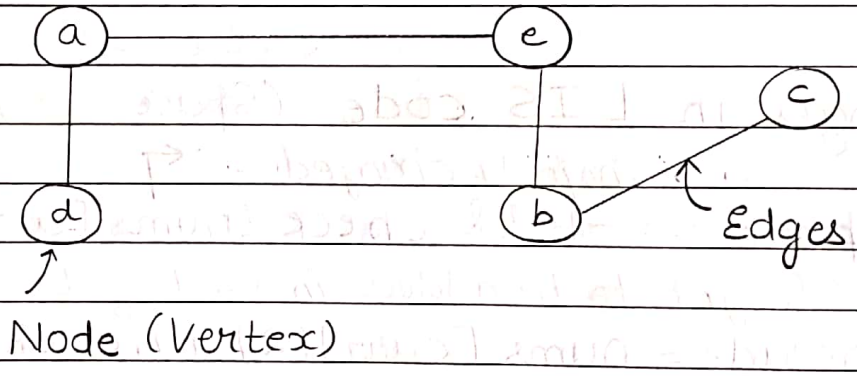
17/06/2023

Graphs

It is a data structure which is formed by combination of nodes and edges

Node → Stores data

Edge → Connect different nodes



Vertex / nodes → a, b, c, d, e

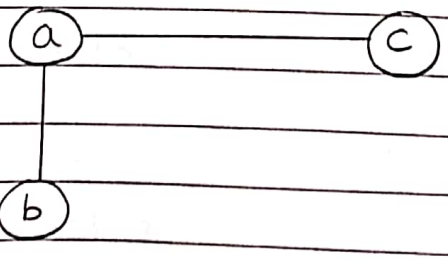
Edges → ae, ad, be, bc

Applications

Bus system (route deciding), google maps, facebook etc. ↳ Dijkstra algorithm

Types of graph

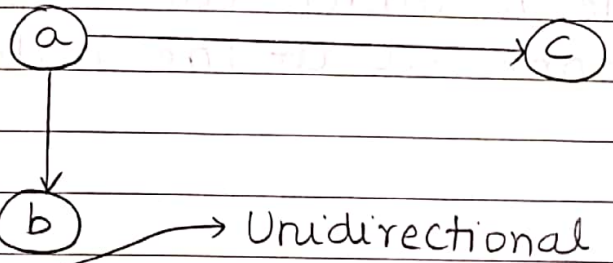
1) Undirected graph



Edges → ac, ca, ab, ba

↳ Bidirectional edges

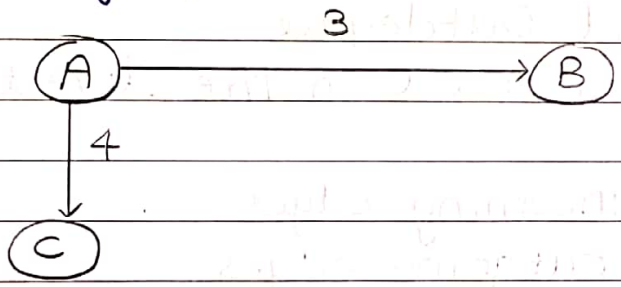
2) Directed graph



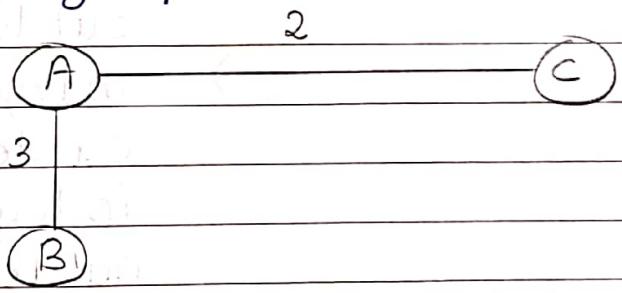
Edges → ac, ab

Here ca and ba are not considered as edges.

3) Weighted graph



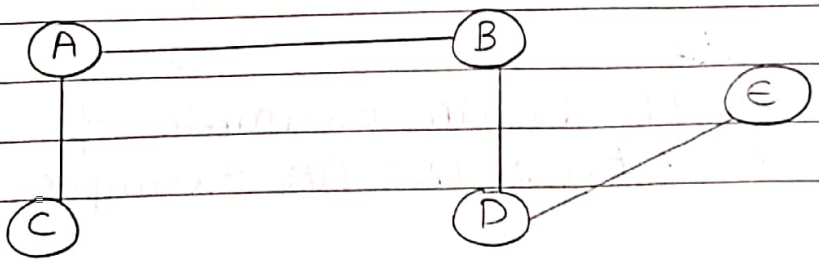
The above graph is an example of weighted directed graph



The above graph is an example of weighted undirected graph.

Note → If the weights are not given, then we consider 1 as the weight for each edge.

Degree

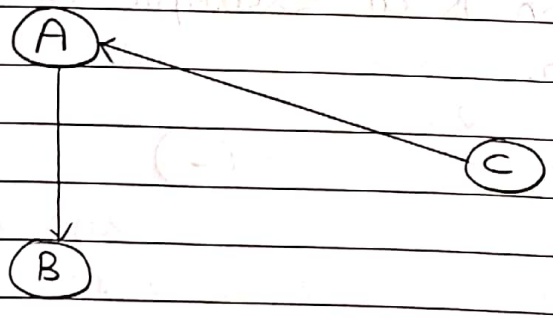


Degree of node in undirected graph is the no. of edges connected to the node.

degree (A) = 2
degree (B) = 2
degree (C) = 1
degree (D) = 2
degree (E) = 1

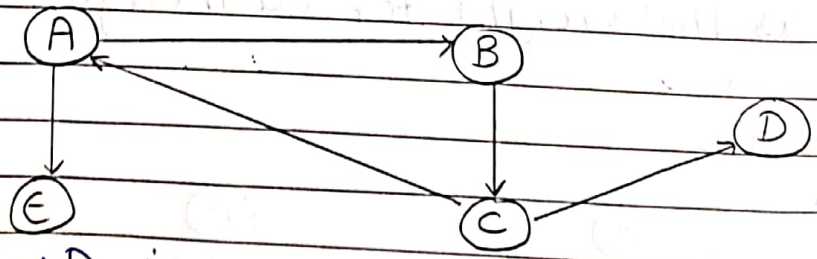
Indegree and Outdegree
This concept is used in the directed graph.

Indegree → incoming edges
Outdegree → outgoing edges



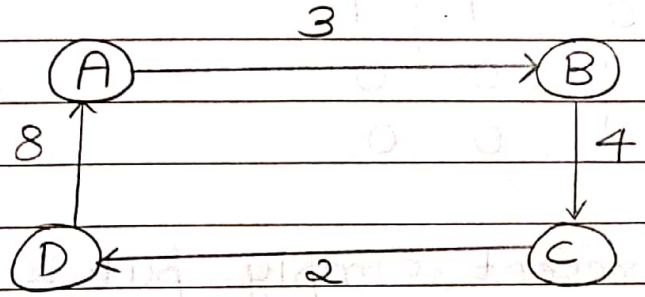
indegree (A) = 1
outdegree (A) = 1
indegree (B) = 1
outdegree (B) = 0
indegree (C) = 0
outdegree (C) = 1

Path
Nodes can not be repeated in a path. Path is defined as sequence of nodes.



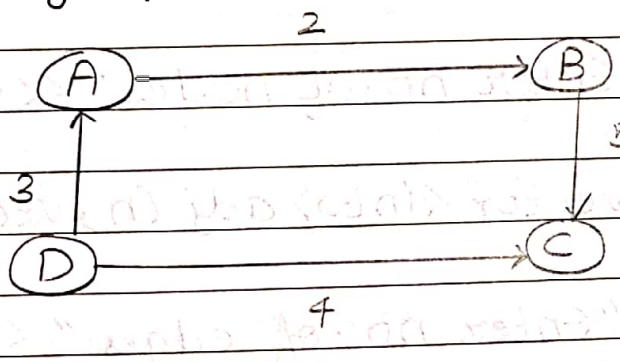
A → B → C → D is an example of path
A → B → C → A is not an example of path
↳ repeating

Cyclic graph



The above graph is an example of weighted cyclic directed graph.

Acyclic graph

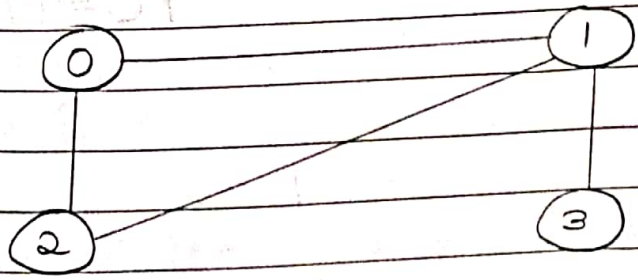


The above graph is an example of weighted acyclic directed graph

Implementation of graphs

1) Adjacency matrix → 2D array
2) Adjacency list → map / list / vector / set

* Adjacency matrix



No. of nodes = 4

Simply make 4 X 4 matrix

	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	0
3	0	1	0	0

If edge is present, simply put 1 else put 0. We simply require the edge list in i/p.

Code

```
int main () {
    int n;
    cout << "Enter no. of nodes" << endl;
    cin >> n;
    vector <vector <int>> adj (n, vector <int>(n,0));
    int e;
    cout << "Enter no. of edges" << endl;
    cin >> e;

    for (int i = 0; i < e ; i++) {
        int u, v;
        cin >> u >> v;
        // mark 1
        adj [u][v] = 1;
    }
    return 0;
}
```

SC = O(V²)

Output

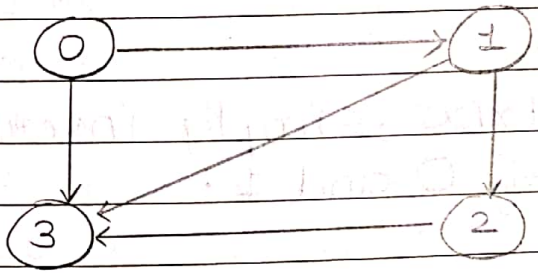
3 → nodes (number)
6 → edges (number)
0 1
1 0

1	2
2	1
0	2
2	0

Print adjacency matrix →

0	1	1
1	0	1
1	1	0

* Adjacency list



SC = O(V+E) → average
SC = O(V²) → worst

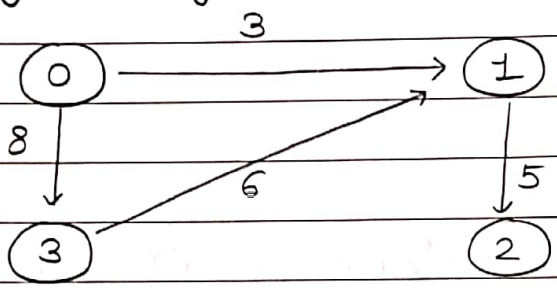
List of edges
0 - 1
1 - 3
1 - 2
2 - 3
0 - 3

Adjacency list :-

0 → {1, 3}
1 → {2, 3}
2 → {3}
3 → { }

Note: Both of the implementation can be used for undirected and directed graph.

Weighted graph



node ← → distance

0 → {(1,3), (3,8)}
1 → {(2,5)}
2 → {}
3 → {(1,6)}

In adjacency matrix, simply insert the weight instead of 0 and 1.

Code

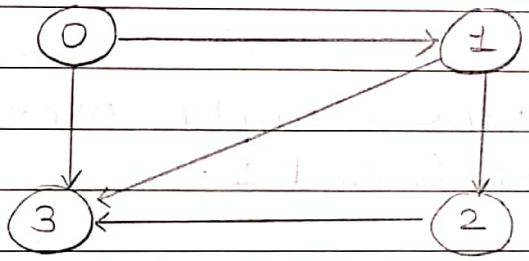
```
Class Graph {
    public :
        unordered_map<int, list<int>> adjList;
        void addEdge (int u, int v, bool
        direction){
        // direction = 0 → undirected
        // direction = 1 → directed
        // Create edge from u to v
        adjList[u].push_back(v);
        // add edge from v to u if undirected
        if (direction == 0)
            adjList[v].push_back(u);
    }
    void printAdjList() {
```

1	2
2	1
0	2
2	0

Print adjacency matrix →

0	1	1
1	0	1
1	1	0

* Adjacency list



SC = O(V+E) → average
SC = O(V²) → worst

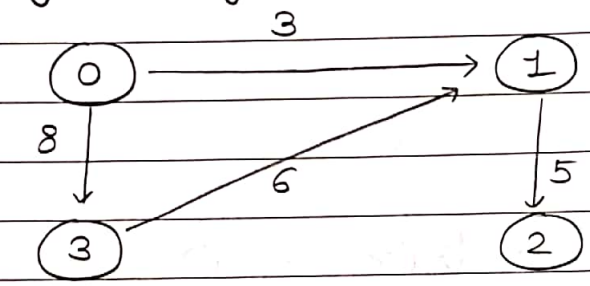
List of edges
0 - 1
1 - 3
1 - 2
2 - 3
0 - 3

Adjacency list :-

0 → {1, 3}
1 → {2, 3}
2 → {3}
3 → { }

Note → Both of the implementation can be used for undirected and directed graph.

Weighted graph



node ← → distance

0 → {(1,3), (3,8)}
1 → {(2,5)}
2 → {}
3 → {(1,6)}

In adjacency matrix, simply insert the weight instead of 0 and 1.

Code

```
class Graph {
    public:
        unordered_map<int, list<int>> adjList;
        void addEdge(int u, int v, bool
        direction){
        // direction = 0 → undirected
        // direction = 1 → directed
        // Create edge from u to v
        adjList[u].push_back(v);
        // add edge from v to u if undirected
        if(direction == 0)
            adjList[v].push_back(u);
    }
    void printAdjList(){
```

```
for (auto node : adjList) {
    cout << node.first << " → ";
    for (auto neighbours : node.second) {
        cout << neighbours << ", ";
    }
    cout << endl;
  }
}
};
```

Note → In case of weighted graph, changes in code

1) unordered_map <int, list <pair <int, int >>> adjList; Pass weight in addEdge function.
2) adjList [u].push_back ({v, weight});
   adjList [v].push_back ({u, weight});

3) In print function, inside inner for loop.

```
cout << "(" << neighbours.first << ", " << neighbours.second << ")" << ", ";
```

Note → In adjacency matrix, instead of marking with 1, simply mark with weight.

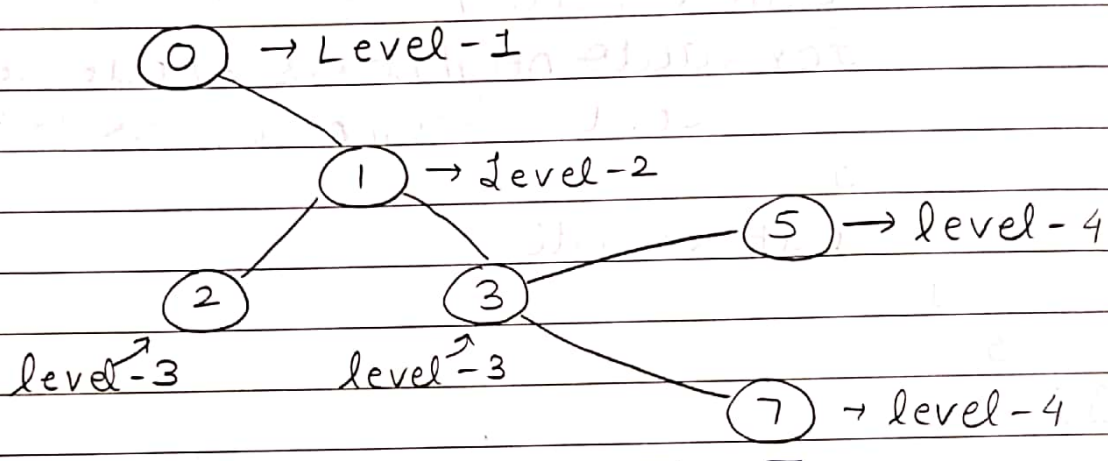
Generic graph
It can be created with the help of template.

```
template <typename T>
```

Now replace int with T in the code.

Graph <char> g; → Creation of graph

Breadth first search → level order traversal



O/p → 0 1 2 3 5 7

1) Simply create edge list and then create the adjacency list.

0 → {1}
1 → {2, 3, 0}
2 → {1}
3 → {1, 5, 7}

5 → {3}
7 → {3}

2) Make a queue and push the source node in queue.

queue → {0}
Find front node. Pop it & insert the neighbours. Also we need to maintain a visited data structure.

0 → F T
1 → F T
2 → F T
3 → F T
5 → F T
7 → F T

We have to push the node in queue if it is not visited.

1) o/p → 0

queue → {1}

2) o/p → 0 1
queue → {2, 3} (0 not inserted)

3) o/p → 0 1 2
queue → {3} (1 not inserted)

4) o/p → 0 1 2 3
queue → {5, 7} (1 not inserted)

5) o/p → 0 1 2 3 5
queue → {7} (3 not inserted)

6) o/p → 0 1 2 3 5 7
queue → { } → empty & hence stop

Code

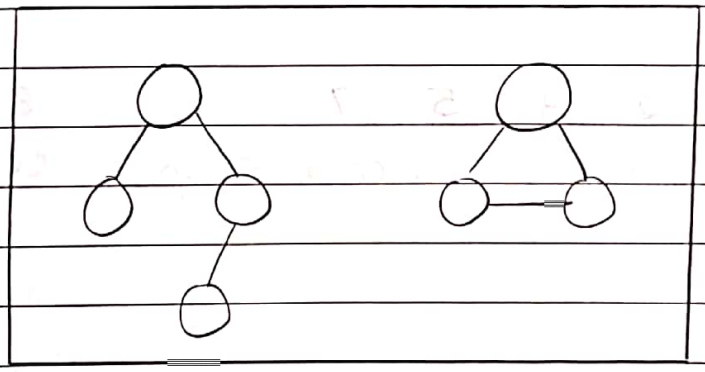
```
void bfs (int src) {
    queue <int> q;
    unordered_map <int, bool> visited;
    q.push (src);
    visited [src] = true;
    while (! q.empty ()){
        // Front node
        int frontNode = q.front ();
```

```
q.pop();

cout<<frontNode;
//insert neighbours
for(auto neighbours:adjList[frontNode]){
        //If not visited, then insert
        if(!visited[neighbours]){
                q.push(neighbours);
                visited[neighbours]=true;
        }
    }
  }
}
```

Time complexity = $O(V+E)$

Components of graph

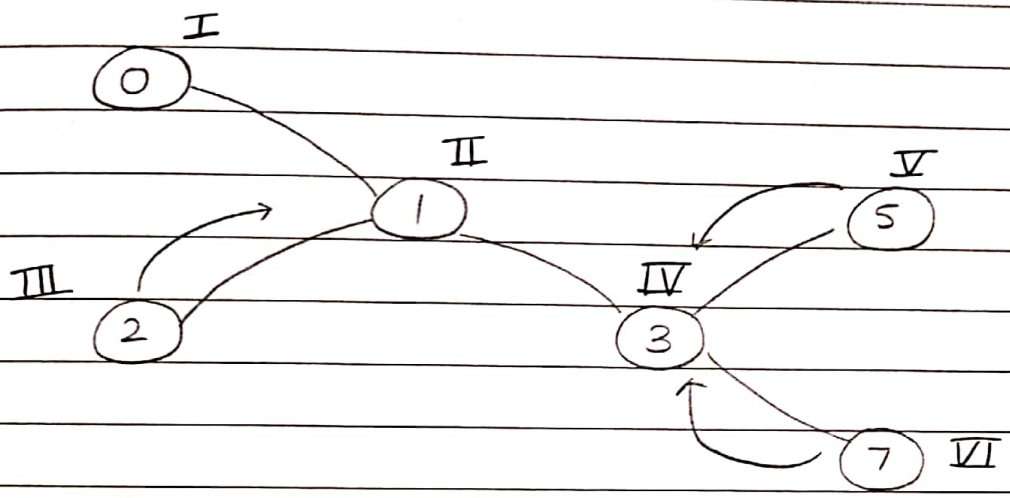


G1

As no. of components >1, this is known as disconnected graph. To handle this run a for loop on number of nodes.

```
for(int i=0 ; i<=(7); i++){   // 7 : no. of nodes
        if(!visited[i])
                bfs(i ,visited);
}
```

Depth first Search



0 1 2 3 5 7

Print source node and then recursive call for neighbours. (if not visited)

Code

```
void dfs (int src, unordered_map <int, bool> &
visited) {
    cout << src << " ";
    visited [src] = true;
    for (auto neighbour : adjList [src]) {
        if (! visited [neighbour])
            dfs (neighbour, visited);
    }
}
}
```

Why no base case required?
Recursive call will go only if not visited but at some point of time all nodes will be visited.

TC = O (V+E)