Aryan Nanda **22107003**

Experiment 10

Aim : To check if the graph is Cyclic or Acyclic

Theory : In graph theory, a cyclic graph contains at least one cycle, while an acyclic graph has no cycles. A cycle in a graph is a path that starts and ends at the same vertex, traversing distinct edges without repeating vertices except for the starting and ending vertices.
To check if a graph is cyclic or acyclic, various algorithms can be applied. One common approach is using Depth-First Search (DFS) to detect cycles in an undirected graph. Here's how it works:
An **adjacency list** is a data structure used to represent connections or relationships between vertices (or nodes) in a graph. It's a compact way to store a graph where each vertex in the graph is associated with a list of its neighboring vertices.

Detecting Cycles Using DFS:

- Initialization: Start DFS traversal from any vertex.
- Traverse Graph: While traversing the graph, mark each visited vertex.
- Back Edge Detection: While exploring edges from a vertex, if DFS visits an adjacent vertex that's already visited and isn't the parent of the current vertex (in the DFS traversal), it indicates the presence of a cycle.

Aryan Nanda **22107003**

Algorithm -
Graph Creation:-
Classes:

- ListNode: Represents a node in the graph. It contains an integer data and a pointer next to the next node in the linked list.
- List: Implements a linked list to maintain vertices. Each element in this list is a ListNode.
- Graph: Represents the graph structure. It contains:
  - V - The number of vertices in the graph.
  - adjacencyList - An array of List objects. Each element of this array represents a vertex in the graph. Each List maintains the adjacent vertices for the corresponding vertex.

Graph Representation:

- Adjacency List:
  - adjacencyList is an array of List objects, where each index represents a vertex.
  - Each List maintains the adjacent vertices for a specific vertex using a linked list (ListNode).
  - push_back(int val): Adds a new node to the end of the linked list in the List object.
  - getHead(): Returns the head of the linked list associated with a particular vertex.

Graph Operations:

- Constructor (Graph(int V) { ... }):
  - Initializes the graph with V vertices.
  - Allocates memory for the adjacency list, creating an array of List objects.
- addEdge(int v, int u):
  - Adds an edge between vertices v and u.
  - Updates the adjacency list for both v and u by adding the opposite vertex to each other's lists.
- checkCycle():
  - Detects whether the graph contains any cycles.
  - Utilizes DFS through checkCycleUtil() to traverse the graph, marking visited vertices and checking for cycles.
- printGraph():
  - Displays the adjacency list representation of the graph.
  - Prints each vertex along with its adjacent vertices.

Main Function:

- User Interaction:
  - Asks the user for the number of vertices and edge inputs until -1 -1 is entered.
  - Constructs the graph by adding the provided edges.
  - Checks if the graph contains cycles and prints the result.

- Prints the adjacency list representation of the graph.

Cyclic or Acyclic:-

checkCycleUtil() Function:

- This function is a utility for the cycle detection process. It is called recursively from checkCycle() to perform the cycle check.

checkCycle() Function:

Initialization:

- Initializes a boolean array visited to keep track of visited vertices.
- Loops through each vertex in the graph.

Cycle Detection:

- Calls the checkCycleUtil() function for each unvisited vertex.
- Inside checkCycleUtil():
  - Marks the current vertex as visited and explores its adjacent vertices.
  - If an adjacent vertex is visited and is not the parent of the current vertex, a cycle is detected.

Return Result:

- If the checkCycleUtil() detects a cycle during the traversal, the checkCycle() function returns true, indicating that the graph contains a cycle.
- If no cycle is detected after checking all vertices, the function returns false, indicating that the graph is acyclic.

Cycle Detection Logic:

- Uses a DFS-based approach to explore the graph.
- While traversing, it marks visited vertices and tracks the parent of each vertex.
- Detects a cycle if it encounters a visited vertex that isn't the parent of the current vertex.

Code:-

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

class ListNode {
public:
   int data;
   ListNode* next;

   ListNode(int val) : data(val), next(nullptr) {}
};
```

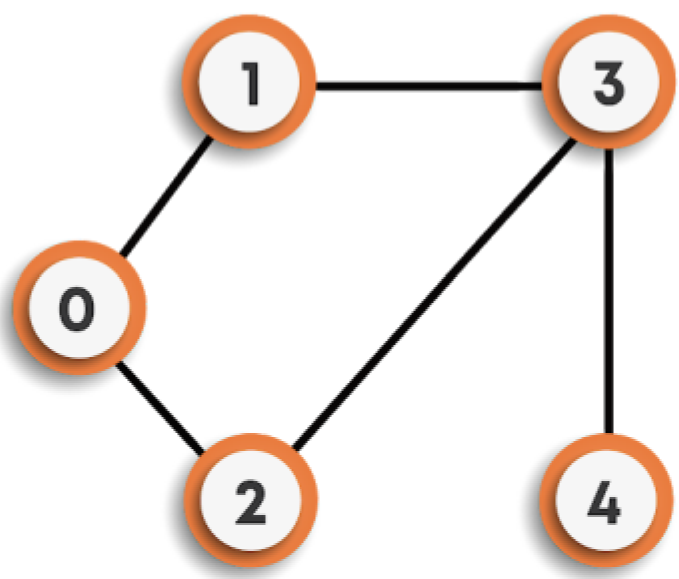
```
class List {
private:
    ListNode* head;

public:
    List() : head(nullptr) {}

    void push_back(int val) {
        ListNode* newNode = new ListNode(val);
        if (!head) {
            head = newNode;
            return;
        }

        ListNode* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }

    ListNode* getHead() {
        return head;
    }
};

class Graph {
    int V;
    List* adjacencyList;
    void DFS(int v, bool visited[]) {
        visited[v] = true;
        ListNode* headNode = adjacencyList[v].getHead();
        while (headNode) {
            int adjVertex = headNode->data;
            if (!visited[adjVertex]) {
                DFS(adjVertex, visited);
            }
            headNode = headNode->next;
        }
    }
    // bool checkCycleUtil(int v, bool visited[], int parent) {
    //     visited[v] = true;
```

```
    //      ListNode* headNode = adjacencyList[v].getHead();
    //      while (headNode) {
    //          int adjVertex = headNode->data;

    //          if (!visited[adjVertex]) {
    //              if (checkCycleUtil(adjVertex, visited, v))
    //                  return true;
    //          } else if (adjVertex != parent) {
    //              return true;
    //          }

    //          headNode = headNode->next;
    //      }

    //      return false;
    // }

public:
    Graph(int V) {
        this->V = V;
        adjacencyList = new List[V];
    }

    void addEdge(int v, int u) {
        adjacencyList[v].push_back(u);
        adjacencyList[u].push_back(v);
    }

    // bool checkCycle() {
    //      bool* visited = new bool[V]();
    //      for (int i = 0; i < V; ++i) {
    //          if (!visited[i] && checkCycleUtil(i, visited, -1)) {
    //              delete[] visited;
    //              return true;
    //          }
    //      }
    //      delete[] visited;
    //      return false;
    // }
    bool isConnected() {
        bool* visited = new bool[V]();
```

```
        DFS(0, visited);

        for (int i = 0; i < V; ++i) {
            if (!visited[i]) {
                delete[] visited;
                return false;
            }
        }
        delete[] visited;
        return true;
    }

    void printGraph() {
        for (int i = 0; i < V; ++i) {
            cout << "Vertex " << i << " --> ";
            ListNode* currentNode = adjacencyList[i].getHead();
            while (currentNode) {
                cout << currentNode->data << " ";
                currentNode = currentNode->next;
            }
            cout << endl;
        }
    }
};
int main() {
    int v, u;
    cout<<"Enter the number of vertices in the graph "<<endl;
    int vertices;
    cin>>vertices;
    Graph g(vertices);
    cout << "Enter edges (Vertex1 Vertex2) [-1 -1 to stop]:\n";
    while (true) {
        cin >> v >> u;
        if (v == -1 && u == -1)
            break;
        g.addEdge(v, u);
    }
    // if (g.checkCycle())
    //     cout << "Graph is cyclic" << endl;
    // else
    //     cout << "Graph is acyclic" << endl;
```

```
    if (g.isConnected())
        cout << "Graph is connected" << endl;
    else
        cout << "Graph is not connected" << endl;

    g.printGraph();
    return 0;
}
```

Example 1:-



Aryan Nanda **22107003**

Input:-

```
~/Doc/SY/DSA Lab/Experiment9                127 ✘
./a.out
Enter the number of vertices in the graph
5
Enter edges (Vertex1 Vertex2) [-1 -1 to stop]:
0 1
0 2
3 1
3 2
3 4
-1 -1
```

Structure of Graph:-

Vertex 0 --> 1 2

Vertex 1 --> 0 3

Vertex 2 --> 0 3

Vertex 3 --> 1 2 4

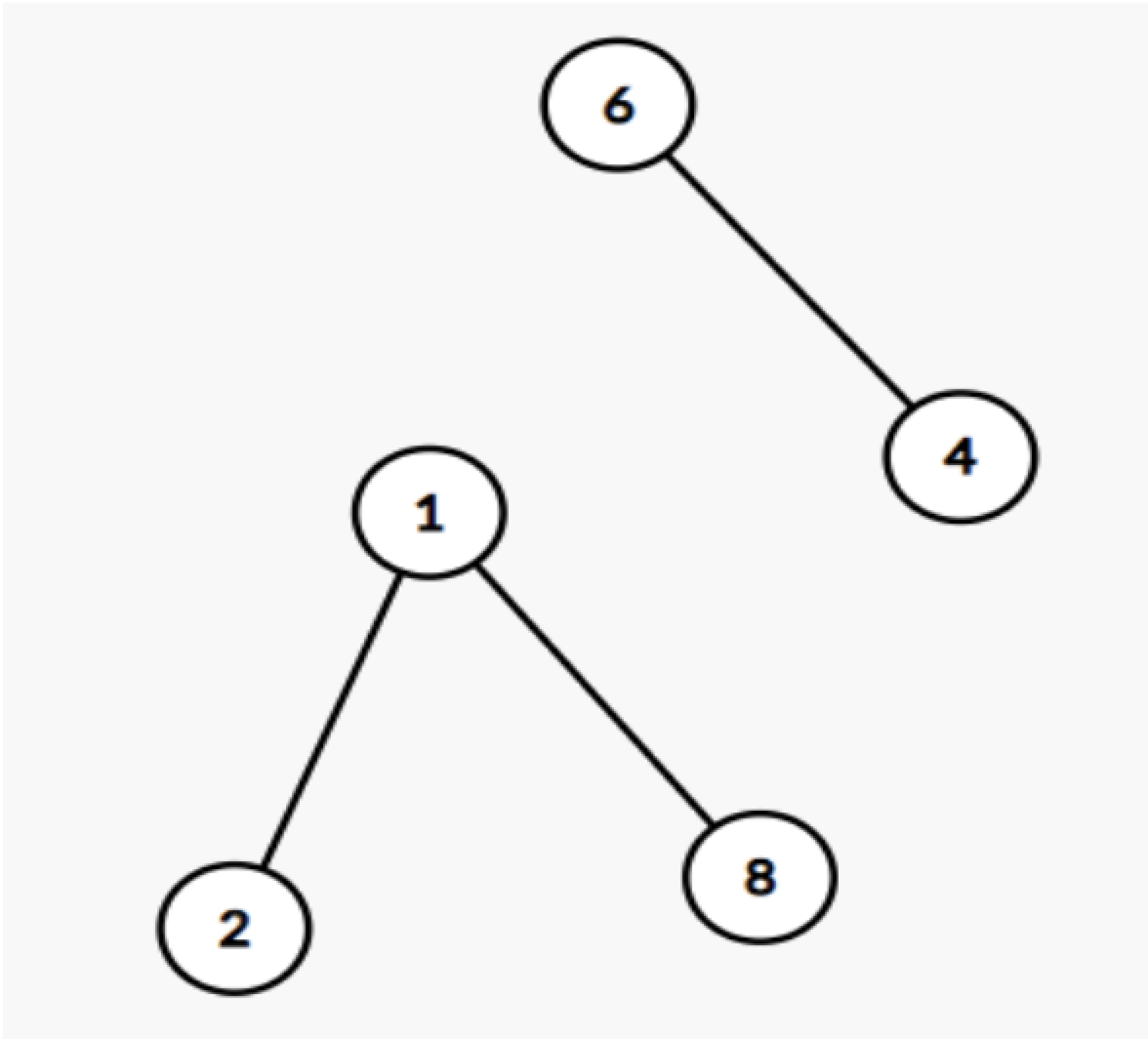
Vertex 4 --> 3

Aryan Nanda **22107003**

Output:-

```
~/Doc/SY/DSA Lab/Experiment9    127 ✘   15
./a.out
Enter the number of vertices in the graph
5
Enter edges (Vertex1 Vertex2) [-1 -1 to stop]:
0 1
0 2
3 1
3 2
3 4
-1 -1
Graph is connected
Vertex 0 --> 1 2
Vertex 1 --> 0 3
Vertex 2 --> 0 3
Vertex 3 --> 1 2 4
Vertex 4 --> 3
```

Example 2 -



Structure of graph:-
Vertex 0 -->
Vertex 1 --> 2 3
Vertex 2 --> 1 4 5
Vertex 3 --> 1
Vertex 4 --> 2
Input and Output:-

```
~/Doc/SY/DSA Lab/Experiment9
./a.out
Enter the number of vertices in the graph
5
Enter edges (Vertex1 Vertex2) [-1 -1 to stop]
1 2
1 3
0 4
-1 -1
Graph is not connected
Vertex 0 --> 4
Vertex 1 --> 2 3
Vertex 2 --> 1
Vertex 3 --> 1
Vertex 4 --> 0
```

Conclusion - From this experiment, we learned how to implement the graph using adjacency list and how to detect if there is a cycle in the graph using dfs traversal.