CS201
Data Structures and Algorithms
Revision Session 9

graph

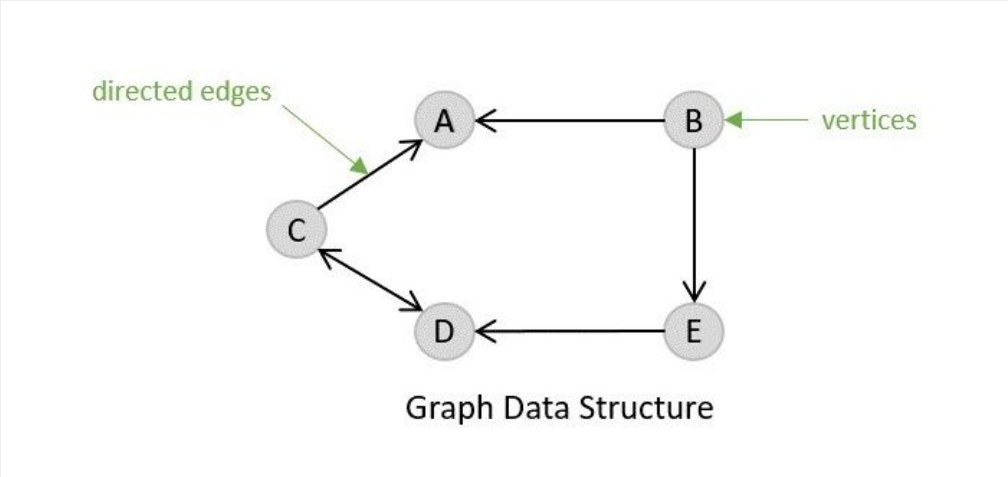
Graph

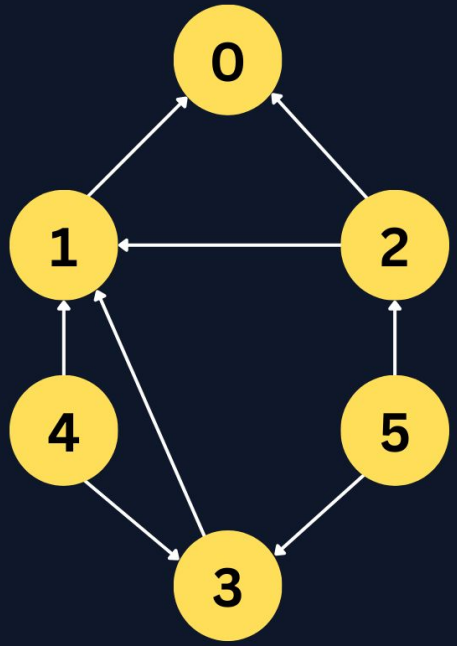
basics:
definition
implementation (adjacency matrix, adjacency list)

operations:
add edge (for adjacency matrix and adjacency list)

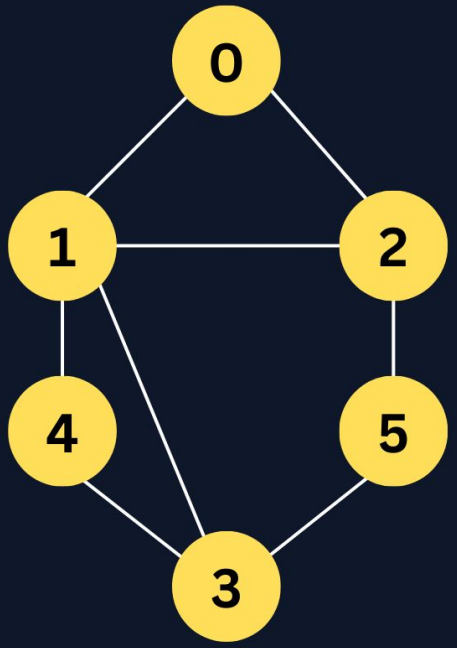
definition

- G = (V, E)
- one or more vertices and lines (edges) connecting those vertices
- might not have any edges, but must have at least one vertex
- the edges can be directed or undirected
- the edges can be weighted or unweighted

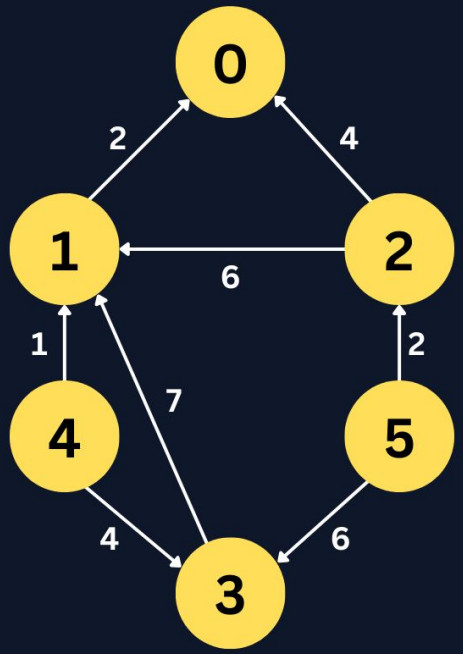




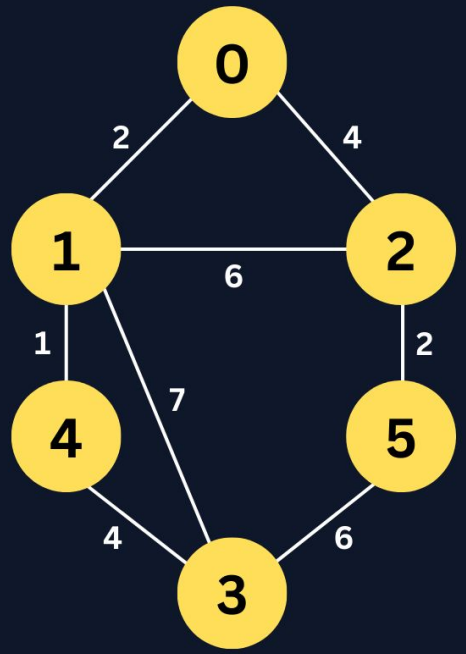
Directed Graph



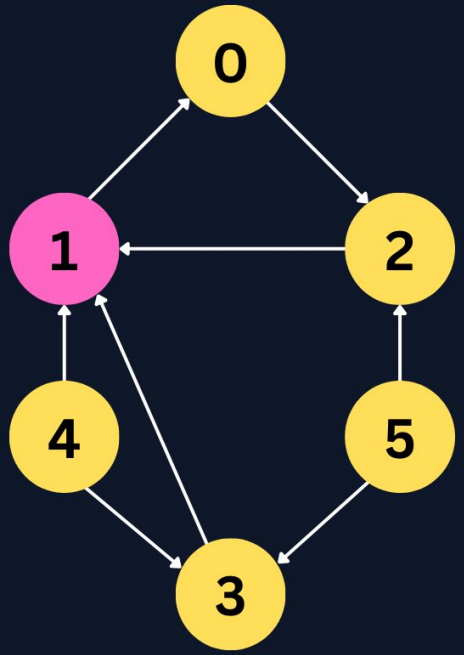
Undirected Graph

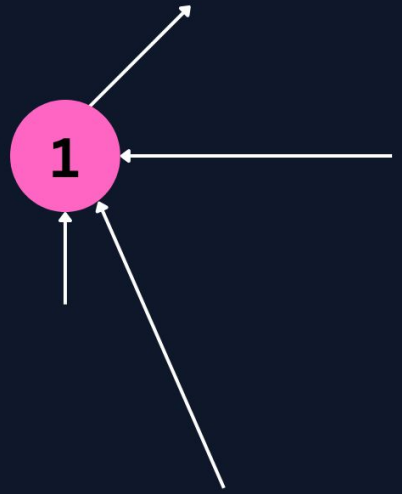


Weighted and Directed Graph



Weighted and Undirected Graph





Indegree of 1 = 3

Outdegree of 1 = 1

representing the vertices and edges

- *adjacency matrix representation*
- *adjacency list representation*

implementation

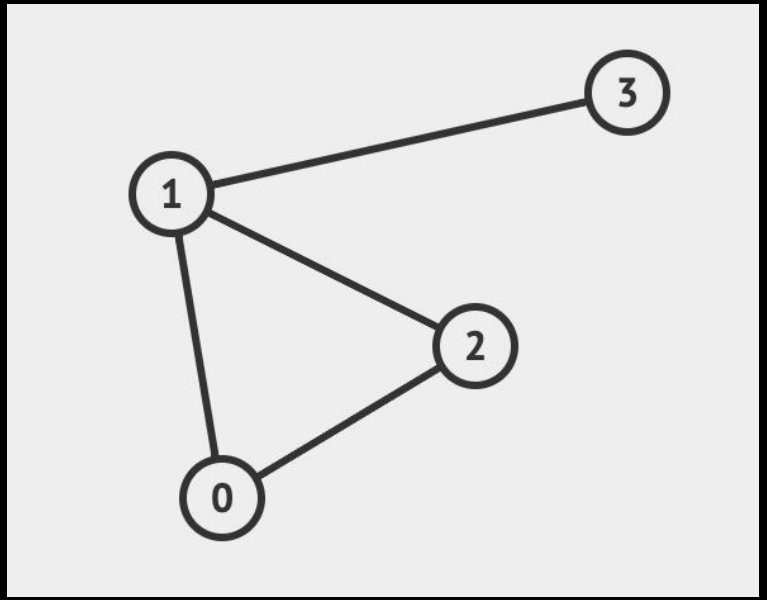
adjacency matrix representation

The **undirected** array is symmetric: if [0][1] is 1 [1][0] also has to be 1.

V = 4, E = 4

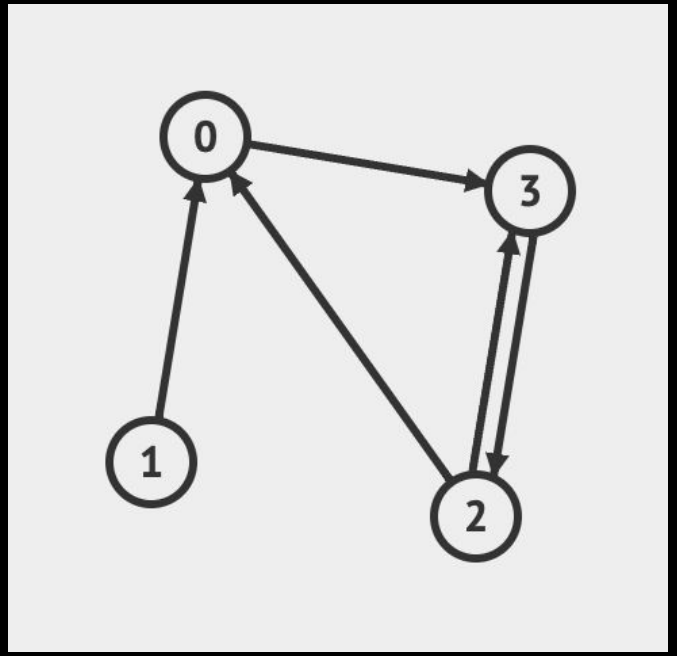
4 edges means there need to be 8 1s in the array.

For each row, we write all vertices it is connected to.



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

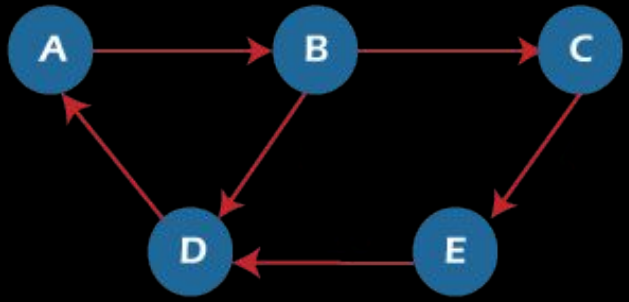
The **directed** array is NOT symmetric: 5 edges mean 5 ones.
For each row, we write the outgoing edges.
V = 4, E = 5

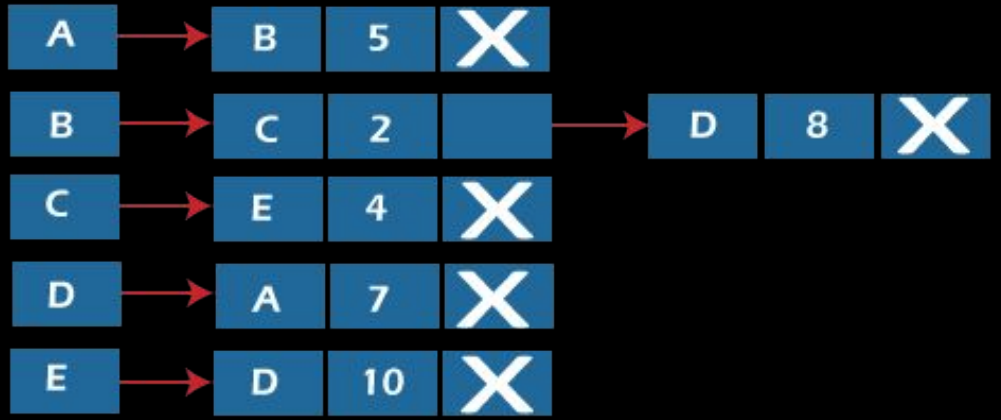


	0	1	2	3
0				1
1	1			
2	1			1
3			1	

```java
public class GraphMatrix {
    2 usages
    int[][] edges;
    1 usage
    int vertexCount;

    1 usage  new *
    public GraphMatrix(int vertexCount){
        this.vertexCount = vertexCount;
        edges = new int[vertexCount][vertexCount];
        for (int i = 0; i < vertexCount; i++){
            for (int j = 0; j < vertexCount; j++){
                edges[i][j] = 0;
            }
        }
    }
}
```

adjacency list representation


A
B
C
D
E


A
B
5
B
C
2
D
8
C
E
4
D
A
7
E
D
10

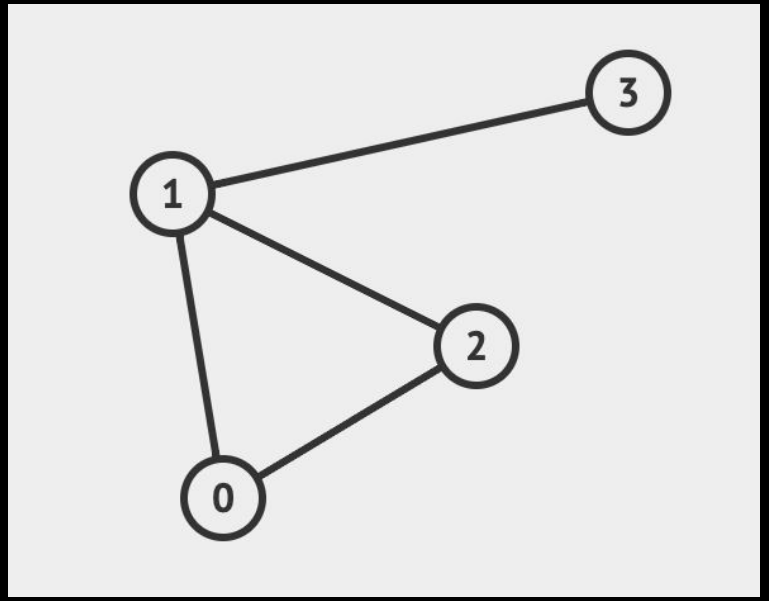
```java
public class Edge {
    1 usage
    int from;
    1 usage
    int to;
    1 usage
    int weight;
    2 usages
    Edge next;

    1 usage  new *
    public Edge(int from, int to, int weight){
        this.from = from;
        this.to = to;
        this.weight = weight;
        this.next = null;
    }
}
```

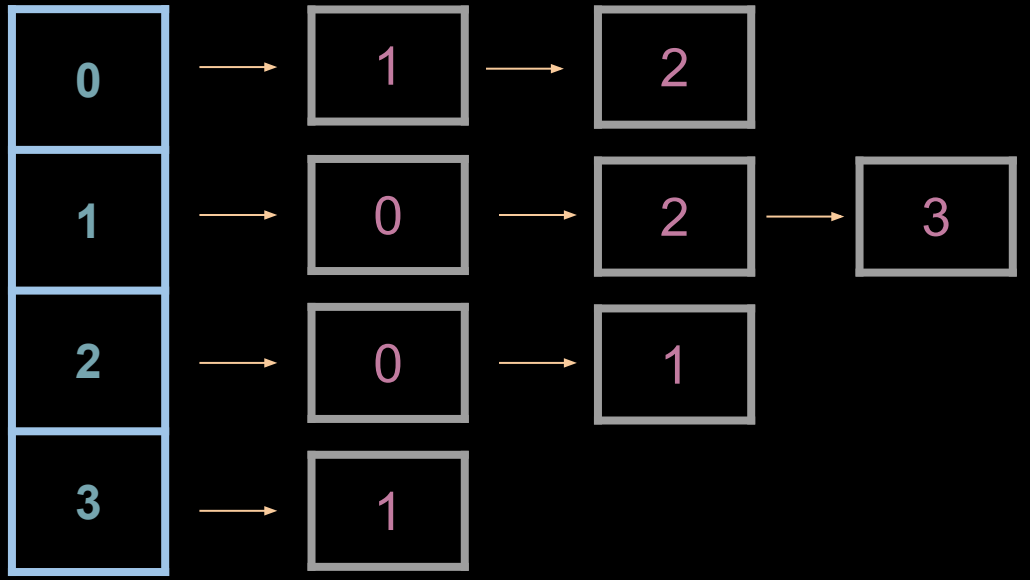
```java
public class EdgeList {
    3 usages
    Edge head;
    3 usages
    Edge tail;

    1 usage  new *
    public EdgeList(){
        head = null;
        tail = null;
    }
    1 usage  new *
    public void insert(Edge newEdge){
        if (head == null){
            head = newEdge;
        } else{
            tail.next = newEdge;
        }
        tail = newEdge;
    }
}
```

```java
public class GraphList {
    2 usages
    EdgeList[] edges;
    1 usage
    int vertexCount;

    1 usage  new *
    public GraphList(int vertexCount){
        this.vertexCount = vertexCount;
        edges = new EdgeList[vertexCount];
        for(int i = 0; i < vertexCount; i++){
            edges[i] = new EdgeList();
        }
    }
}
```

Undirected graphs

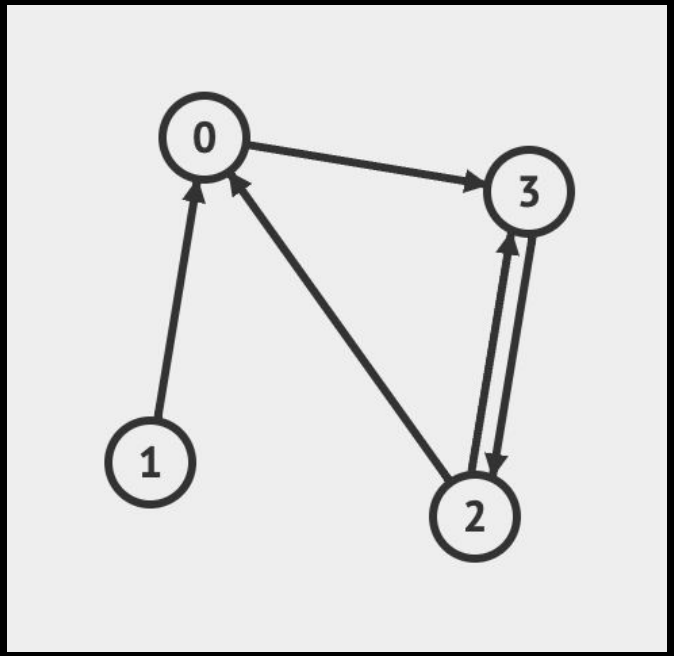
An array of linked lists: for each node, every connection it has is listed.

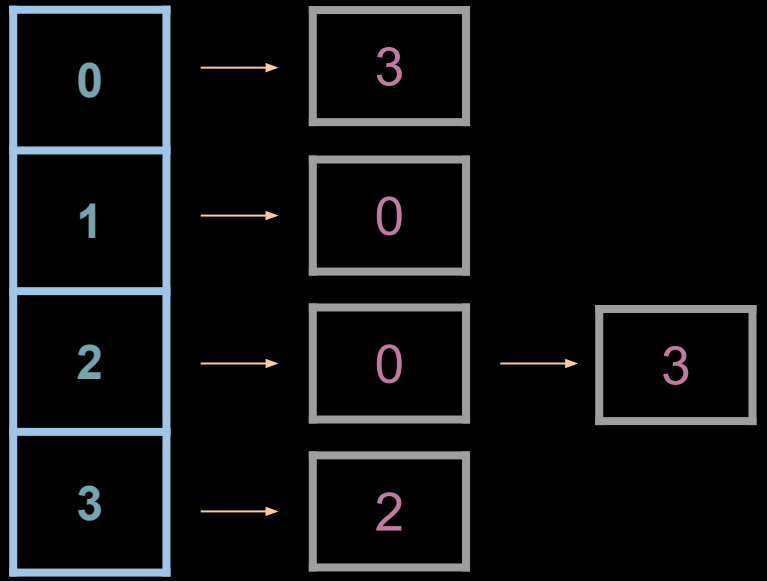




Directed graph

For each vertex, the list contains the outgoing edges. If a vertex has 2 ingoing edges, it will appear 2 times in the lists. If it has none, it is not on any list.





add edge (matrix)

add edge from 3 to 2 ⟶ [3][2]

```java
public void addEdge(int from, int to){
    edges[from][to] = 1;
}
1 usage  new *
public void addEdge(int from, int to, int weight){
    edges[from][to] = weight;
}
```

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

add edge (list)

```java
public void insert(int to, int from, int weight){
    Edge edge = new Edge(from, to, weight);
    edges[from].insert(edge);
}
```

- create the edge object
- edges[from] is an EdgeList
- use the insert method from EdgeList class