
Number of leaf nodes: 3
```

Question 8: -

implementation off cyclic graph

```
#include <iostream>
#include <vector> #include
<unordered_map> using
namespace std;

class Graph { private:
   unordered_map<int, vector<int>> adjList;

public:
   void addEdge(int src, int dest) {
adjList[src].push_back(dest);
   }
   bool detectCycleUtil(int node, unordered_map<int, int>& visited) {
      if (visited[node] == 1) {
return true;
      }
      visited[node] = 1;
      for (int neighbor : adjList[node]) {
            if (visited[neighbor] != 2 && detectCycleUtil(neighbor, visited))
{
                return true;
            }
        }
        visited[node] = 2;
return false;
    }
    bool detectCycle() {
        unordered_map<int, int> visited;
for (const auto& pair : adjList) {
if (visited[pair.first] == 0) {
                if (detectCycleUtil(pair.first, visited)) {
return true;
                }
            }
        }
        return false;
    }
};

int main() {
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 0);

    if (g.detectCycle()) {
        cout << "Cycle detected in the graph!" << endl;
    } else {
        cout << "No cycle detected in the graph." << endl;
    }

    return 0;
}
```

OUTPUT: -



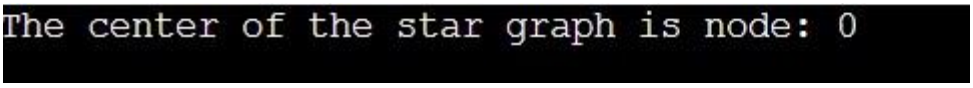
```
Cycle detected in the graph!
```

Question 9: -

find the centre of the star graph

```
#include <iostream>
#include <vector> using
namespace std; int
findCenter(vector<vect
or<int>>& adjList) {

    int n = adjList.size();       for
(int i = 0; i < n; i++) {        if
(adjList[i].size() == n - 1) {
return i;
        }
    }

    return -1
}

int main() {
int n = 5;
    vector<vector<int>> adjList(n);
adjList[0] = {1, 2, 3, 4};
adjList[1] = {0};      adjList[2] =
{0};      adjList[3] = {0};
adjList[4] = {0};

    int center = findCenter(adjList);
    cout << "The center of the star graph is node: " << center << endl;
return 0;
}
```

OUTPUT: -



```
The center of the star graph is node: 0
```

Question 10: -

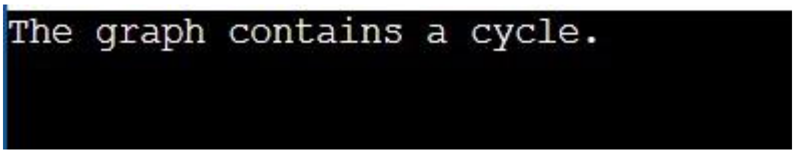
Write a program to detect a cycle in a directed graph by using DFS

```
#include <iostream>
#include <vector> using
namespace std;

class Graph { public:
   int V;
   vector<vector<int>> adj;

   Graph(int V);       void
addEdge(int u, int v);
   bool dfs(int node, vector<bool>& visited, vector<bool>&
recursionStack);       bool hasCycle();
};
Graph::Graph(int V) {     this-
>V = V;     adj.resize(V);
}
void Graph::addEdge(int u, int v) {
adj[u].push_back(v);
}
bool Graph::dfs(int node, vector<bool>& visited, vector<bool>&
recursionStack) {     visited[node] = true;
recursionStack[node] = true;     for (int neighbor : adj[node]) {
if (recursionStack[neighbor]) {            return true;
       }
```

```
        if (!visited[neighbor] && dfs(neighbor, visited, recursionStack)) {
return true;
        }
    }
    recursionStack[node] = false;
return false;
}
bool Graph::hasCycle() {
vector<bool> visited(V, false);
vector<bool> recursionStack(V, false);
for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            if (dfs(i, visited, recursionStack)) {
return true;
            }
        }
    }
    return false;
}

int main() {
int V = 4;
Graph g(V);
```

```
   g.addEdge(0, 1);
   g.addEdge(1, 2);
   g.addEdge(2, 3);
   g.addEdge(3, 1);
if (g.hasCycle()) {
        cout << "The graph contains a cycle." << endl;
   } else {
        cout << "The graph does not contain a cycle." << endl;
   }

   return 0;
}
```

OUTPUT: -



```
The graph contains a cycle.
```