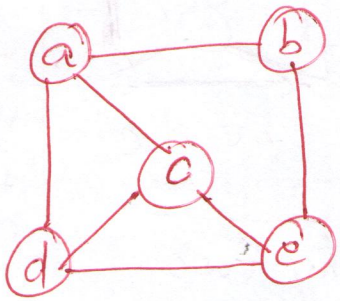
Graphs.

Graphs → Definitions
→ Examples
→ The Graph ADT

What is a Graph?



$V = \{a, b, c, d, e\}$

$E = \{(a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e)\}$

A graph $G = (V, E)$ is composed of:

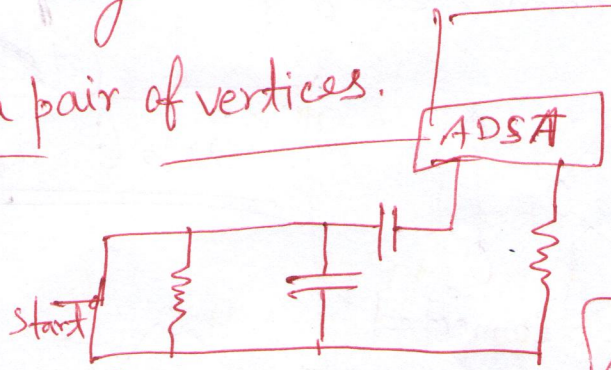
V: set of vertices

E: set of edges connecting the vertices in V.

An edge $e = (u, v)$ is a pair of vertices.

Applications

Electronic circuits



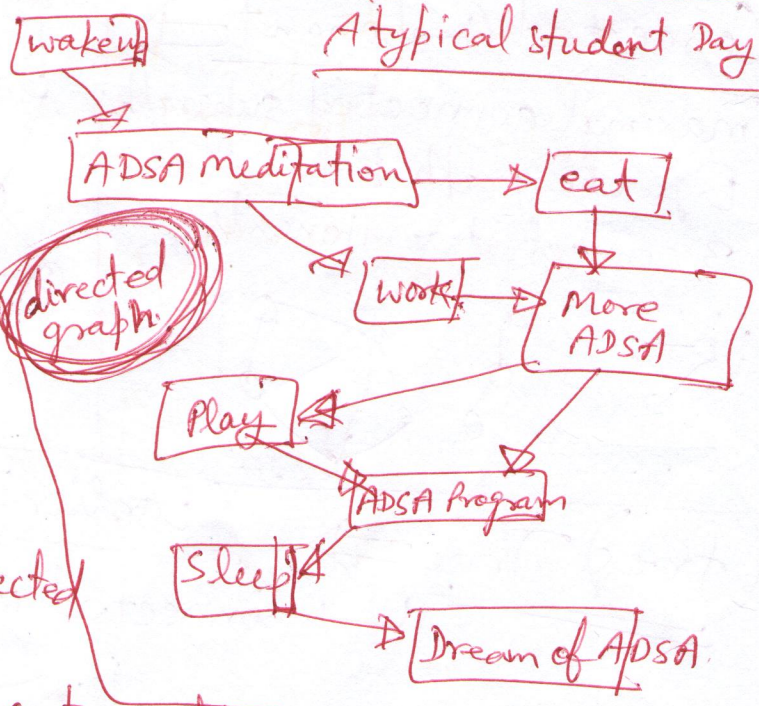
⇒ Find the path of least resistance to ADSA.

Networks (roads, flights)

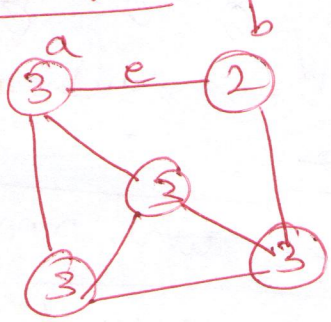
Communication networks

A Typical Example

A typical student Day



Graph Terminology.



directed graph.

Adjacent vertices: vertices connected by an edge.

Degree (of a vertex): # of adjacent vertices.

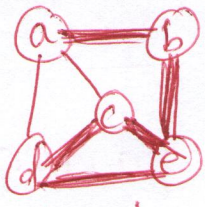
⇒ $e = (a, b)$ is incident to vertices a & b.

What is the sum of the degrees of all vertices?

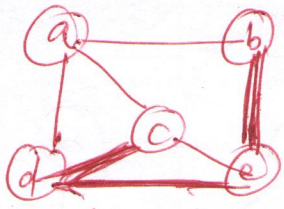
↳ Twice the number of edges.

↳ since adjacent vertices each count the adjoining edge, it will be counted twice.

(2) **Path**: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



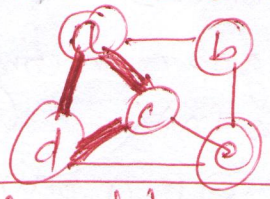
a b e d c e



b e d c

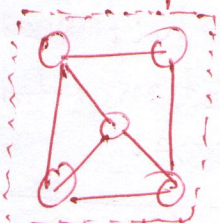
Simple Path: No repeated vertices.

Cycle Path: simple path, except that the last vertex is the same as the ~~first~~ first vertex.

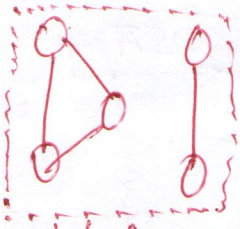


a c d a.

Connected Graph: any two vertices are connected by some path.



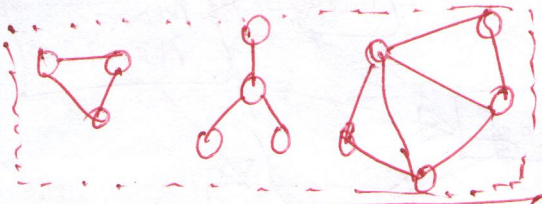
Connected

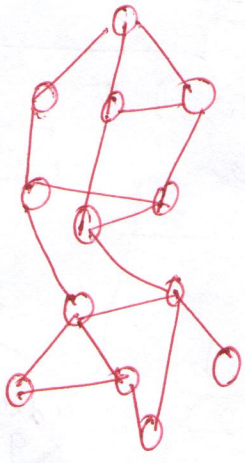


Not Connected

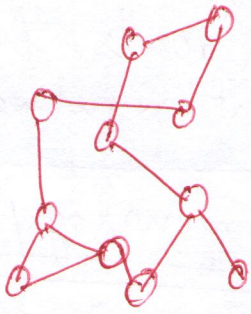
Sub Graph: subset of vertices and edges forming a graph.

Connected Component ⟹ maximal connected subgraph, E.g. the graph below has 3 connected components.





G



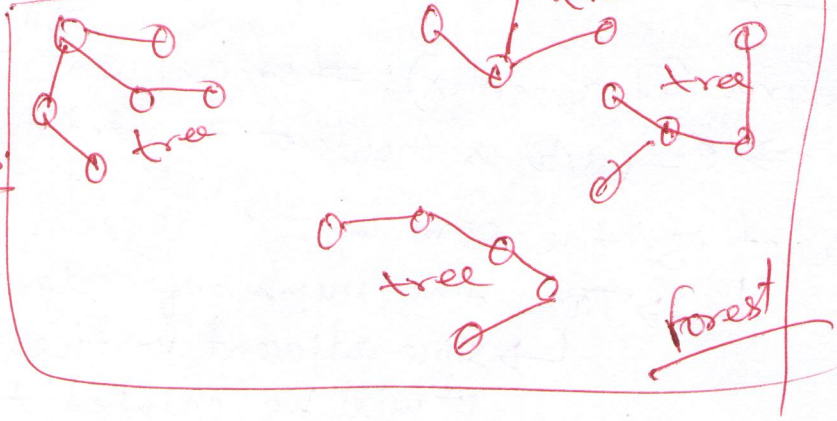
subGraph of G.

(free) Tree ⟹ (Earlier tree was rooted tree)

↳ Connected Graph without cycles.

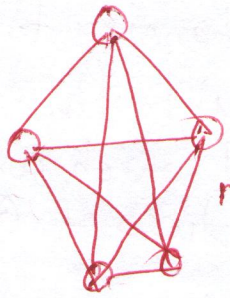
#forest

↳ collection of trees.



Connectivity

Let n = #vertices, and m = #edges



$n=5$

$m = \frac{5 \times 4}{2} = 10$

Complete Graph: one in which all pairs of vertices are adjacent (i.e., every pair of vertices has an edge).

How many edges does a complete graph have?

There are $n(n-1)/2$ pairs of vertices and so $m = n(n-1)/2$

For a tree $m = n-1$, where n = #vertices and m = #edges.

$\Rightarrow$ A tree on n vertices has $n-1$ edges.

Proof by induction $\Rightarrow$

Base Case: $n=2$ 

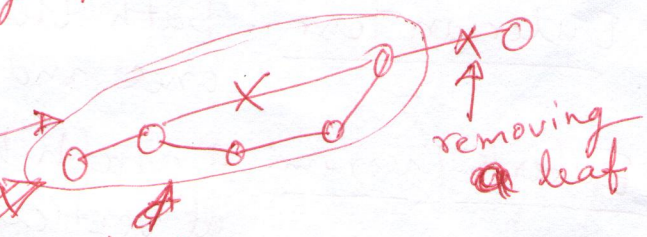
no. of edges $= n-1 = 1$

Induction Hypothesis: statement is true for all $n \leq K$

Induction step: Given a ~~graph~~ (tree) on $K+1$ vertices

A leaf is a vertex of degree 1.

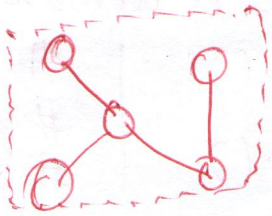
Every tree has at least 1 leaf.



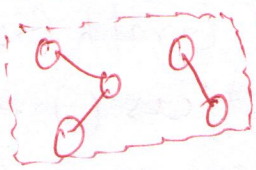
still connected tree after removing the leaf. This is a tree on K vertices & has $K-1$ edges (by induction hypothesis).

If $m < n-1$, then G is not connected.

(m: # edges, n: # Elements (vertices))



$n=5$, $m=4$



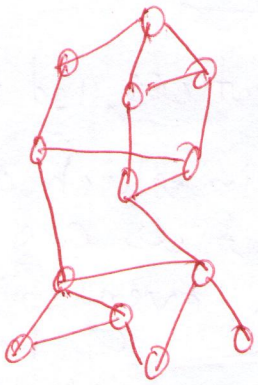
$n=5$, $m=3$

If a graph has $n-K$ number of edges then it can have K or more number of connected components.

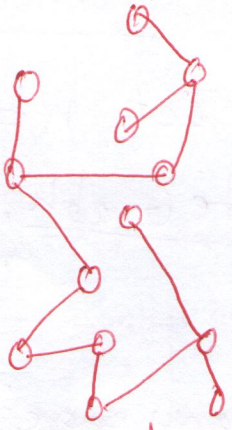
(K: without cycle; more number: with cycle)

⑤ Spanning Tree ⇒

A spanning tree of G is a subgraph which is a tree and which contains all vertices of G.



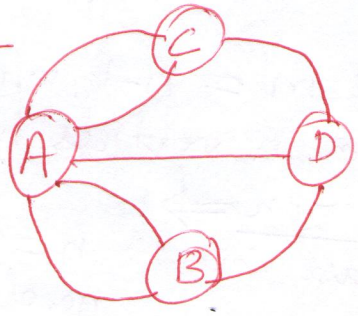
G



spanning tree of G.

Failure on any edge leads to disconnects system.
↳ Least fault tolerance.

Graph Model (with parallel edges)



Suppose you are a postman, and you do not want to retrace your steps.

In 1736, Euler proved that this is not possible.

Eulerian Tour : path that traverses every edge exactly once and returns to the first vertex.

Euler's Theorem: A graph has a Eulerian tour if and only if all vertices have even degree.

The Graph ADT ⇒ The graph ADT is a positional container whose positions are the vertices and the edges of the graph.

size() Return the number of vertices + number of edges of G.

# isEmpty()	# positions()
# elements()	# swap()
	# replaceElement()

Notation: Graph G; Vertices V, W; Edge e; Object o

⇒ numVertices() — Return the number of vertices of G.

⇒ numEdges() — Return the number of edges of G.

⇒ vertices() — Return an enumeration of the vertices of G.
↳ To iterate across all vertices

⇒ edges() — Return an enumeration of the edges of G.

directedEdges() – enumeration of all directed edges in G.
undirectedEdges() – enumeration of all undirected edges in G.
incidentEdges(v) – enumeration of all edges incident on v.
inIncidentEdges(v) – enumeration of all edges entering v.
outIncidentEdges(v) – enumeration of all edges leaving v.
opposite(v,e) – an end point of e distinct from v.
degree(v) – the degree of v.
inDegree(v) – the in-degree of v.
outDegree(v) – the out-degree of v.
adjacentVertices(v) – enumeration of vertices adjacent to v.
inAdjacentVertices(v) – enumeration of vertices adjacent to v along incoming edges.
outAdjacentVertices(v) – enumeration of vertices adjacent to v along outgoing edges.
areAdjacent(v,w) – whether vertices v and w are adjacent.
endVertices(e) – the end vertices of e.
origin(e) – the end vertex from which e leaves.
destination(e) – the end vertex at which e arrives.
isDirected(e) – true iff e is directed.] → In mixed graph

Update Methods →

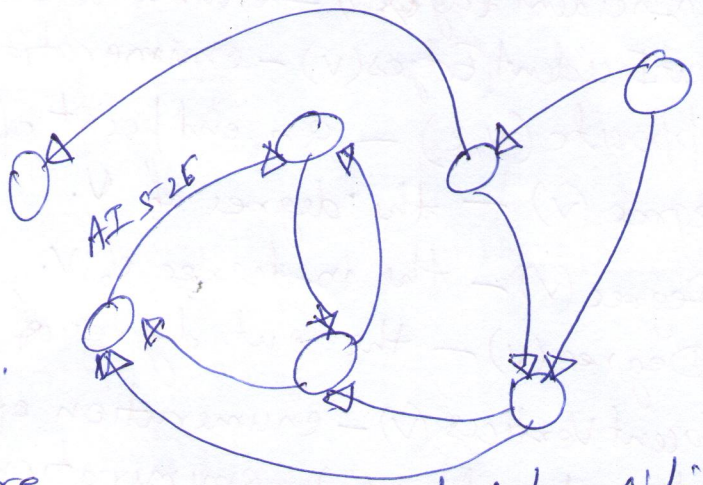
makeUndirected(e) – Set e to be an undirected edge.
reverseDirection(e) – Switch the origin and destination vertices of e.
setDirectionFrom(e,v) – sets the direction of e away from v, one of its end vertices.
setDirectionTo(e,v) – sets the direction of e toward v, one of its end vertices.
insertEdge(v,w,o) – Insert and return an undirected edge between v & w, storing o at this position.
insertDirectedEdge(v,w,o) – Insert and return a directed edge between v & w, storing o at this position.
insertVertex(o) – Insert and return a new (isolated) vertex storing o at this position.

⑥ # Data Structures for Graphs

Edge List Adjacency list Adjacency matrix.

Edge List ⇒

We store the vertices and the edges into two containers, and each edge object has references to the vertices it connects.



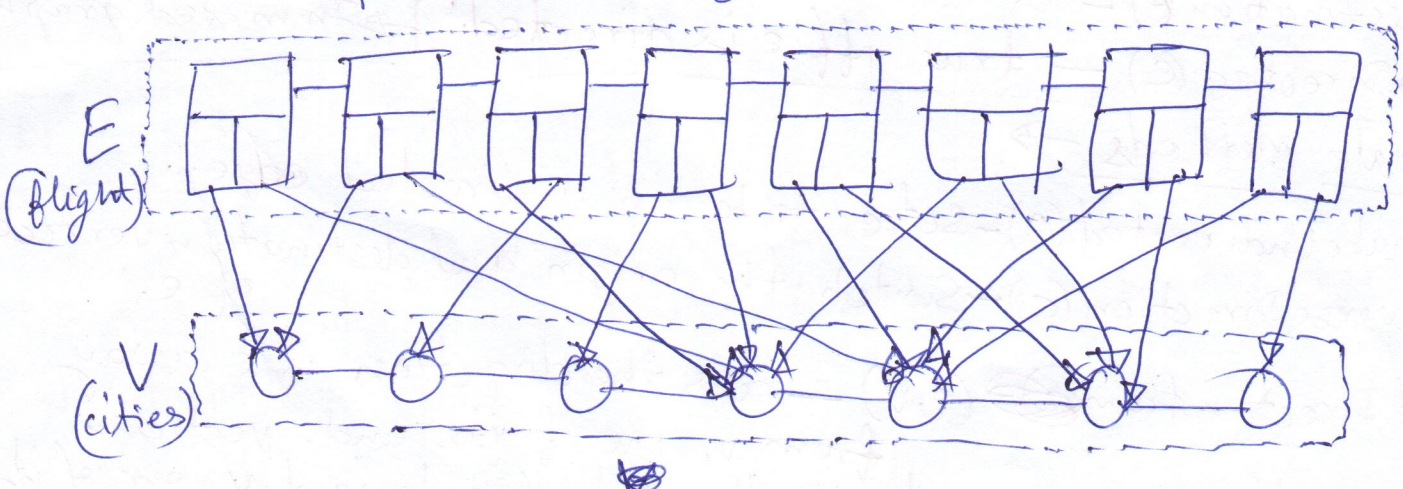
Flights b/w cities.

Vertex → City
Edge → Flight

⇒ The edge list structure simply stores the vertices and the edges in two unsorted sequences.

⇒ Easy to implement.

⇒ Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence.



finding the adjecent vertices is costly.

Performance

size, isEmpty, replaceElement, swap, numVertices, numEdges → O(1)

vertices → O(n), edges, directEdges, undirectEdges → O(m)

elements, positions → O(n+m), endVertices, opposite, origin, destination, isDirected → O(1)

incidentEdges, adjacentVertices, areAdjacent, degree → O(m)

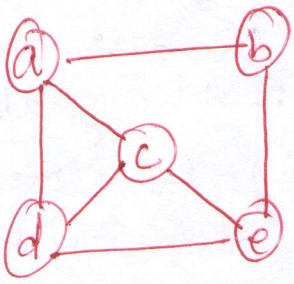
insertVertex, insertEdge, removeEdge, reverseDirection, setDirectionFrom, setDirectionTo → O(1)

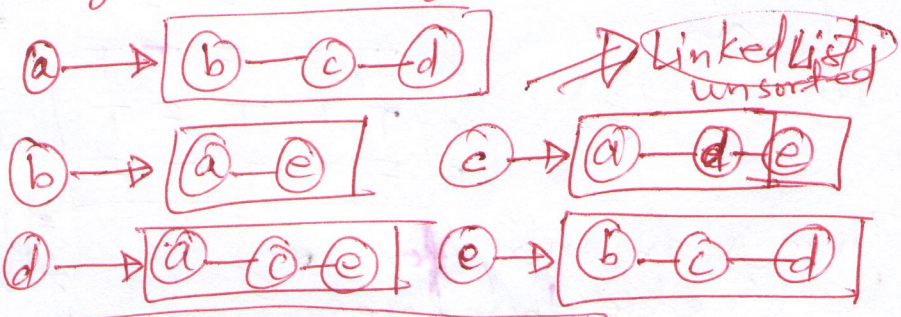
removeVertex → O(m)

Adjacency List (traditional)

adjacency list of a vertex V; sequence of vertices adjacent to V.

Represent the graph by the adjacency lists of all the vertices.



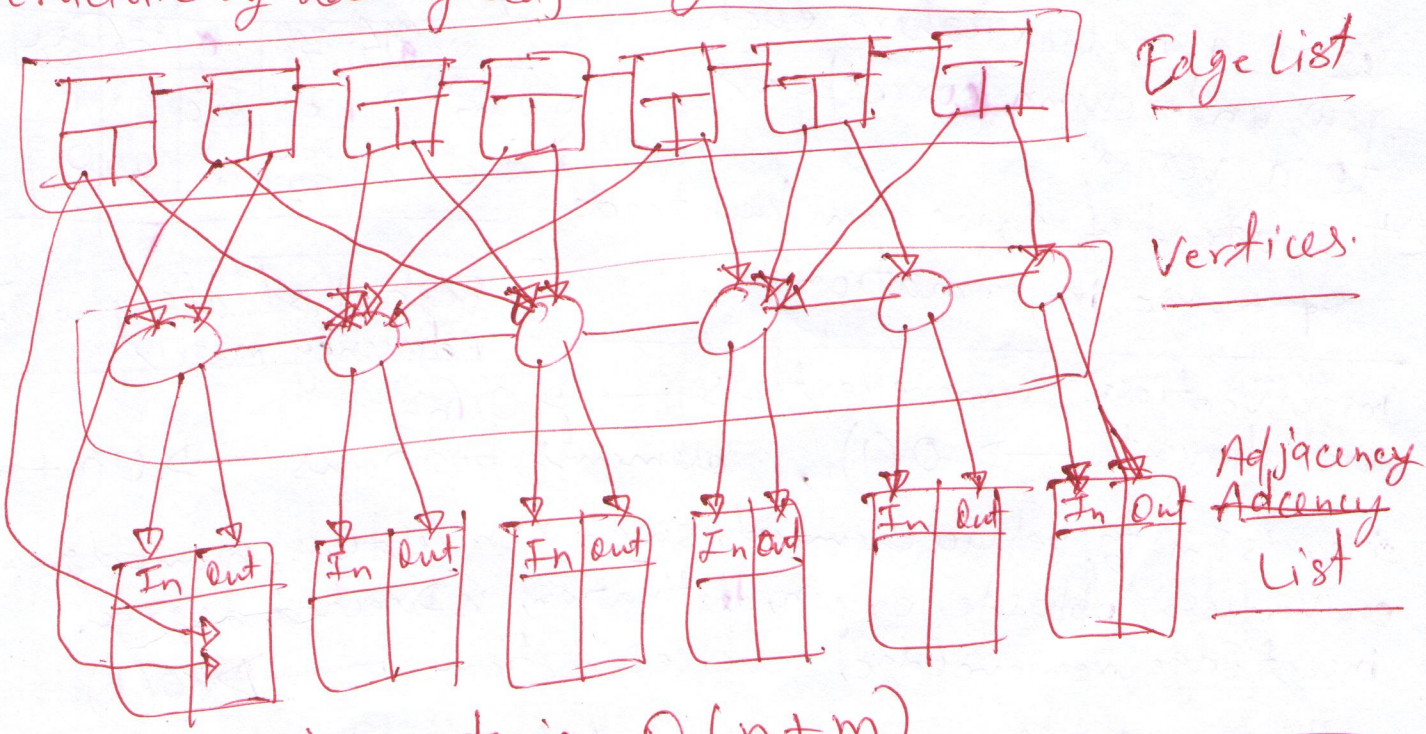


$$\text{Space} = \theta(N + \sum \deg(v)) = \theta(N+M)$$

array of size N. → 2M

Adjacency List (modern)

The adjacency list structure extends the edge list structure by adding adjacency lists to each vertex.



The space requirement is $O(n+m)$

size, isEmpty, replaceElement, swap, numVertices, numEdges, endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree, insertVertex, insertEdge, removeEdge, reverseDirection → O(1)

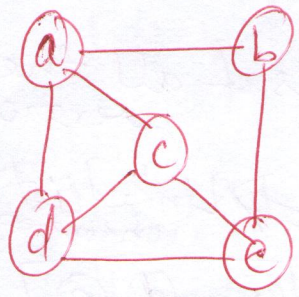
vertices → O(n), edges, directedEdges, undirectedEdges → O(m)

elements, positions → O(n+m)

incidentEdges(v), adjacentVertices(v), removeVertex(v) → O(deg(v))

areAdjacent(u,v) → O(min(deg(u), deg(v)))

Adjacency Matrix (traditional)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

Matrix M with entries for all pairs of vertices

M[i,j] = true means that there is an edge (i,j) in the graph.

M[i,j] = false means that there is no edge (i,j) in the graph.

There is an entry for every possible edge, therefore:

Space = $\Theta(N^2)$

Adjacency Matrix (modern)

The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.

Instead of 1, we can keep track of edge information.

	0	1	2	3	4	5
0	0	0	DTW 221	0	MAA 212	0
1	0	0	0	LKB	SAT 14	0
2	CXC 259	0	0	0	AIS	DEL 25
3	0	MAA 917	IN 76	0	AI6	DEL 26
4	SFO 987	0	0	IN 976	0	0
5	0	0	DEL 92	0	0	0

Edge list along with Adjacency matrix

insertVertices, removeVertices → O(n2)

areAdjacent → O(1), elements, positions → O(n+m)

size, isEmpty, replaceElement, swap, numVertices, numEdges, endVertices, opposite, origin, destination, isDirected, degree, insertEdge, removeEdge, reverseDirection → O(1).

vertices → O(n), edges, directedEdges, undirectedEdges → O(m)

incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices → O(n).