Data Structures Lab
Session 12

Course: Data Structures (CS2001) **Semester:** Fall 2022
Instructor: **T.A:** N/A

Note:

- Lab manual cover following topics
  { Simple representation of Graph ,Depth First Search}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

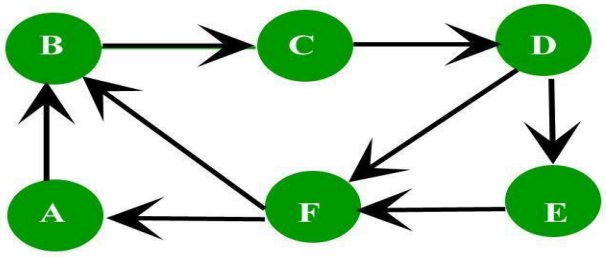
A simple Representation of graph

Introduction: A set of items connected by edges. **Each** item is called a vertex or node. **Formally,** a graph is a set of vertices and a binary relation between vertices, adjacency.

Graphs are ubiquitous in computer science. They are used to model real-world systems such as the Internet (each node represents a router and each edge represents a connection between routers); airline connections (each node is an airport and each edge is a flight); or a city road network (each node represents an intersection and each edge represents a block). The wireframe drawings in computer graphics are another example of graphs.

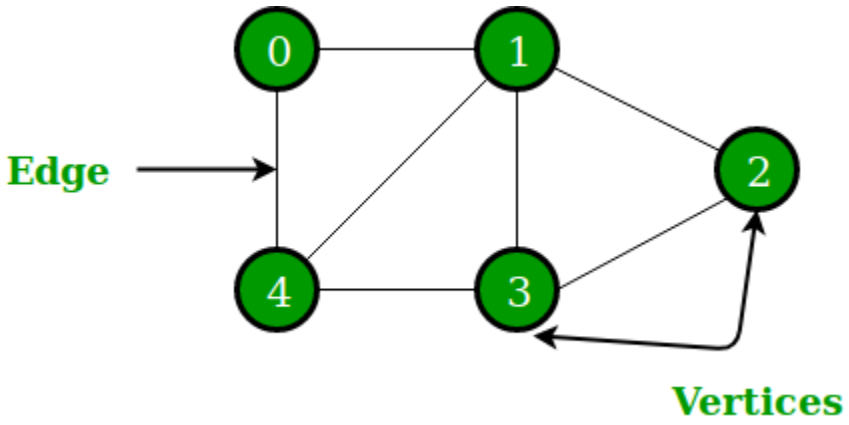
A graph is a data structure that consists of the following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pairs of the form (u, v) called an edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

A **directed graph** is a set of **vertices** (nodes) connected by **edges**, with each node having a direction associated with it.



Directed Graph

In an **undirected graph** the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tells us that the graph is undirected.



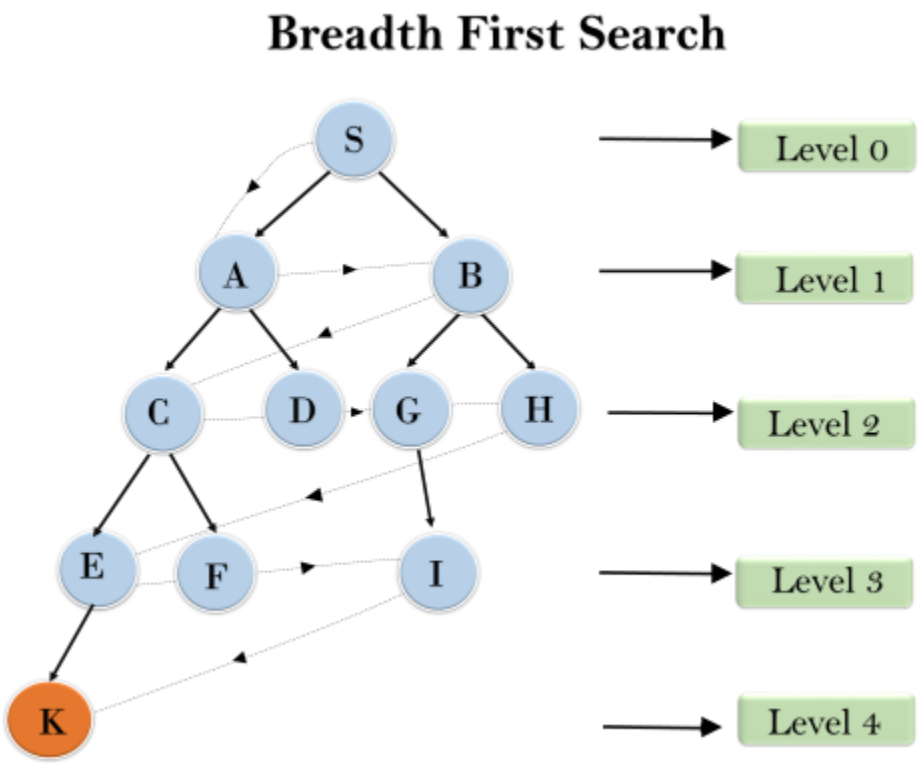
```
void addEdge(vector<int> adj[], int u, int v)
{
   adj[u].push_back(v);
   adj[v].push_back(u);
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(vector<int> adj[], int V)
{
   for (int v = 0; v < V; ++v)
   {
      cout << "\n Adjacency list of vertex "
           << v << "\n head ";
      for (auto x : adj[v])
         cout << "-> " << x;
      printf("\n");
   }
}
```

Task-1-2

- ✔ **Build the driver code of the upper code and print the adjacent list of each vertex**
- ✔ **Create a Templet type of Graph class through doubly linked list which have above functionality**

Breadth First Search

BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.




```
S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K
```

```
Graph::Graph(int V)
{
   this->V = V;
   adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
   // Mark all the vertices as not visited
   bool *visited = new bool[V];
```

```
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}
```

Task:03 By using the above program add at least 10 edges and traverse it from a given source.

Dry Run of Depth First Search

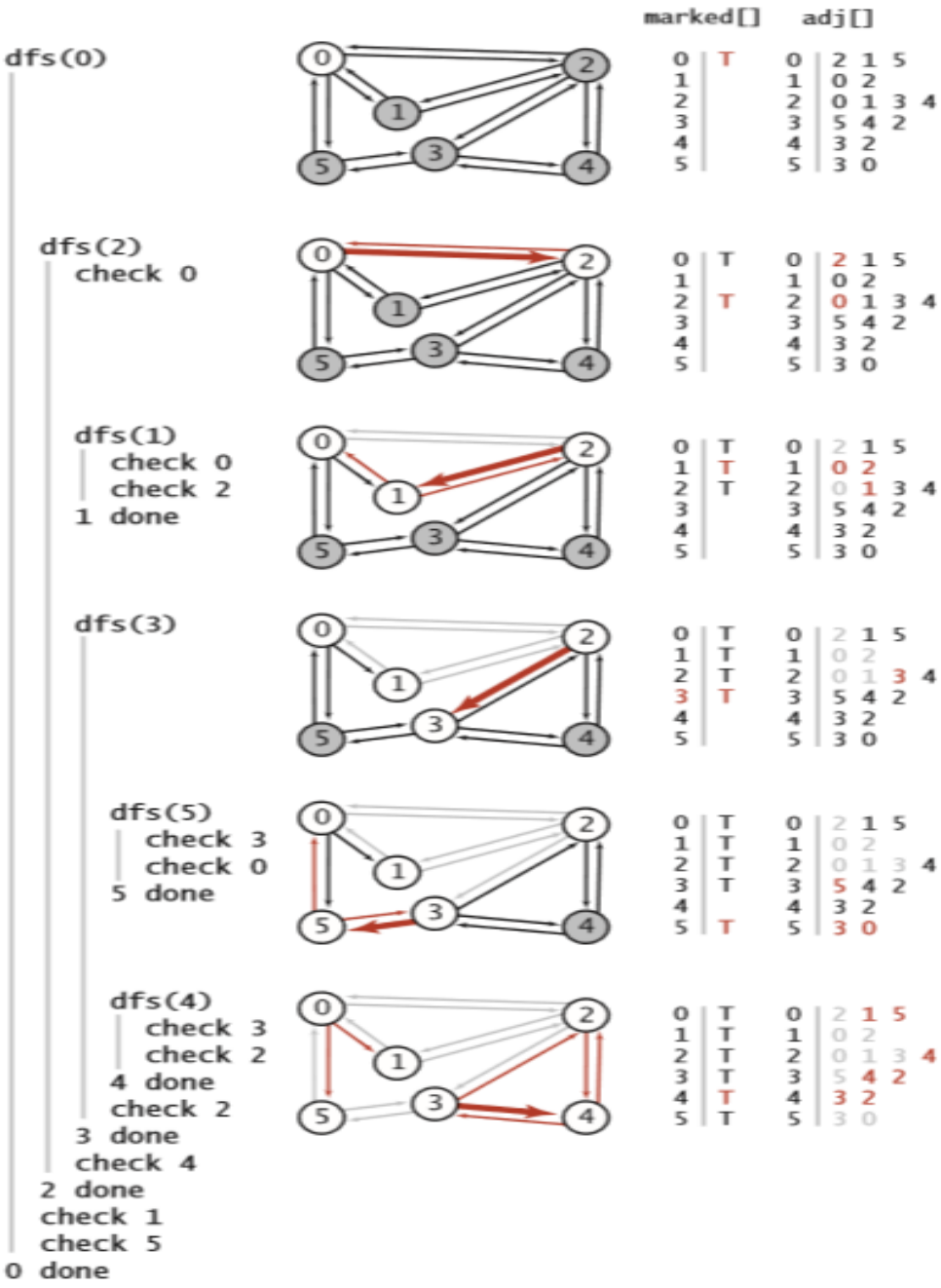


Figure-1

Task-4

- ✔ **Dry Run the above task with changing the configuration of figure-1 starting with Node 3**
- ✔ **Implement the above dry run code**

Summary Discussion

Graph Data structure

A graph is a data structure that consists of the following two components:

- ✔ A finite set of vertices also called as nodes.
- ✔ A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Representation of Graph

The following two are the most commonly used representations of a graph.

- ✔ Adjacency Matrix
- ✔ Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for an undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

Depth-first search

We often learn properties of a graph by systematically examining each of its vertices and each of its edges. Determining some simple graph properties—for example, computing the degrees of all the vertices—is easy if we just examine each edge (in any order whatever). But many other graph properties are related to paths, so a natural way to learn them is to move from vertex to vertex along the graph's edges. Nearly all of the graph-processing algorithms that we consider use this same basic abstract model, albeit with various different strategies.

The simplest is a classic method that we now consider. Searching in a maze. It is instructive to think about the process of searching through a graph in terms of an equivalent problem that has a long and distinguished history—finding our way through a maze that consists of passages

connected by intersections. Some mazes can be handled with a simple rule, but most mazes require a more sophisticated strategy. Using the terminology maze instead of graph, passage instead of edge, and intersection instead of vertex is making mere semantic distinctions, but, for the moment, doing so will help to give us an intuitive feel for the problem.

Lab12: Graph Data Structures -Basic functions and some utility functions.		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		