Data Structures & Algorithms

Adil M. Khan

Professor of Computer Science

Innopolis University
a.khan@innopolis.ru

Recap

- Priority Queue
- Binary Heap (Binomial Heap)
- Heapsort

Objectives

- Graphs
- Graph ADT (What is it and why we need it)
- Graph Representations (Ways to represent graphs)

Graphs

Graphs

- One of the most versatile structures used in computer programming

- *Why do we need graphs, when we already have data structures like trees and hash tables?*

Graphs

- Architectures of the previous data structures are dictated by the algorithm used on them
  - ❖ For example, a binary search tree is shaped the way it is because *it is easy to search and insert data*

Graphs

- Architectures of the previous data structures are dictated by the algorithm used on them
  - ❖ For example, a binary search tree is shaped the way it is because it is easy to search and insert data
- Graphs often have a shape dictated by a physical problem

Graphs

- Graphs are very good at describing problems, where something (or someone) can “travel” between nodes in different well-defined ways.

- E.g.
  - tourist is moving from town to town by roads,
  - current is traveling over the circuit,
  - TCP-frame can travel though internet from router to router,
  - your work activity is travelling from one task to another, etc

Graphs

- Having Graphs as a model, we can answer common questions about these journeys: For example
  - What is the shortest path for [tourist/TCP-frame/worker/…] to get from A to B in terms of [time/resistance/hops/…]? (*shortest path*)

Graphs

- Having Graphs as a model, we can answer common questions about these journeys: For example
  - How can someone visit all the points in the fastest way? (*travelling salesman problem*)

Graphs

- Having Graphs as a model, we can answer common questions about these journeys: For example
  - What is the minimal number of [roads/links/…] can remain to preserve connections between base points? (*spanning tree*)

Graphs

- Having Graphs as a model, we can answer common questions about these journeys: For example
  - How can I arrange my visits to be sure that some points are accessed always after other points? (*topological sort*)

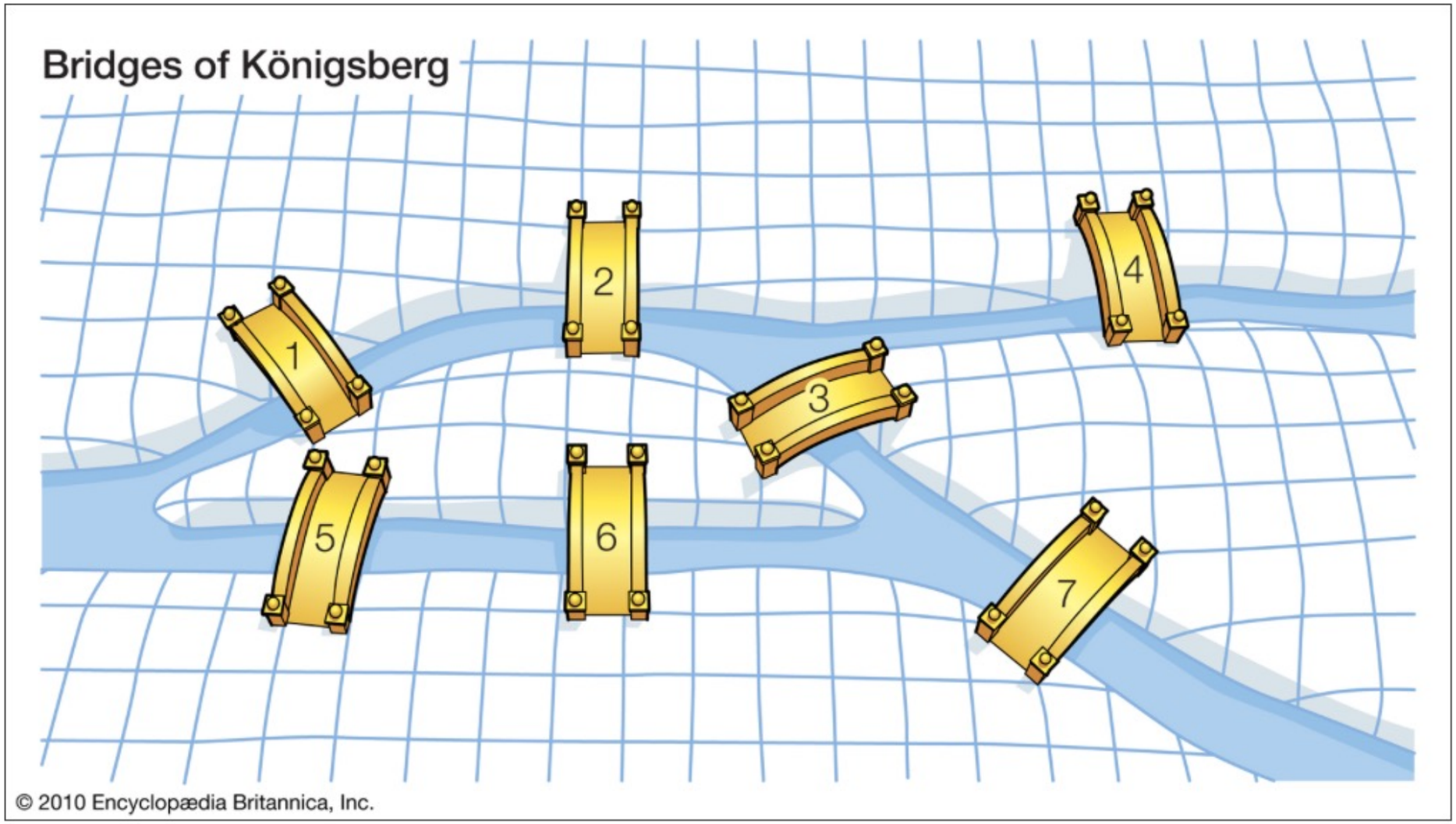
Conclusion

- The key to resolving problems related to previously mentioned real-world situations is to think of them in terms of graphs

- Modeling a real-world problem correctly in terms of graphs enables us to take advantage of rich field of graphy theory

Graphs

- Historical Note



In the 18th century, the Swiss mathematician Leonhard Euler was intrigued by the question of whether a route existed that would traverse each of the seven bridges exactly once. In demonstrating that the answer is no, he laid the foundation for graph theory.

Encyclopædia Britannica, Inc.

First Some Basic Definitions!

UnOrdered Pair

- An unordered pair is a set {a, b} representing the two objects a and b

Friendship b/w Alice and Bob {Alice, Bob}

- Remember {a} is also an unordered pair

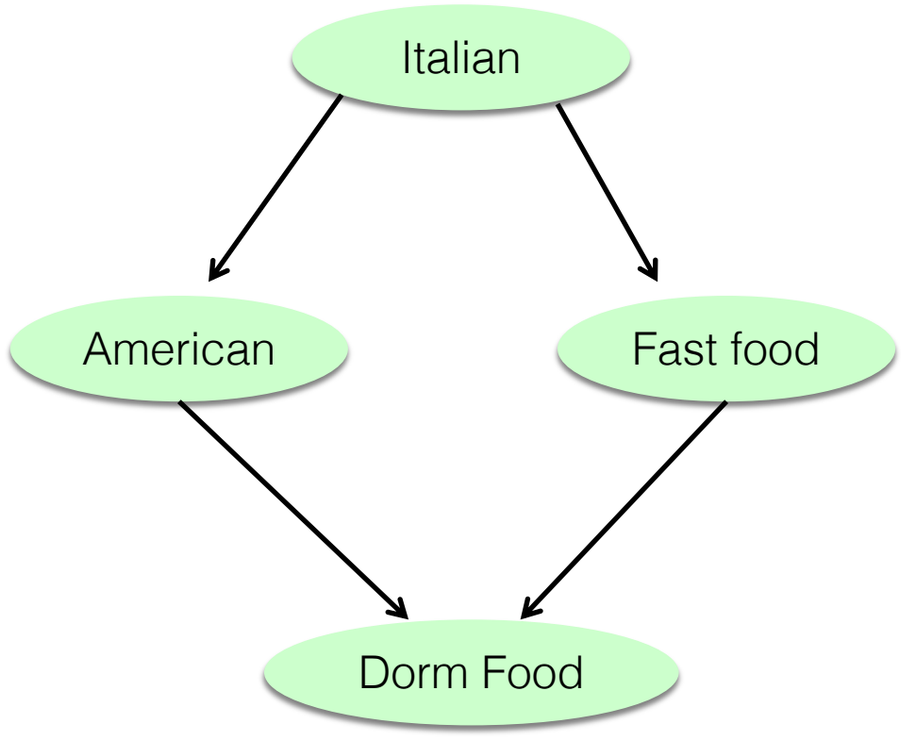
- Useful if we want to pair objects such that none of them is “first” or “second”

Ordered Pair

- Collection of two objects a and b in order

- (a, b)

- For example, an ordered pair of my food preferences



Two ordered pairs (a_0, b_0) and (a_1, b_1) are equal iff $a_0 = a_1$ and $b_0 = b_1$

Graphs

- A graph is defined as G = (V, E) where

  V is a set of vertices, and

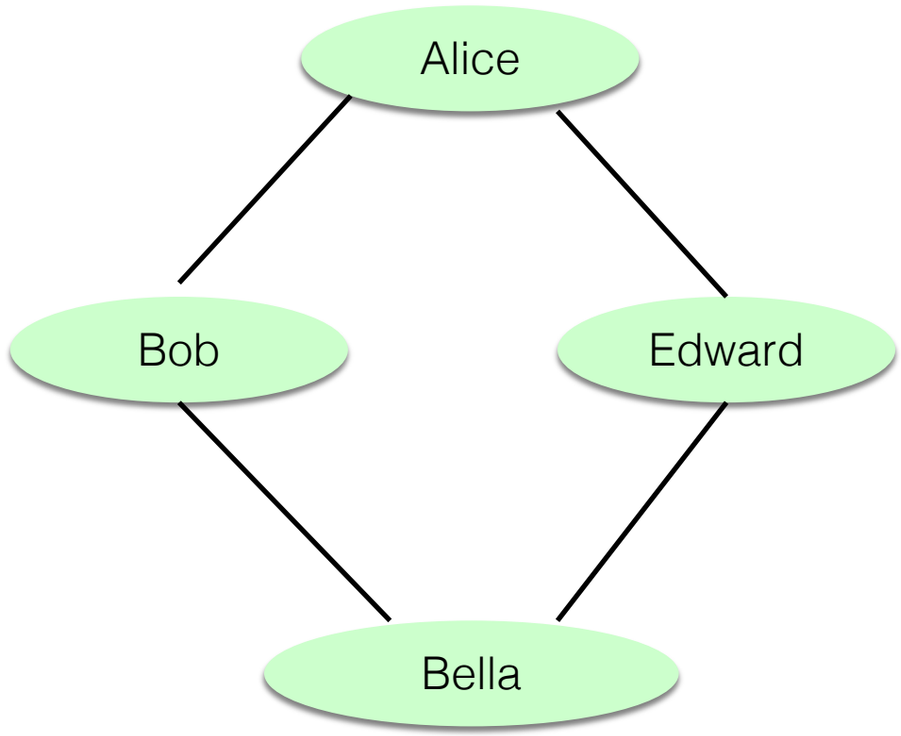
  E is a set of vertex pairs or edges

- **Vertex:** node in a graph

- **Edge:** a pair of vertices representing a connection between two nodes in a graph

Undirected Graphs

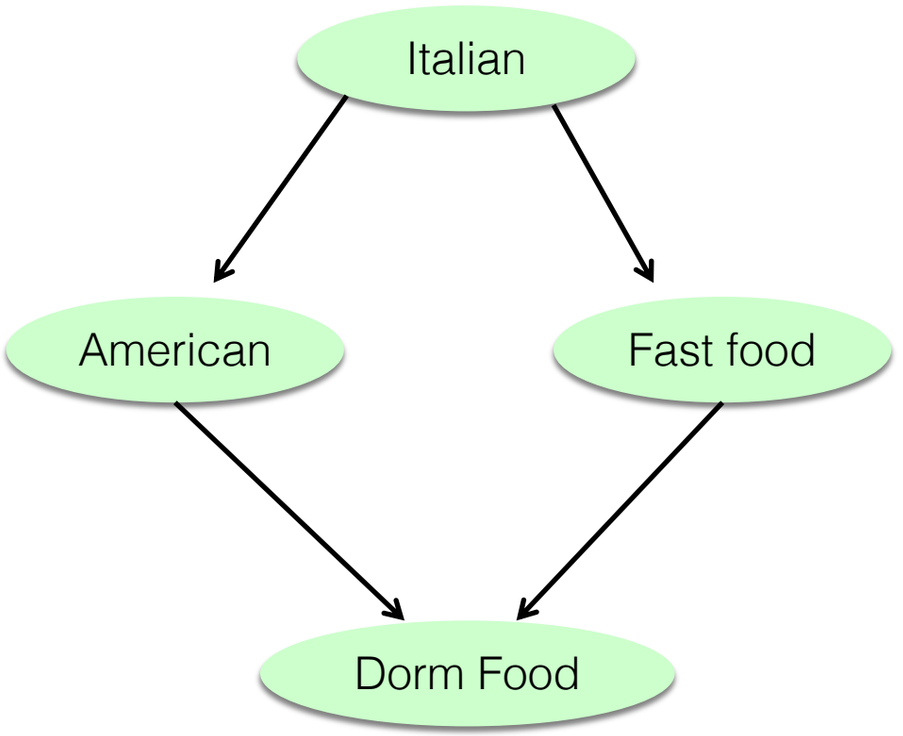
- A graph G = (V,E), where
- V is a set of vertices
- E is a set of unordered pairs



- V = { Alice, Bob, Edward, Bella}
- E = { {Alice, Bob}, {Alice, Edward}, {Bob, Bella}, {Edward, Bella} }

Directed Graphs

- A graph G = (V,E), where
- V is a set of vertices
- E is a set of ordered pairs



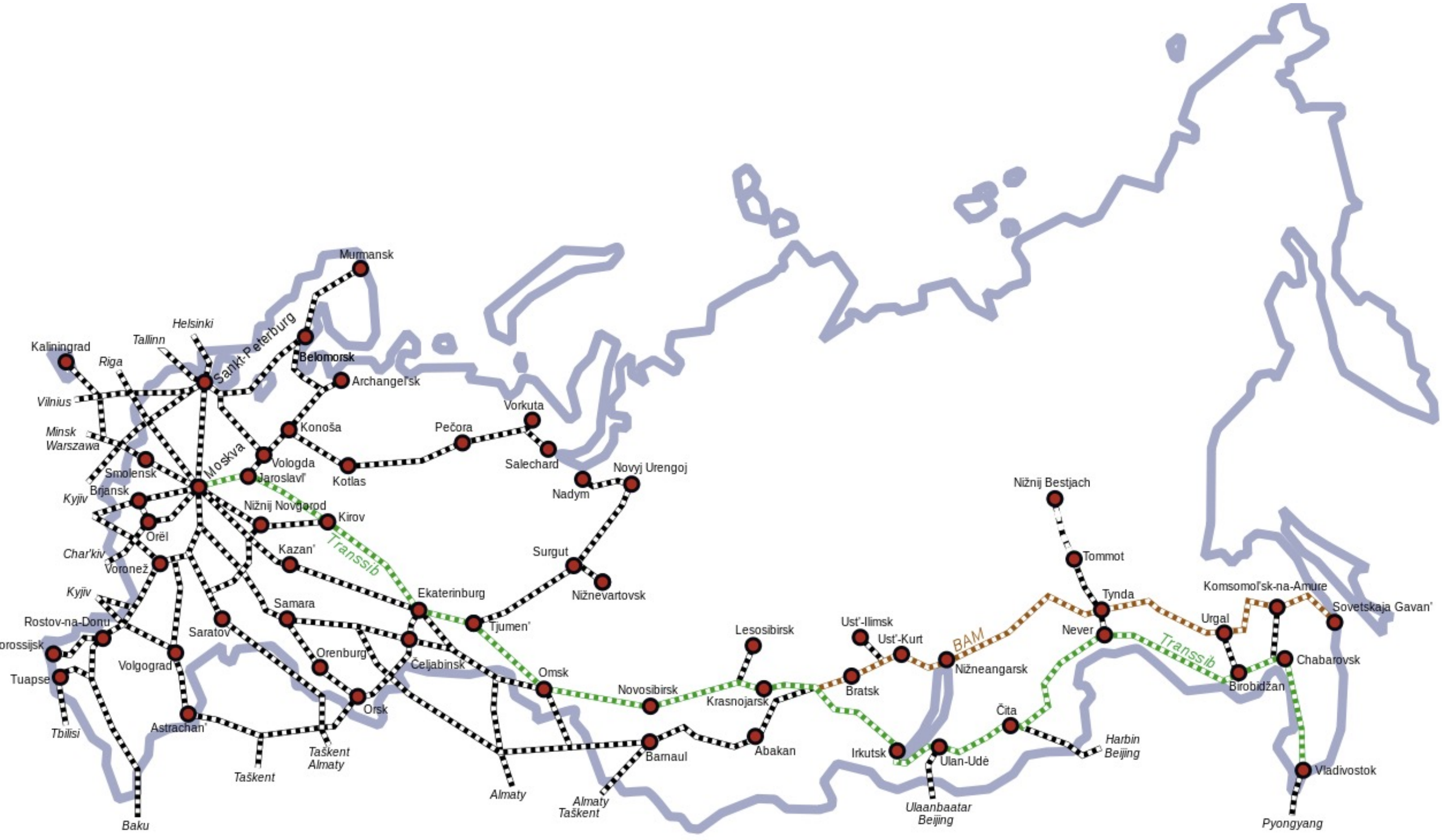
- V = { Italian, American,

  Fast food, Dormfood}

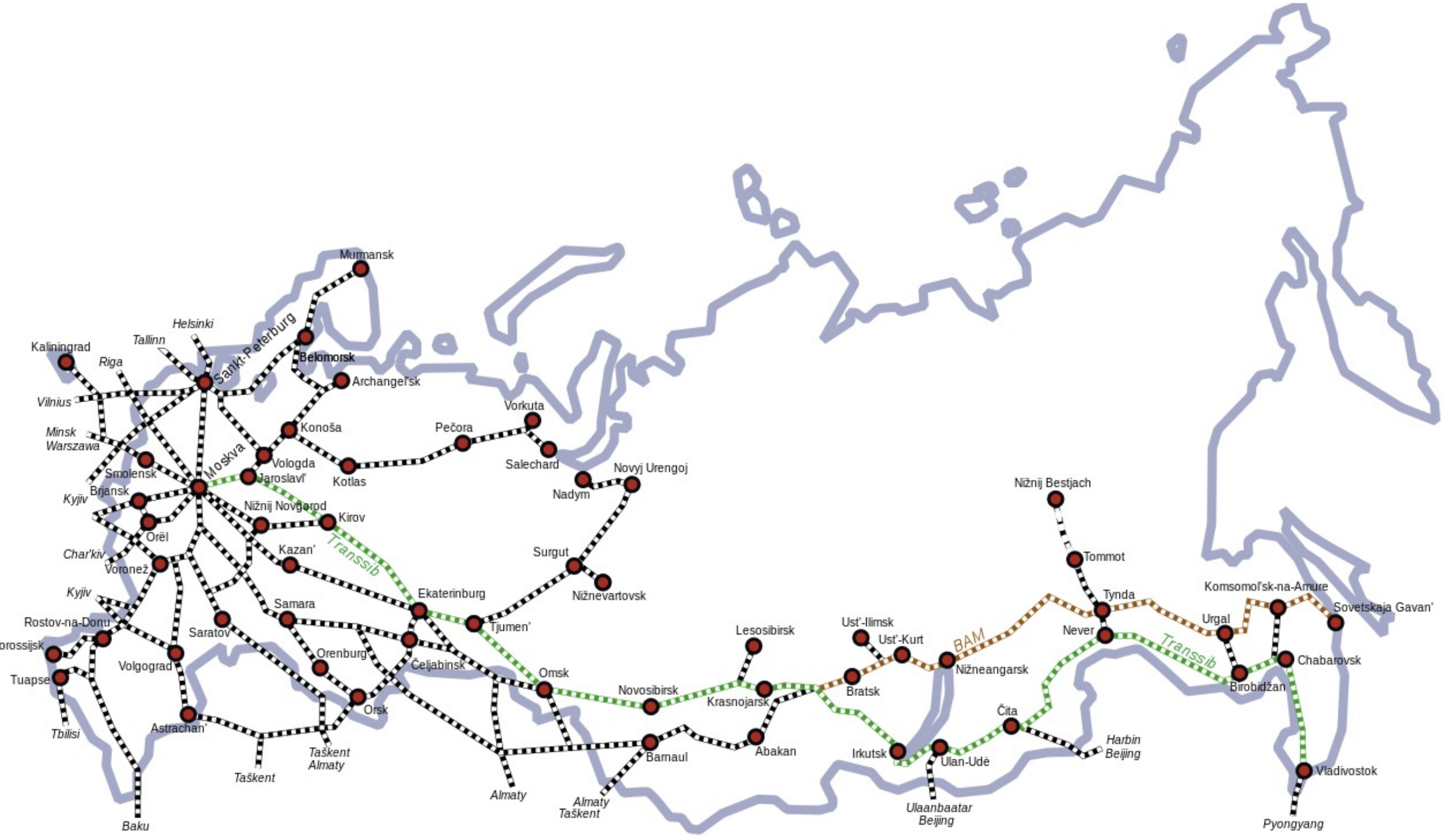
- E = { (Italian, American), (Italian, Fast Food), (American, Dorm Food), (Fast Food, Dorm Food) }

Russian Railway System

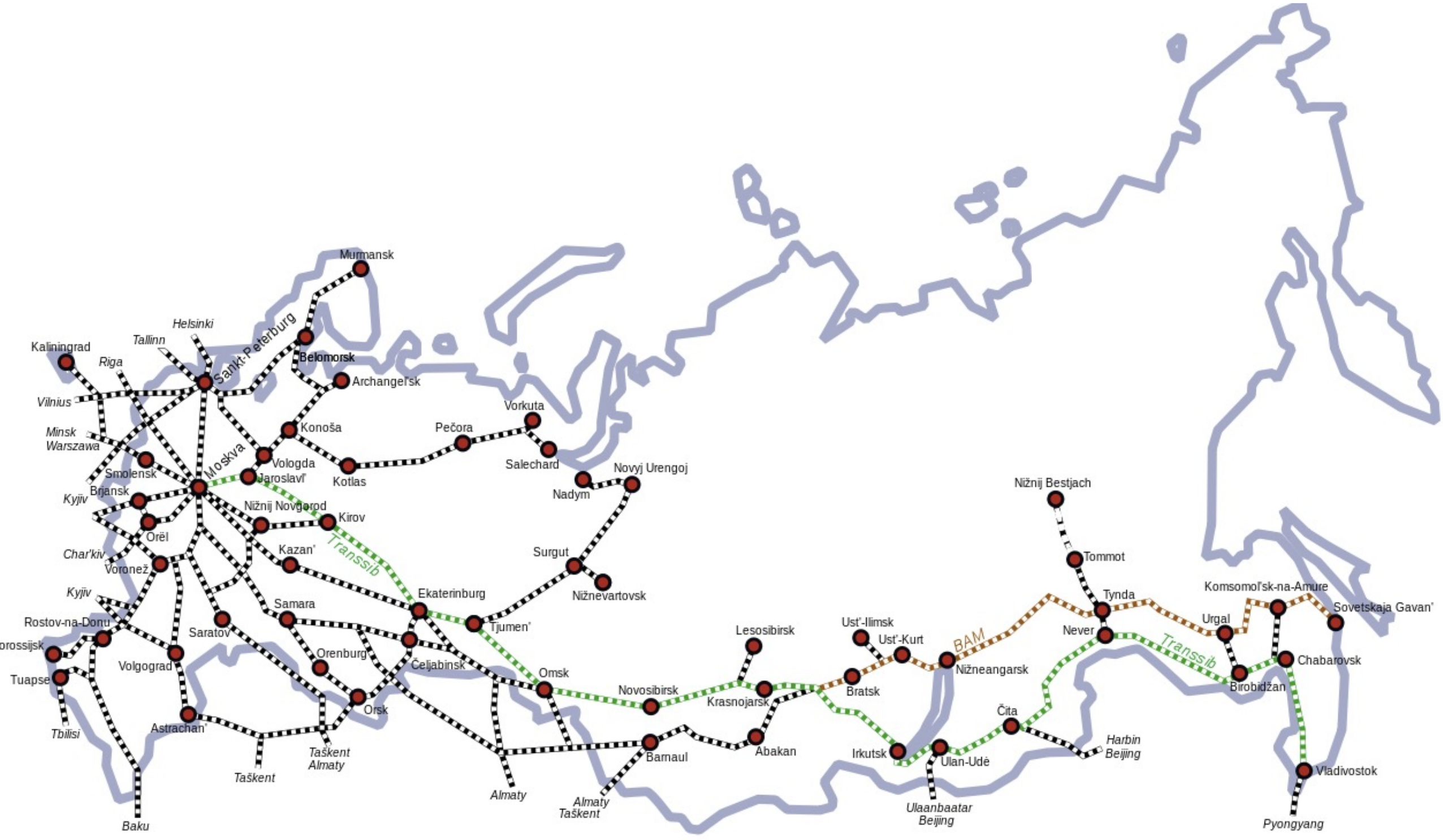
Russian Railway System



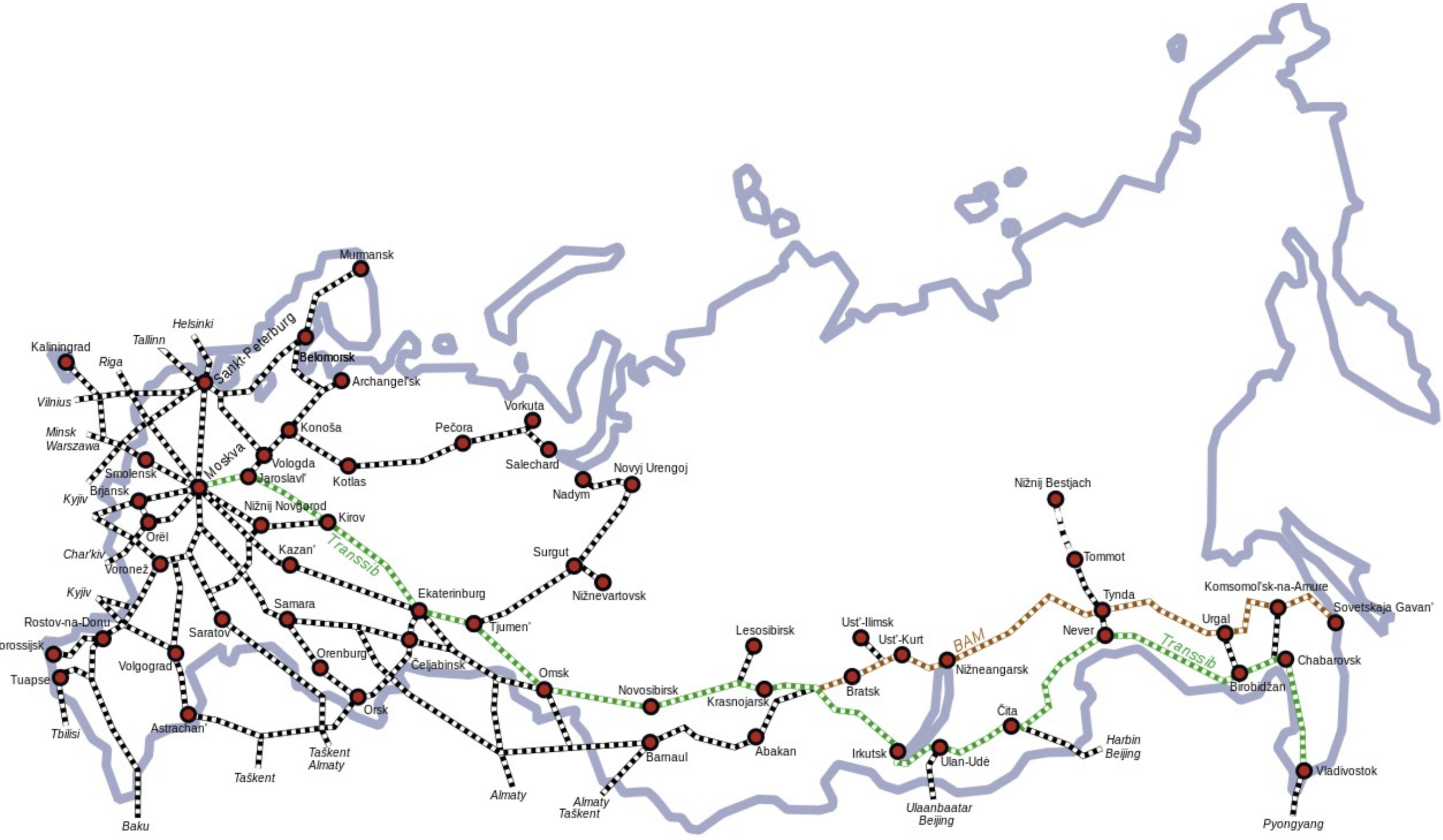
- **Adjacent vertices:** two vertices in a graph that are connected by an edge – Kazan and Moscow



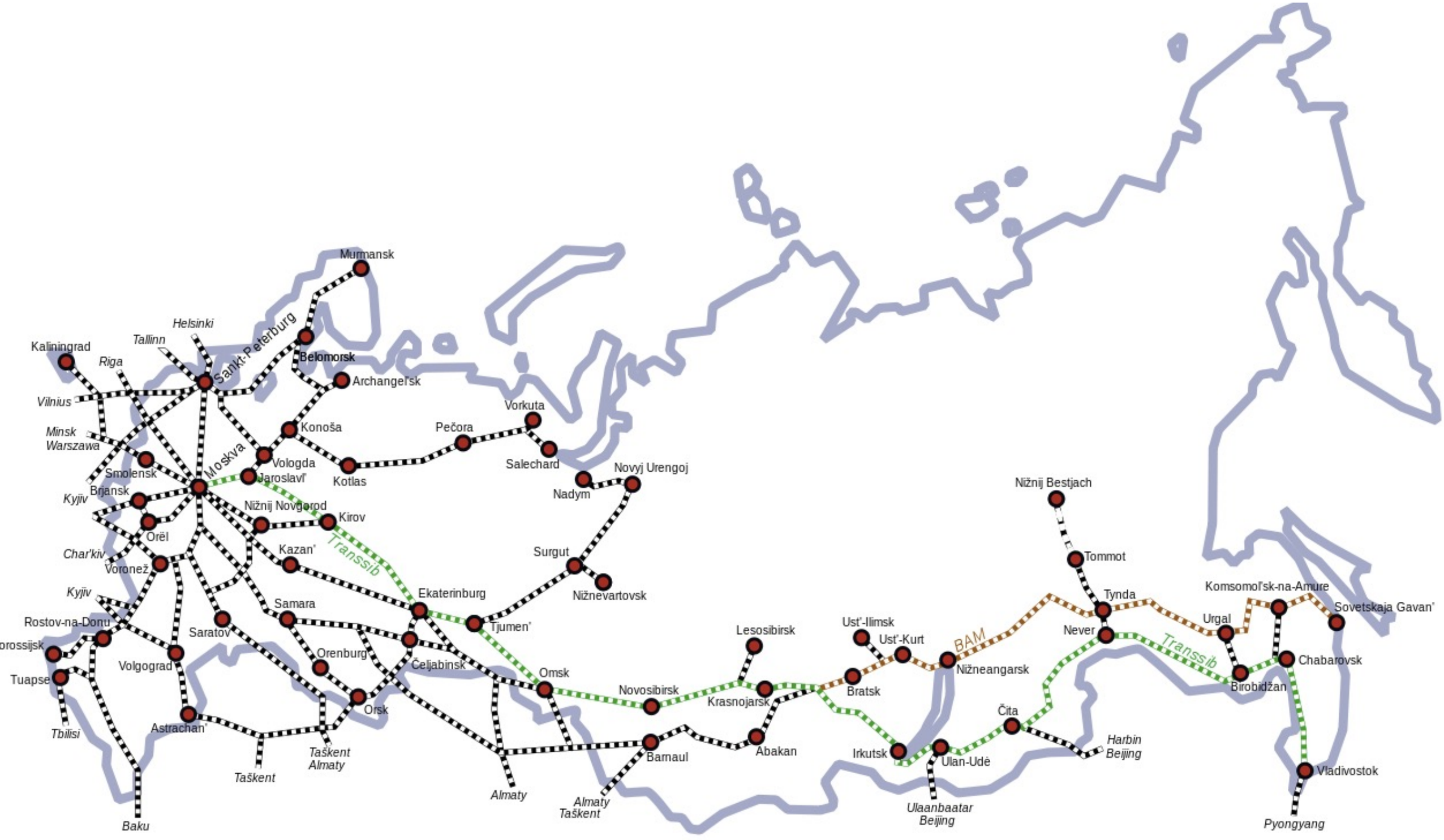
- **Path:** a sequence of vertices that connects two vertices – (Kazan, Moscow, Saint Petersburg)



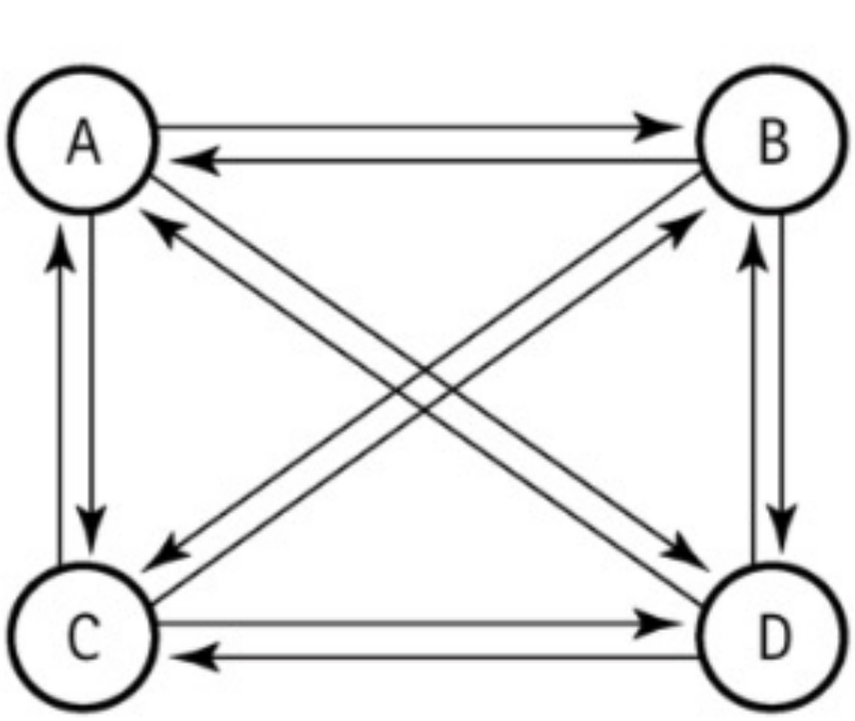
- **Simple Path:** A path with no repeated vertices– For example (Moscow, Kazan, Ekaterinburg) is a simple path



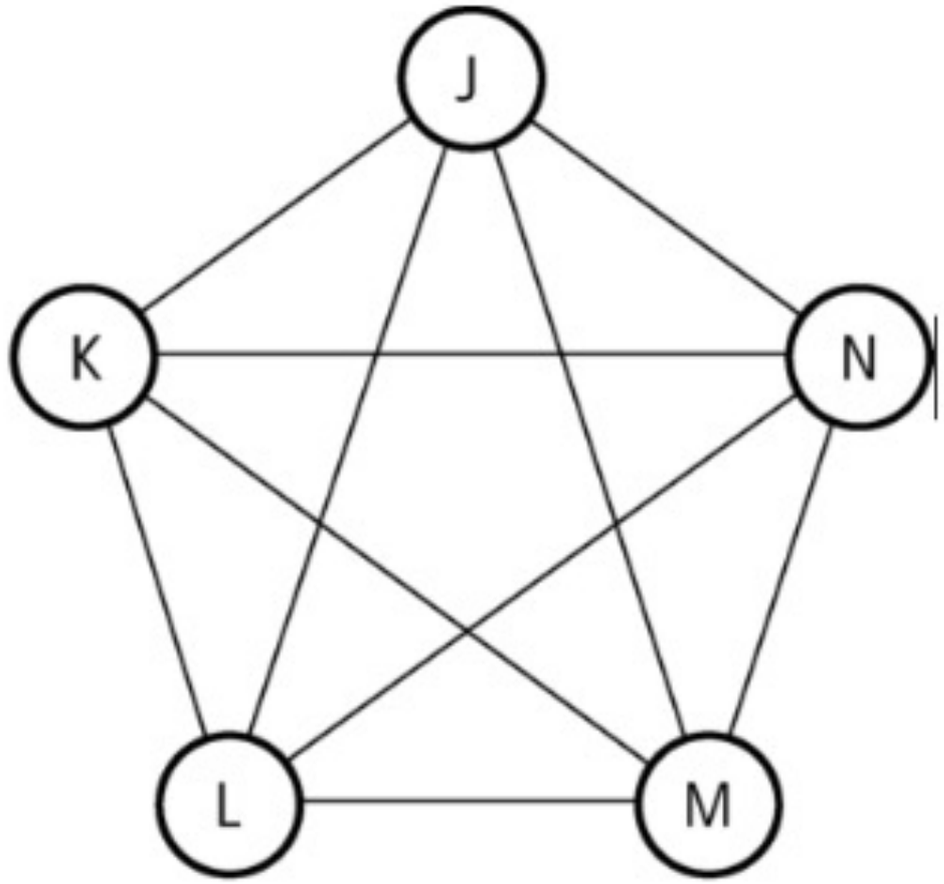
- **Cycle:** A path that starts and ends at the same vertex– (Kazan, Ekaterinburg, Kirov, Jaroslavl', Moscow, Kazan)



- **Simple Cycle:** A cycle that does not contain duplicate vertices – For example (Kazan, Ekaterinburg, Kirov, Ekaterinburg, Kazan) is NOT a simple cycle

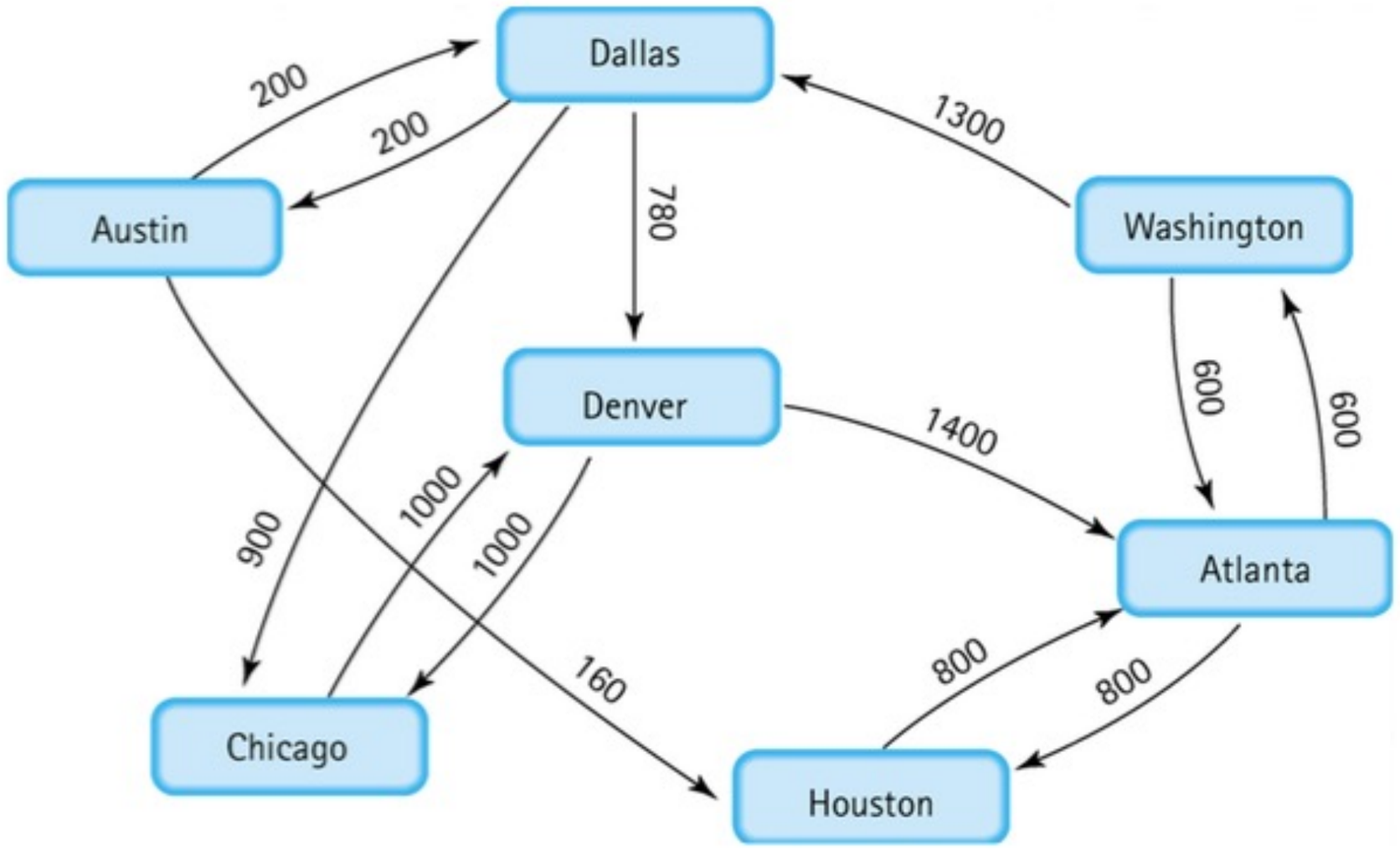


A complete directed graph G



A complete undirected graph G

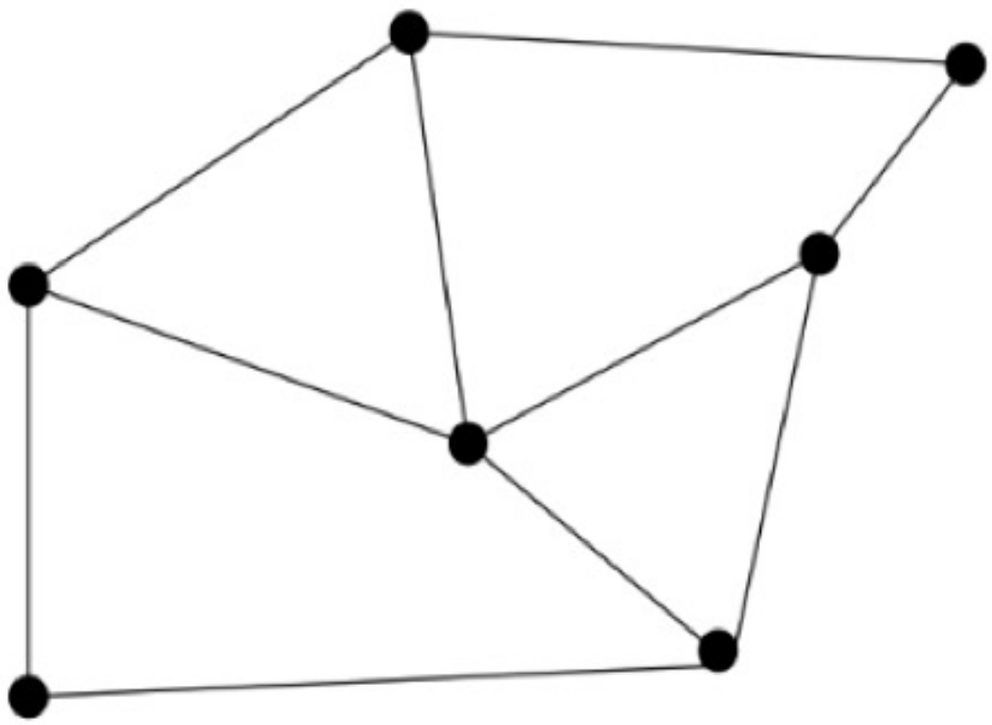
- **Complete:** A graph in which every vertex is directly connected to every other vertex



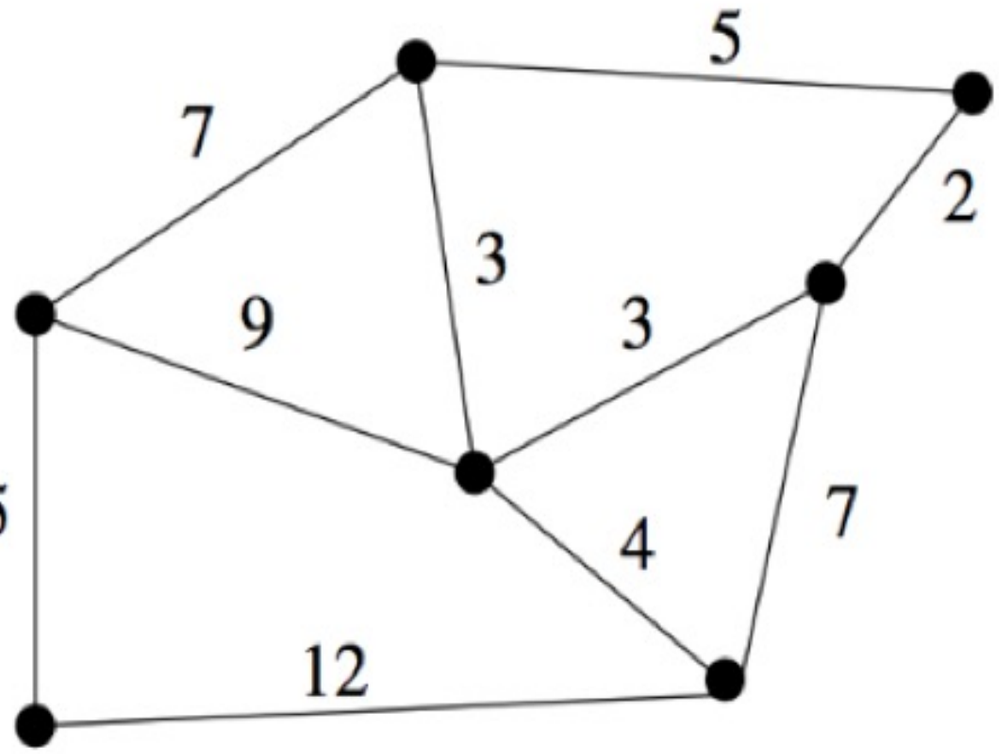
A weighted graph G

- **Weighted graph:** a graph in which each edge carries a value (weight)

Shortest Path



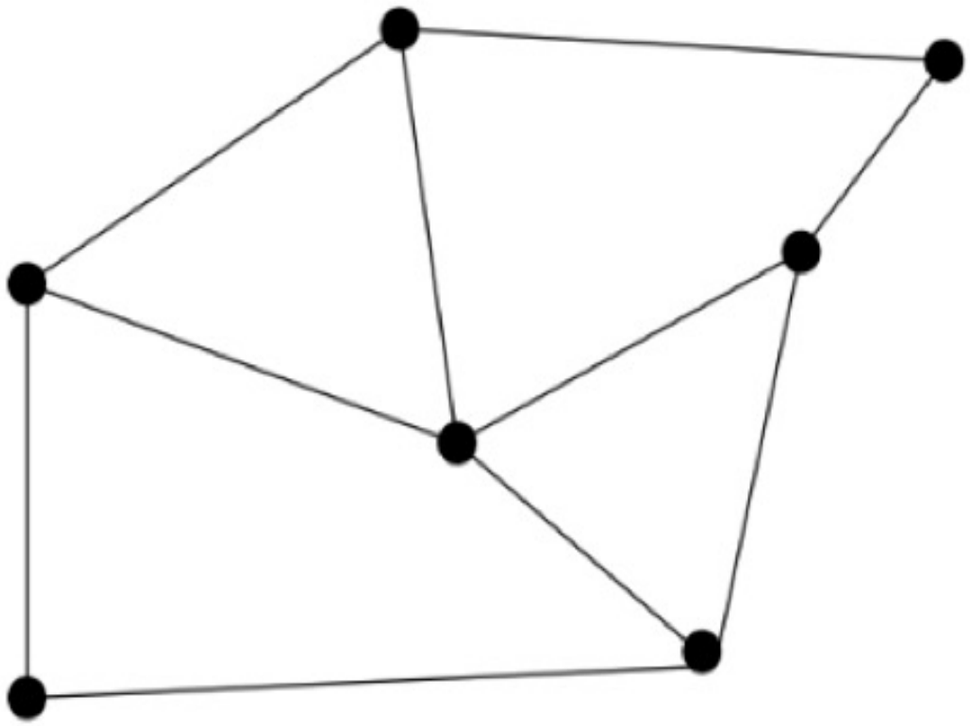
unweighted



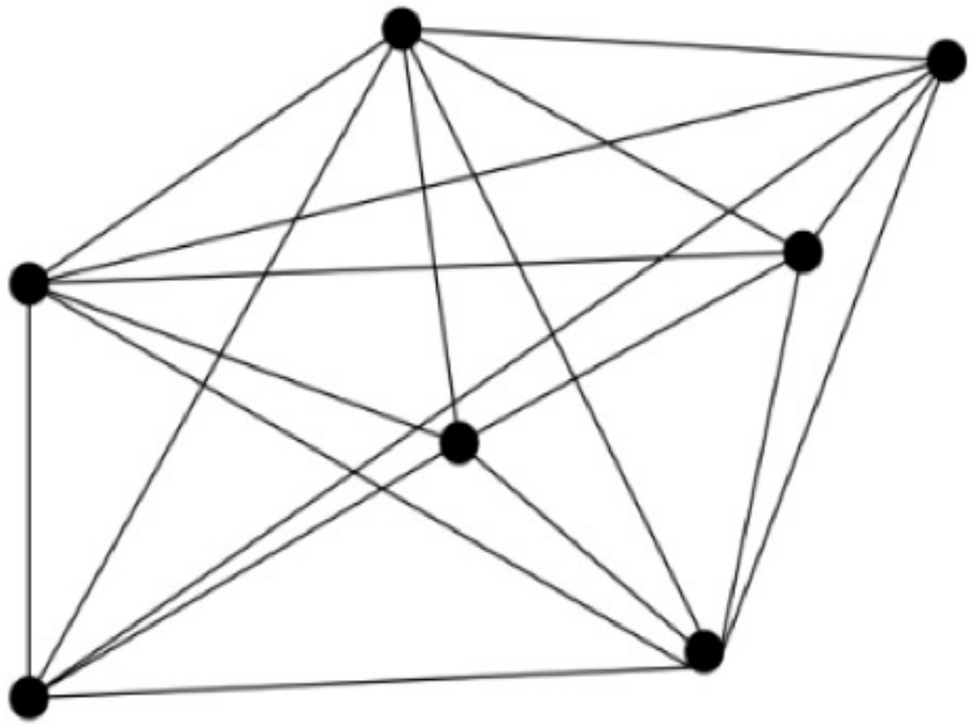
weighted

- For unweighted graphs, it's a path with the fewest number of edges – can be found using BFS or DFS
- For weighted graphs, more sophisticated algorithms are required

Sparse vs. Dense



sparse



dense

- There are maximum n(n-1)/2 total pair of vertices (edges) in an undirected graph of n vertices, with no self loops and no multiple edges

Graphs

- You are expected to know more about graph theory, important properties, theorems and proofs associated with those properties from your "Discrete Math" course

- You will cover some basic properties in tutorial

- Here, we will focus on how to implement them as an abstract data type

Main Methods of the Graph ADT

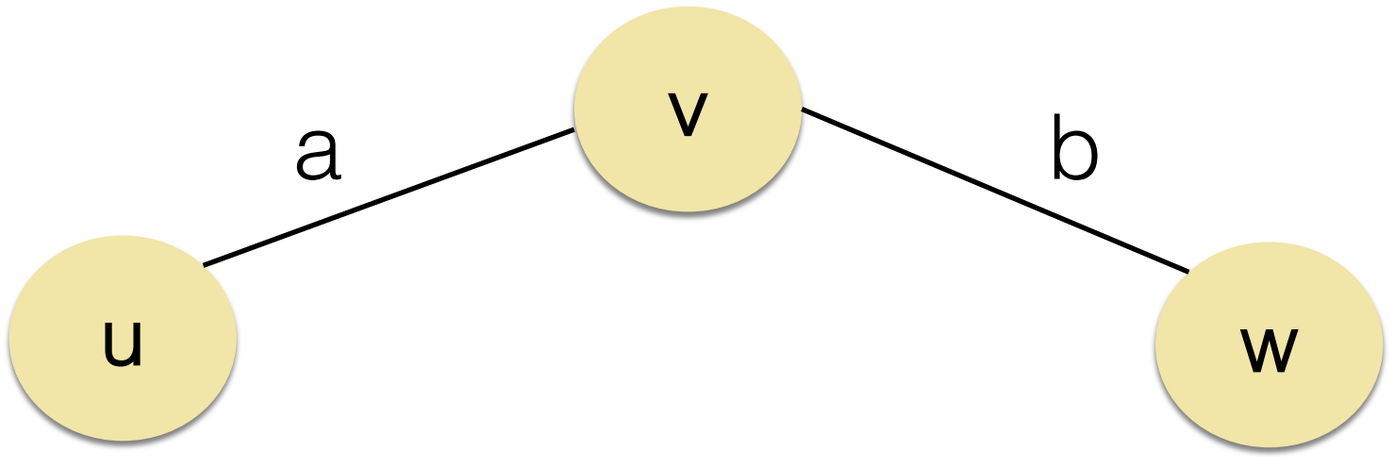
- endVertices(e): an array of the two end vertices of edge e
- opposite(v, e): the vertex opposite of vertex v on edge e
- areAdjacent(v, w): true iff v and w are adjacent vertices
- degree(v): # of incident edges
- insertVertex(o): insert a vertex storing element o
- insertEdge(v, w, o): insert an edge (v,w) storing element o
- removeVertex(v): remove vertex v (and its incident edges)
- removeEdge(e): remove edge e
- incidentEdges(v): edges incident to v

Graph Representations

Using Linked List

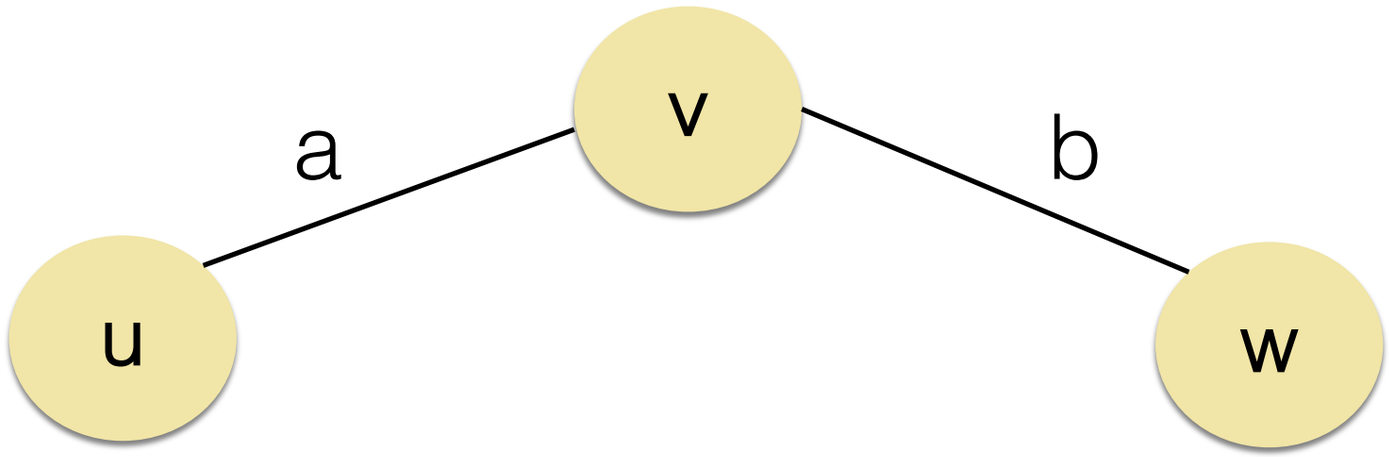
- As we know G = (V, E)
- Let’s use singly linked lists to store vertices and edges
  - Vertex List: stores vertices
  - Edge List: stores edges
- Referred to as the *Edge List Structure*

Example We will Use



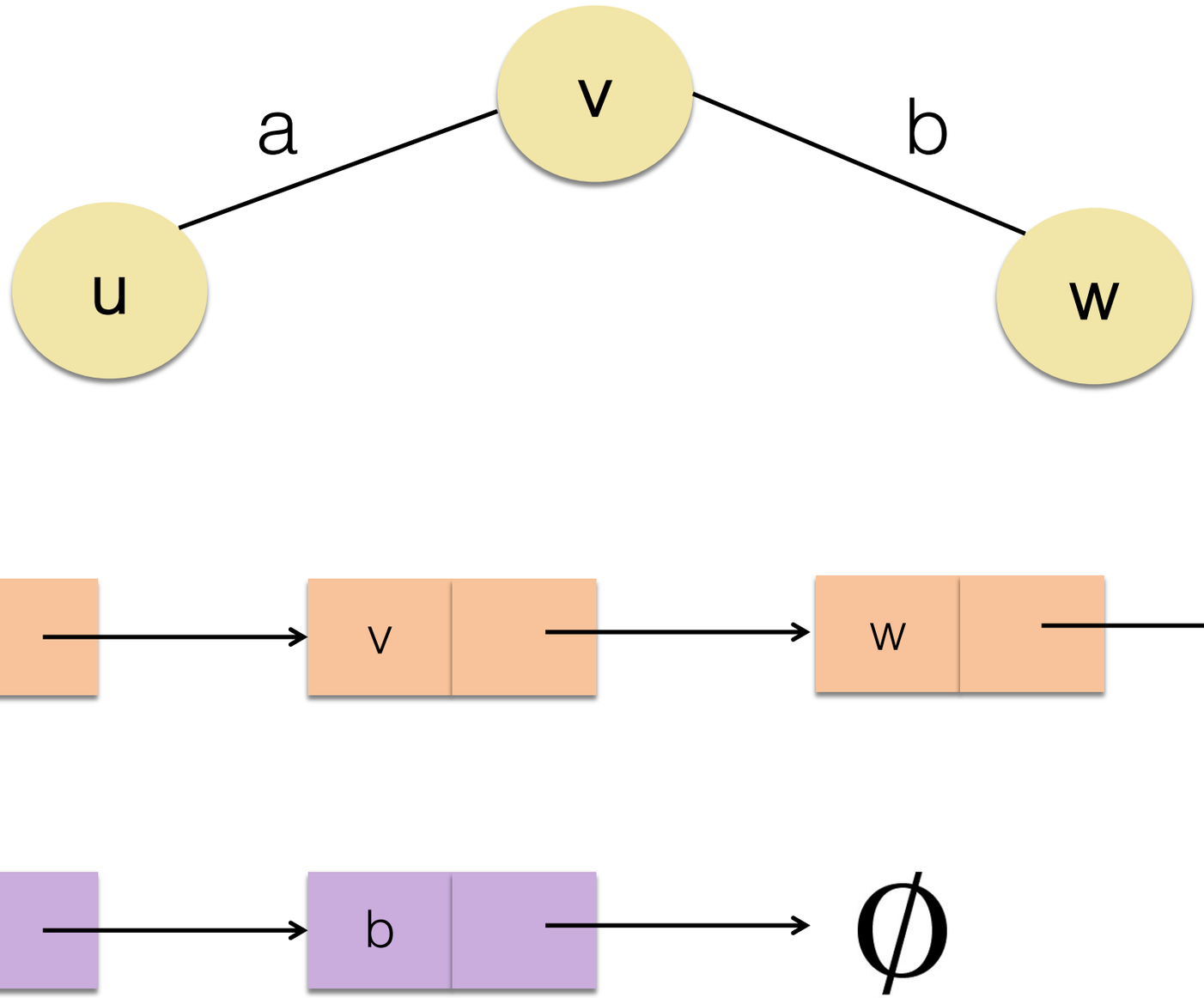
- a and b represent the names of edges
- They do not represent weights
- For example, a can be the highway M7 from Kazan to Moscow

Example We will Use

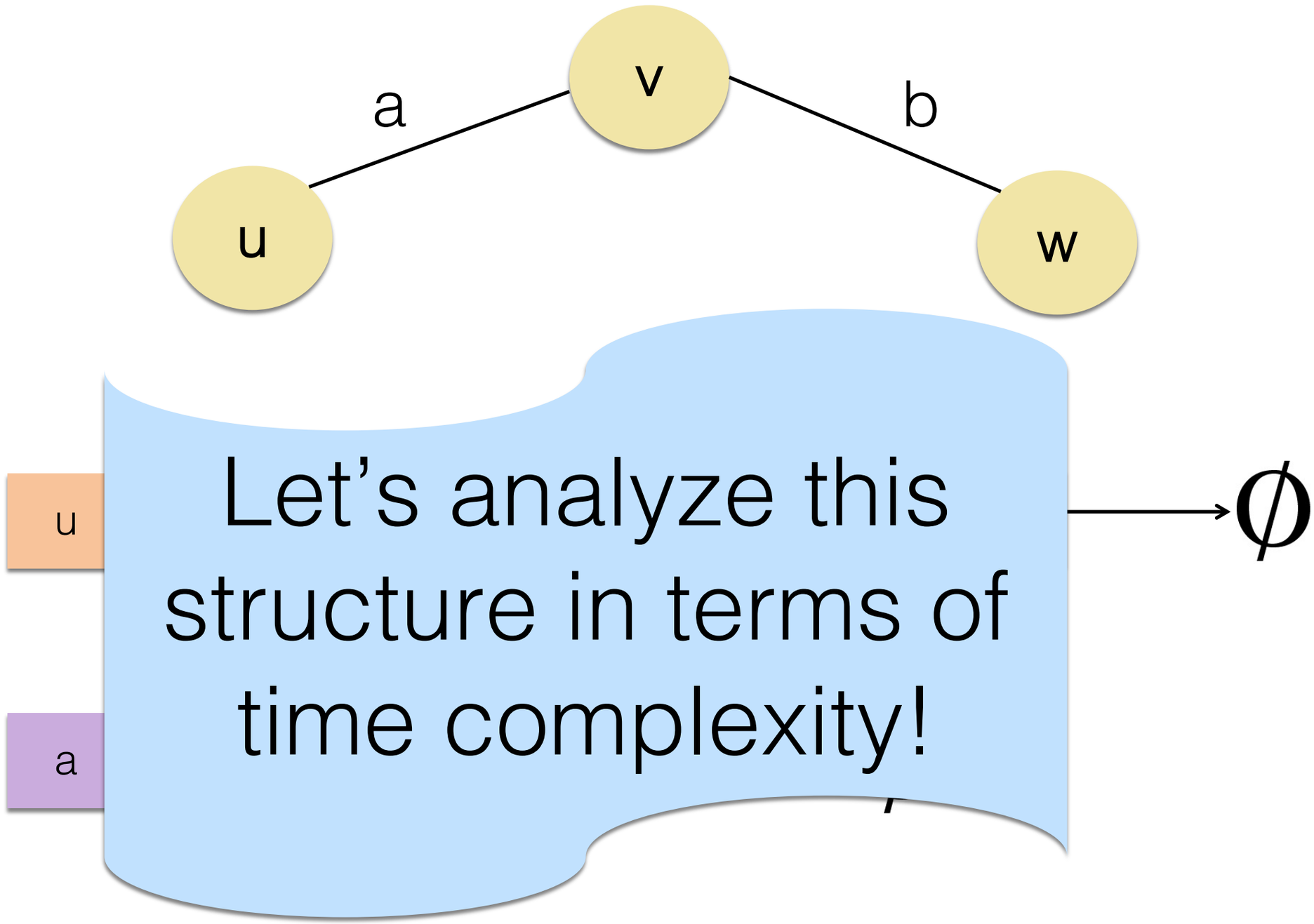


- Also, we will deliberately start very simple (incomplete info)
- Gradually moving towards the complete representation

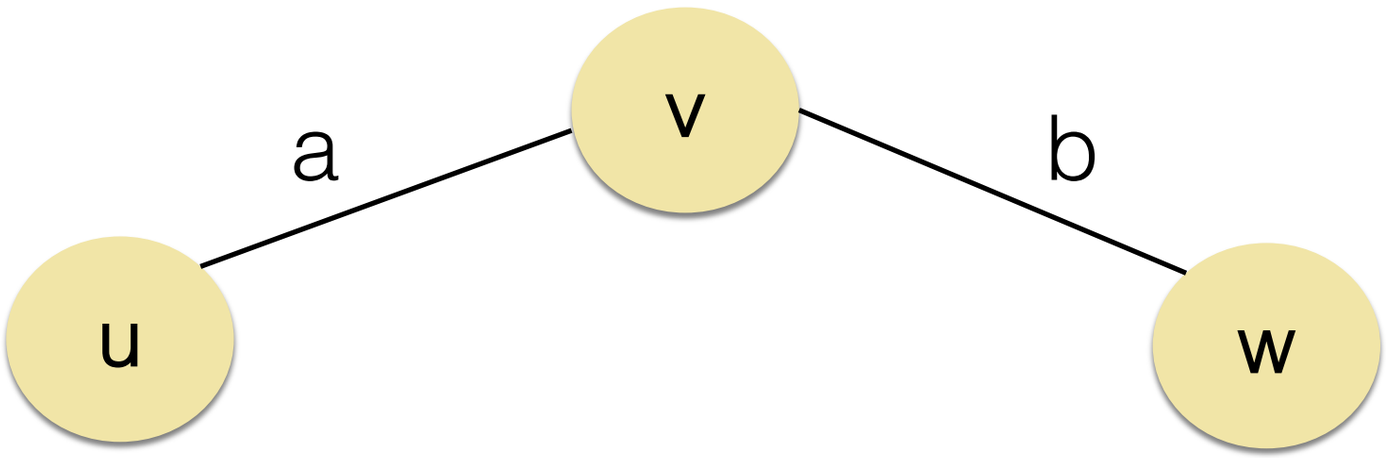
Edge List Structure



Edge List Structure

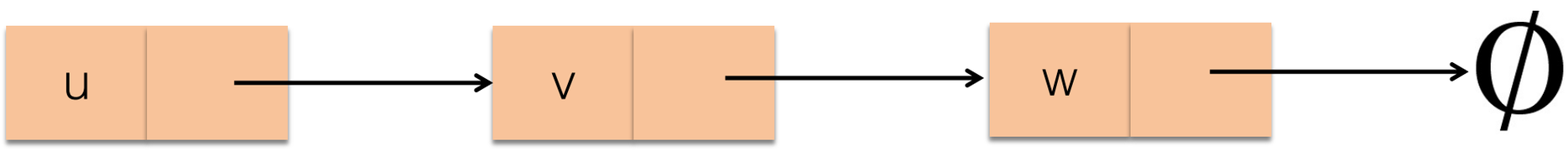


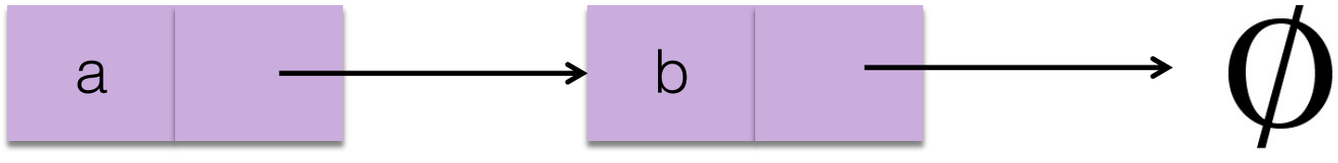
Edge List Structure



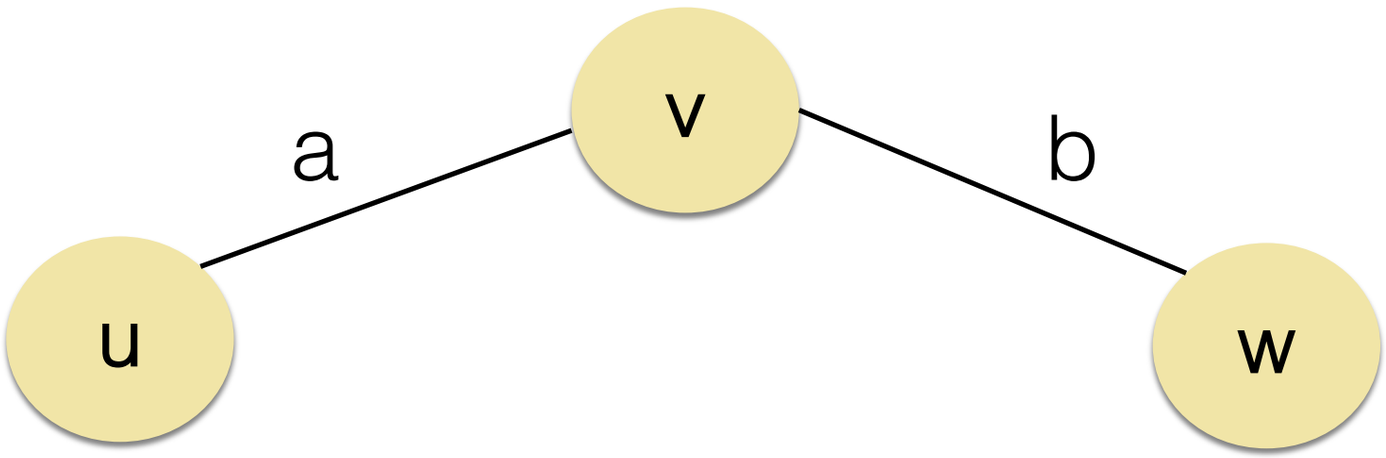
Insertion of a new vertex

O(1)



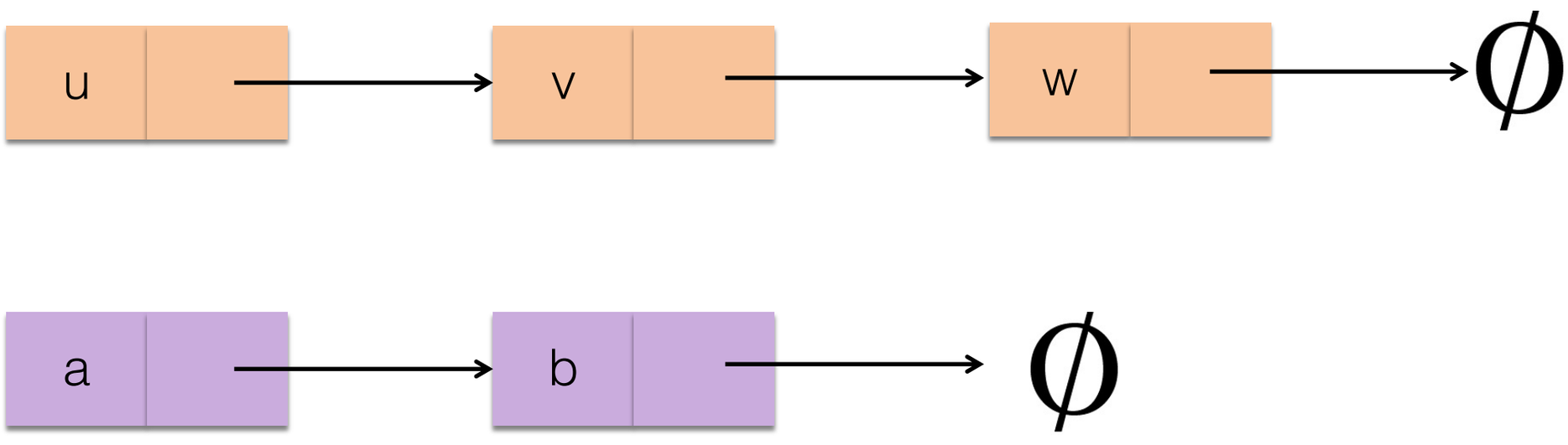


Edge List Structure

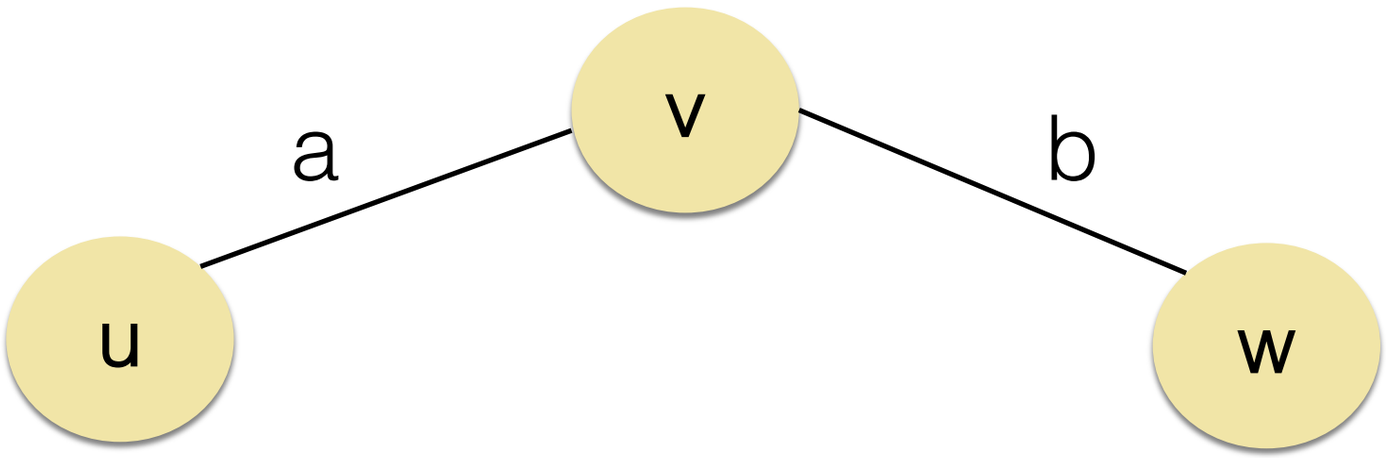


Insertion of a new edge

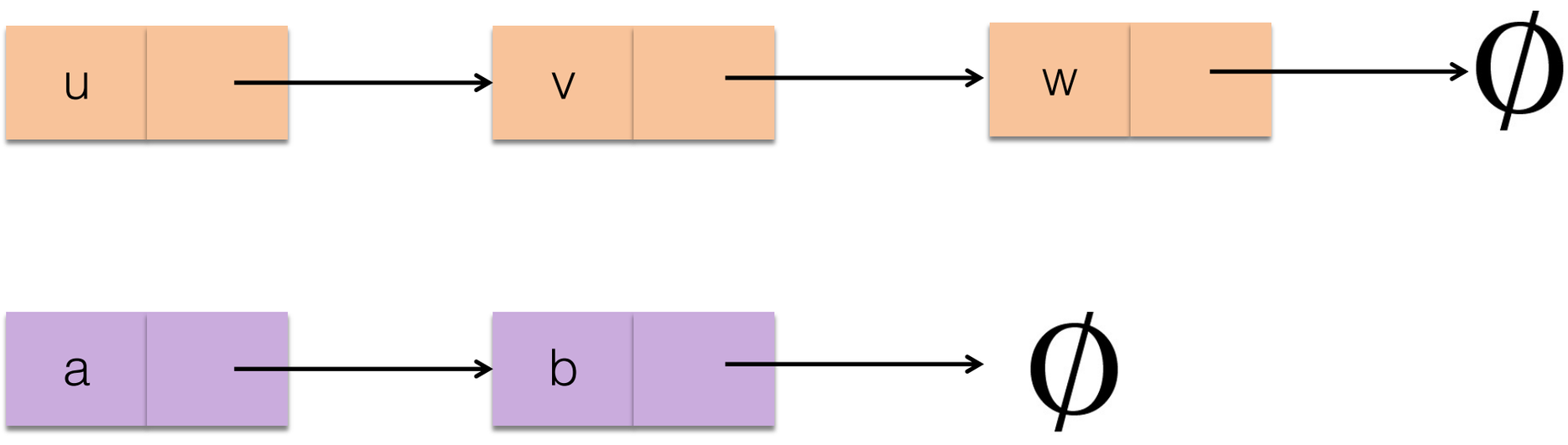
O(1)



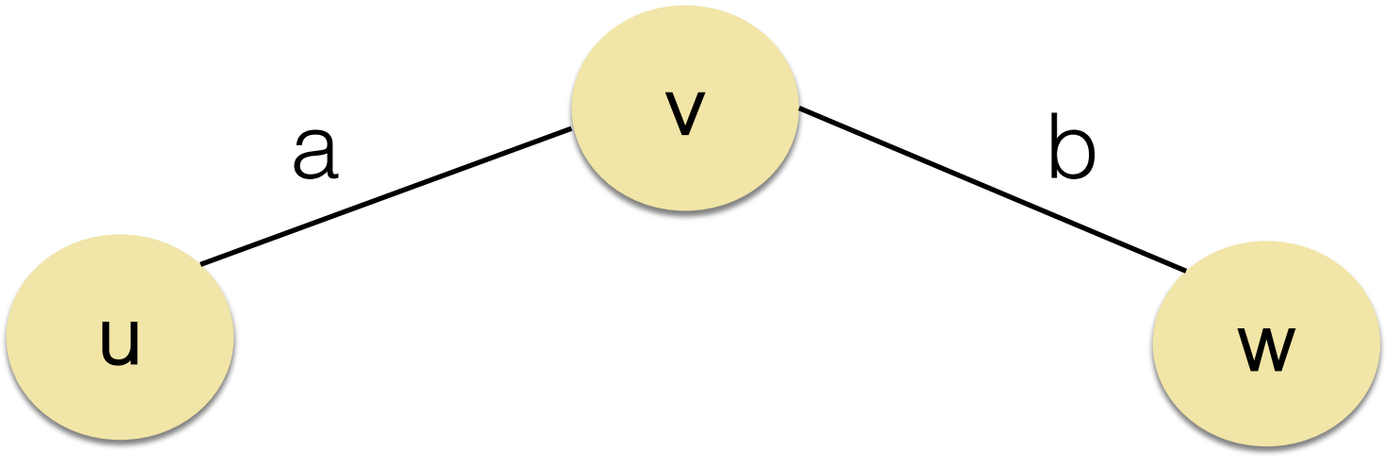
Edge List Structure



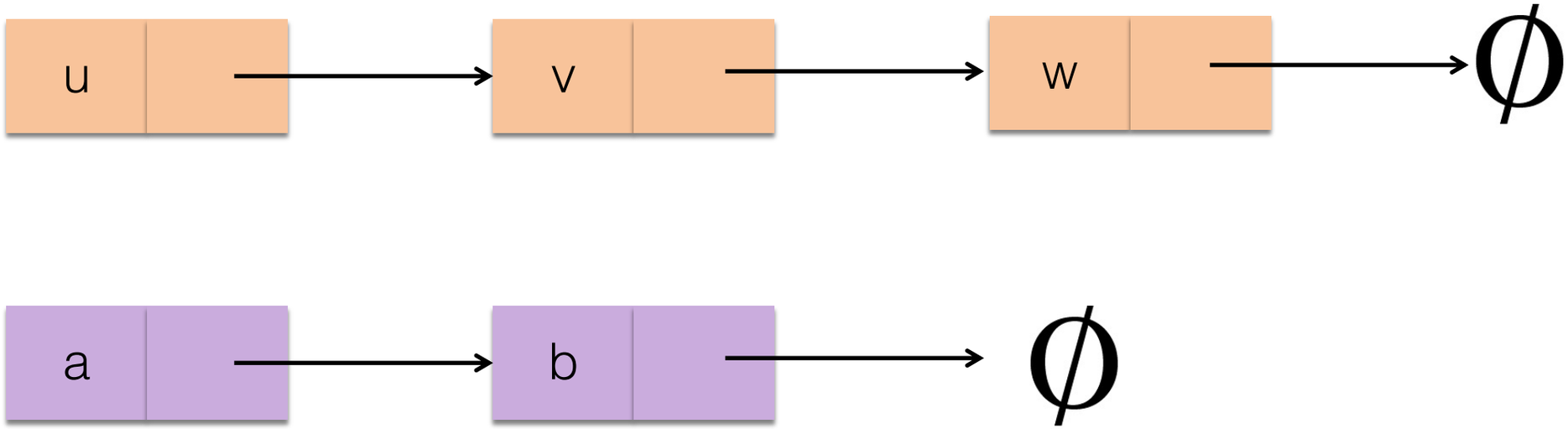
But can you find whether two vertices are adjacent?



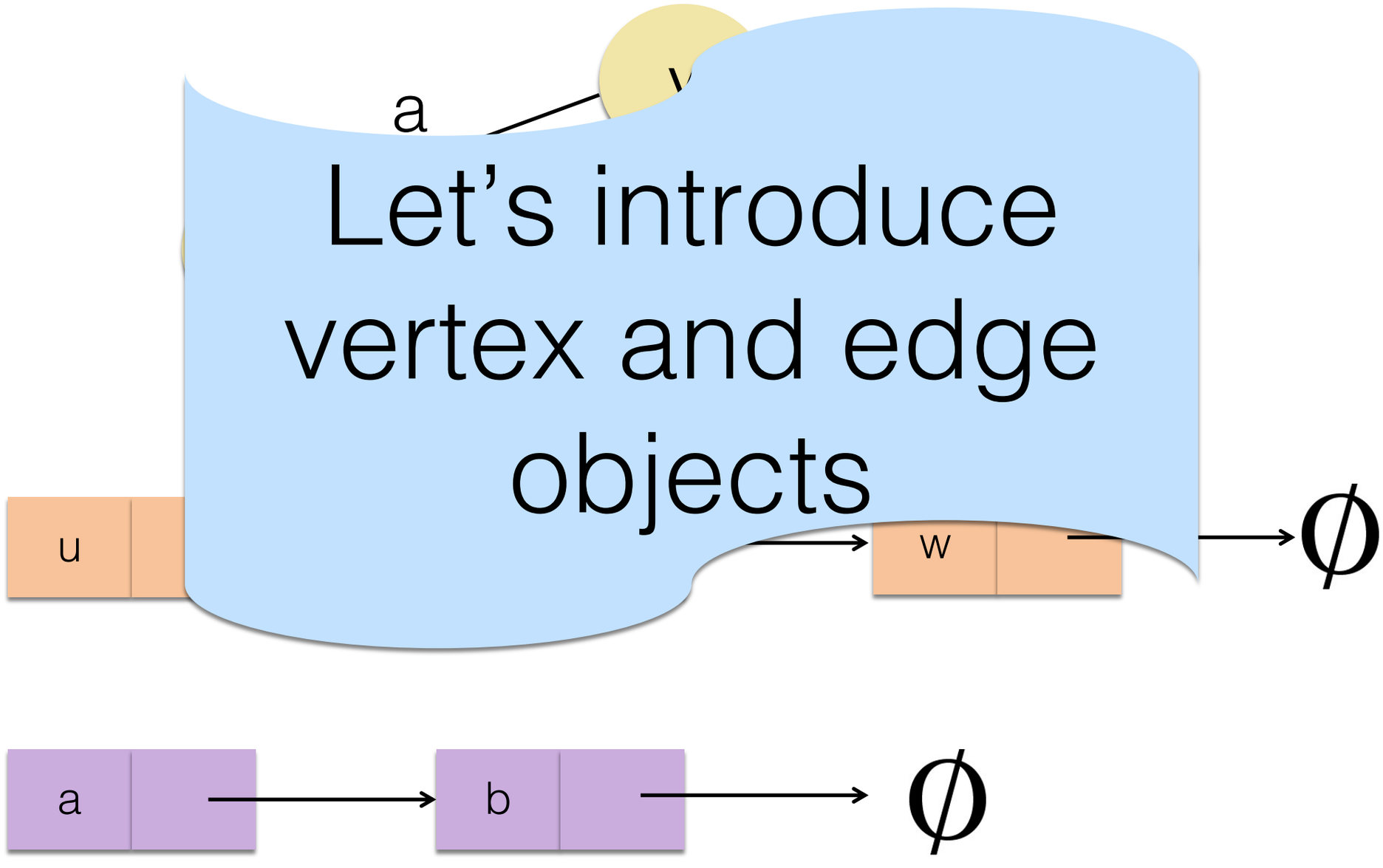
Edge List Structure



Identify some more problems!



Edge List Structure

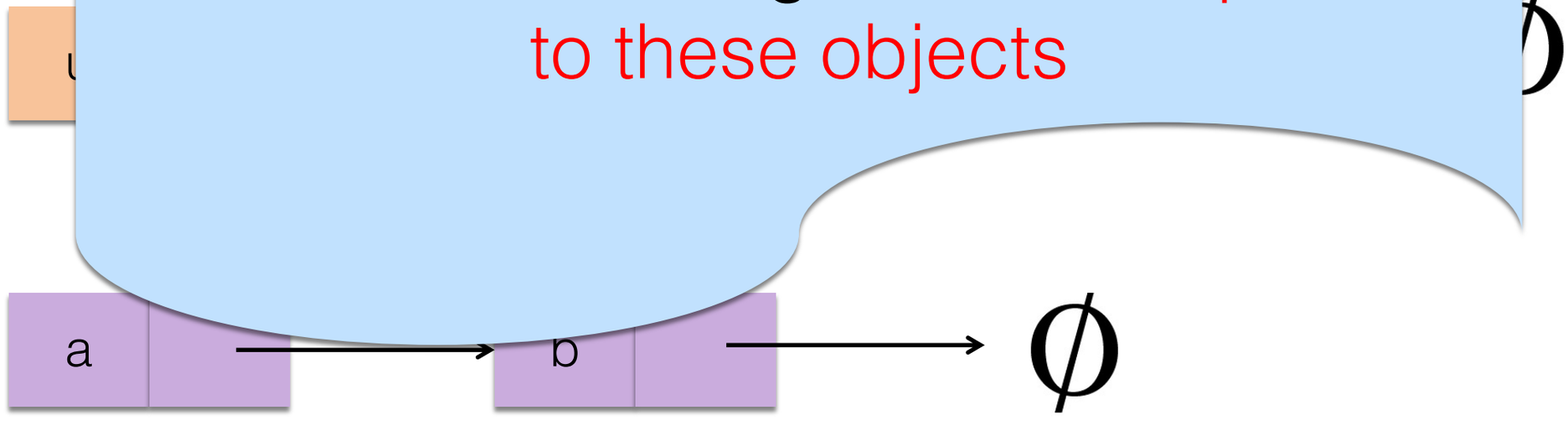


Edge List Structure

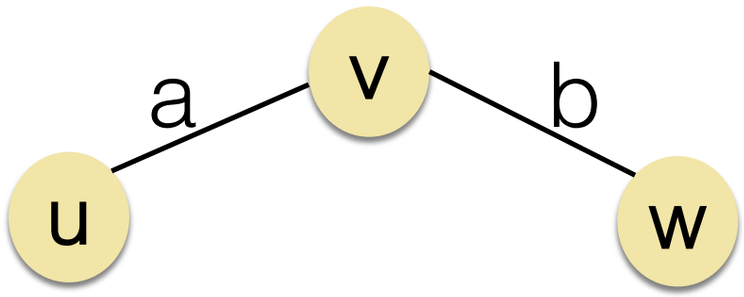
Vertex object stores element!

Edge Object stores element, origin, and destination

Both Vertex and Edge lists store pointers to these objects



Edge List Structure



Vertex List

u

v

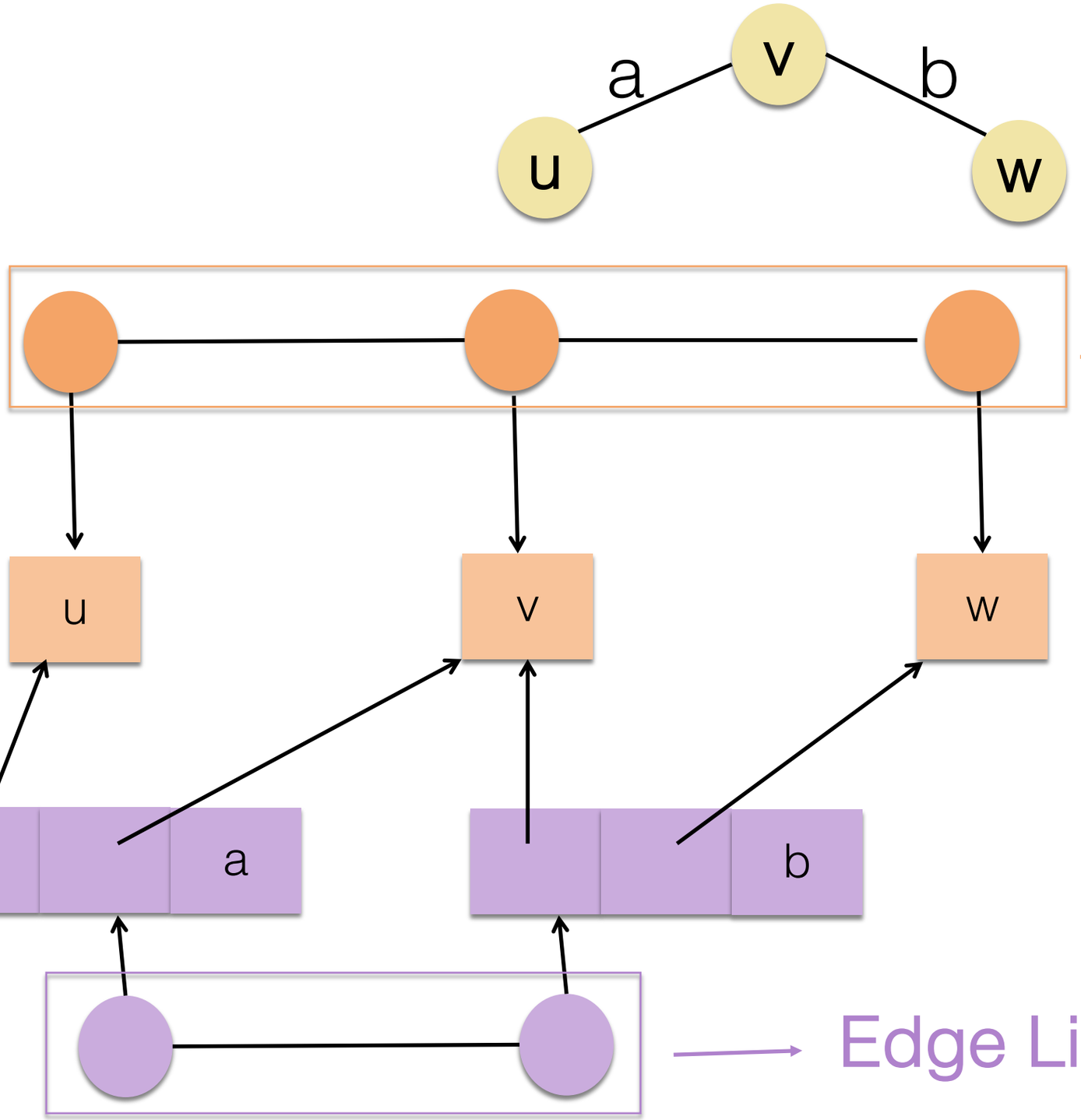
w

a

b

Edge List

Edge List Structure

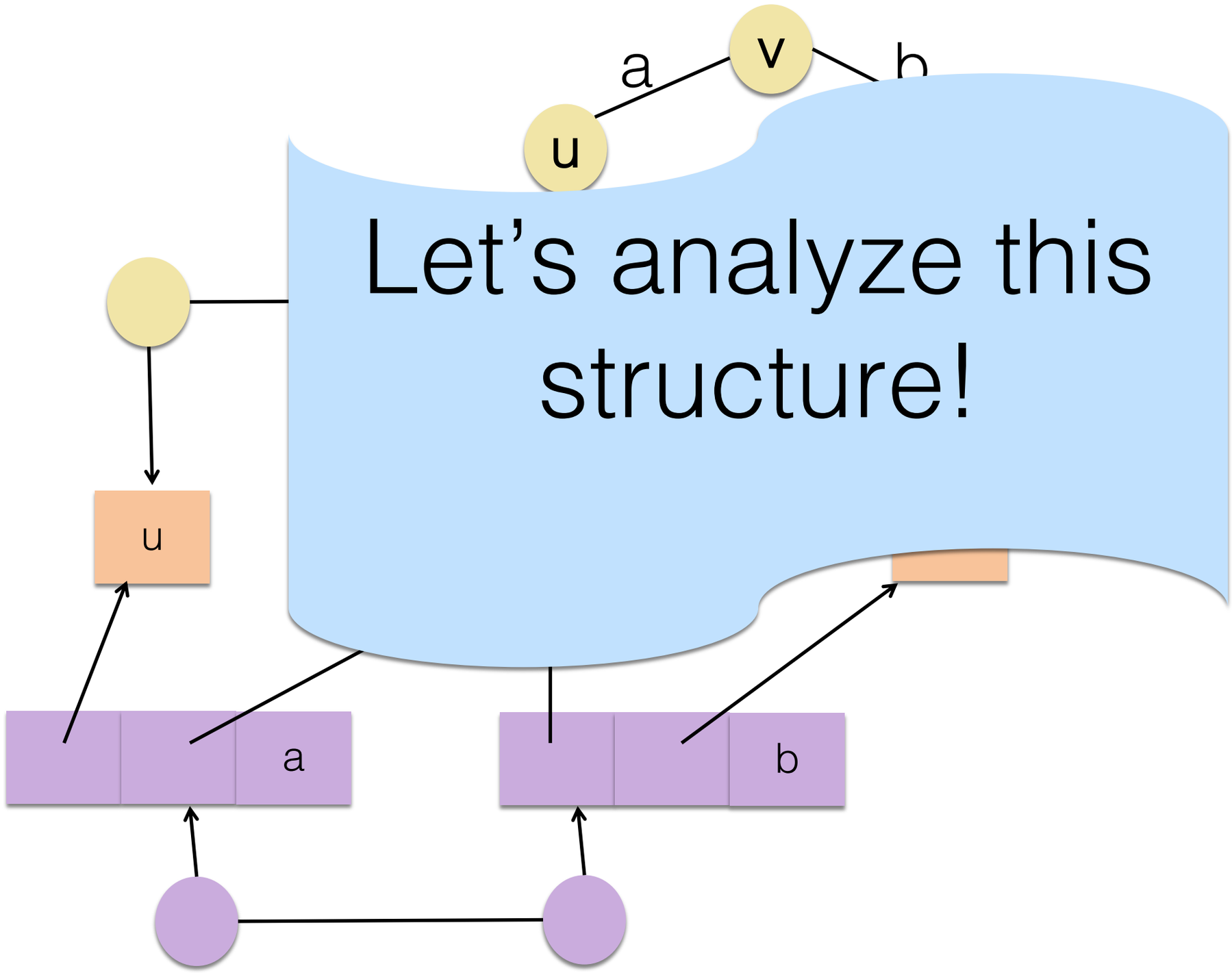


Visually Simplified

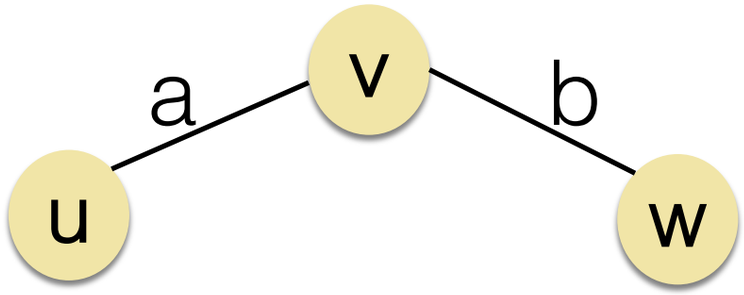
Vertex List

Edge List

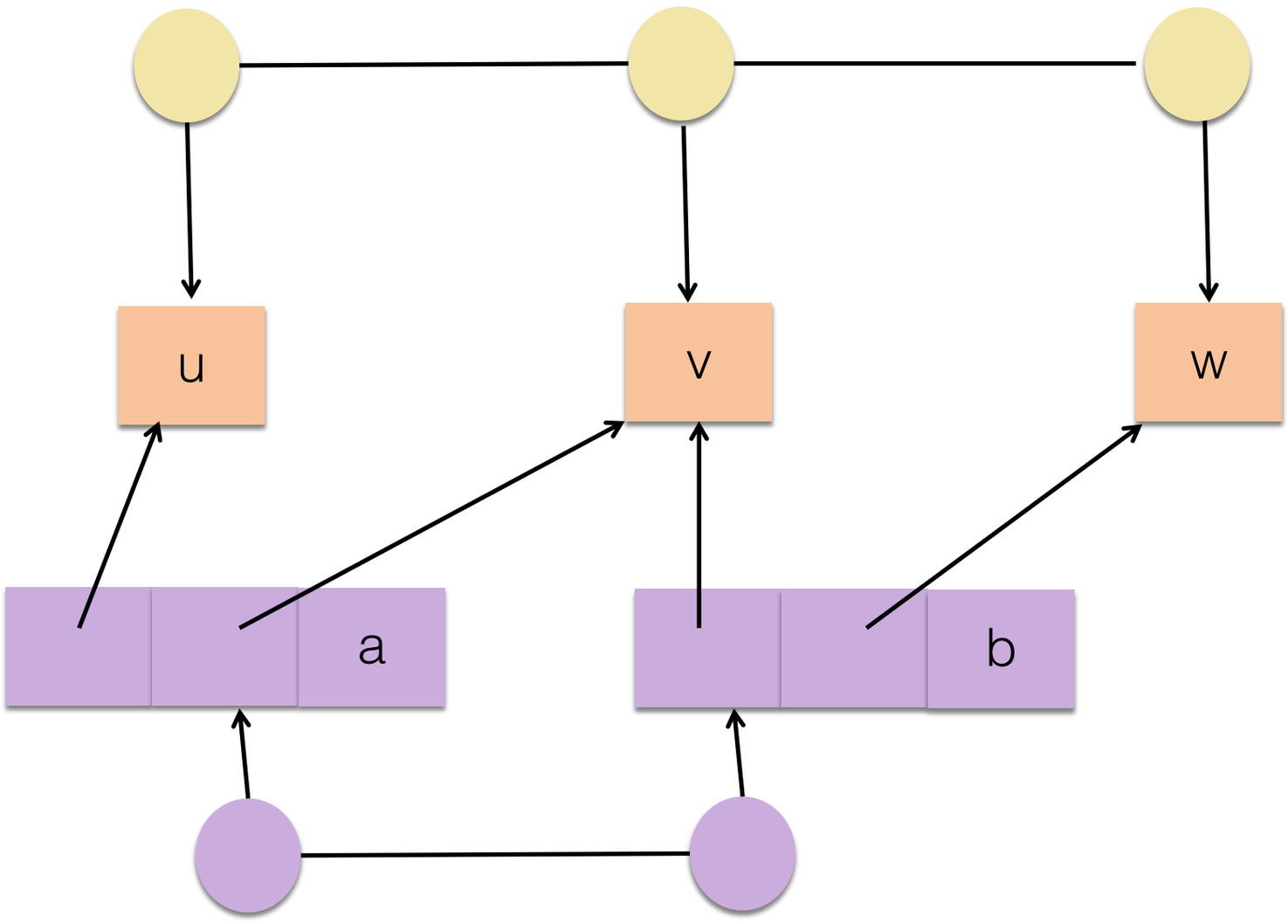
Edge List Structure



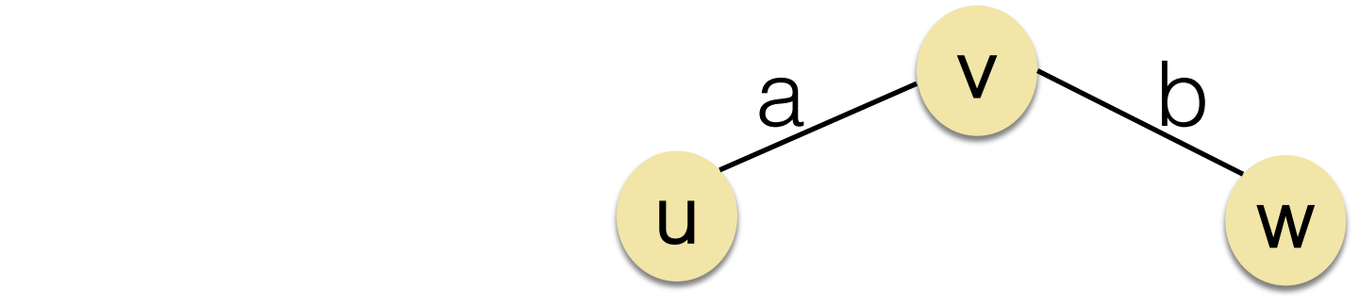
Edge List Structure



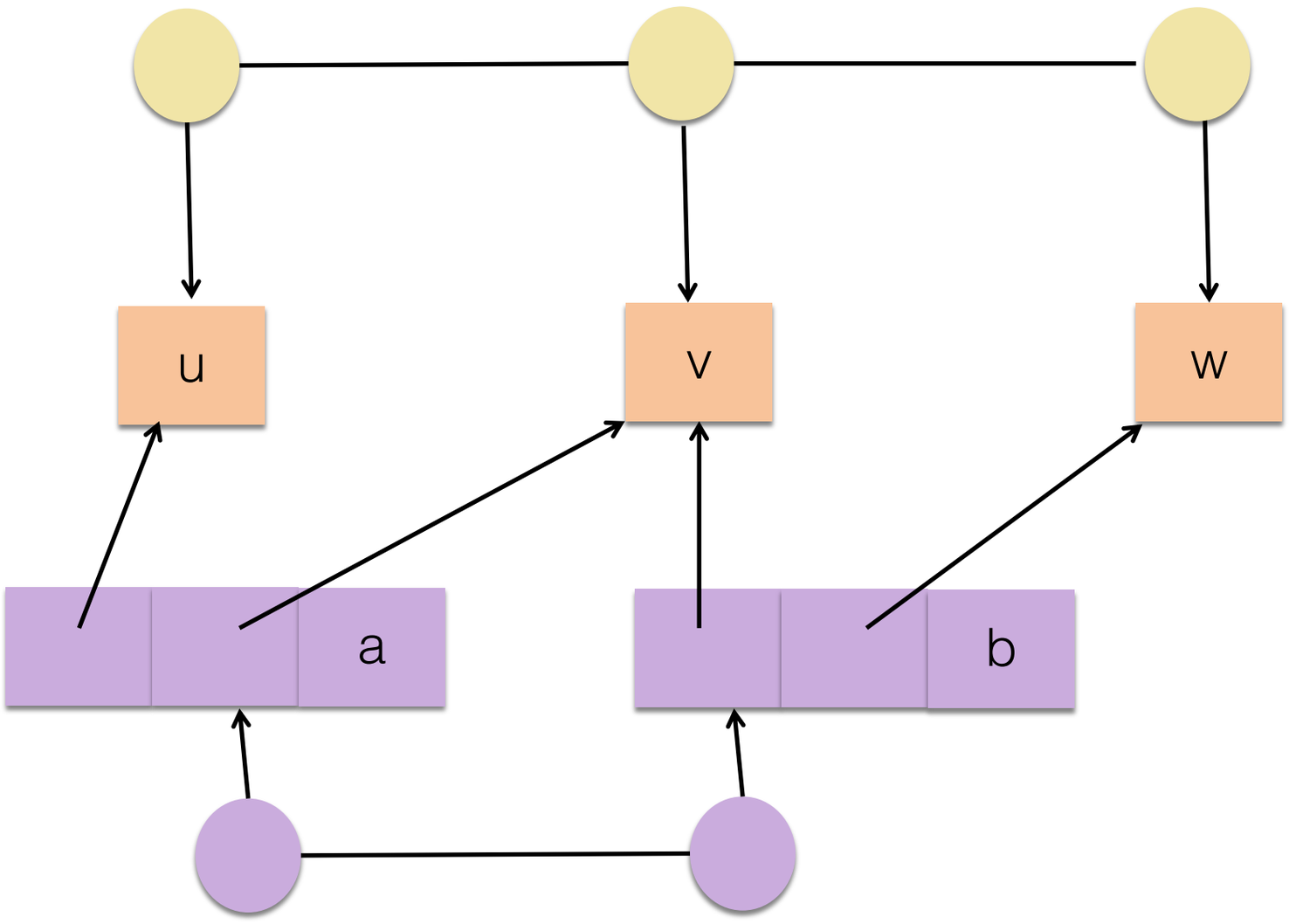
insertVertex(v)
O(1)



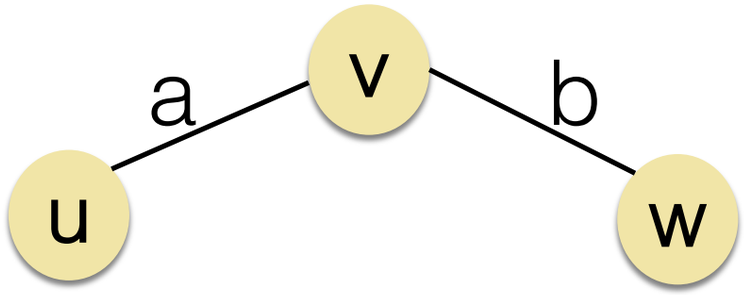
Edge List Structure



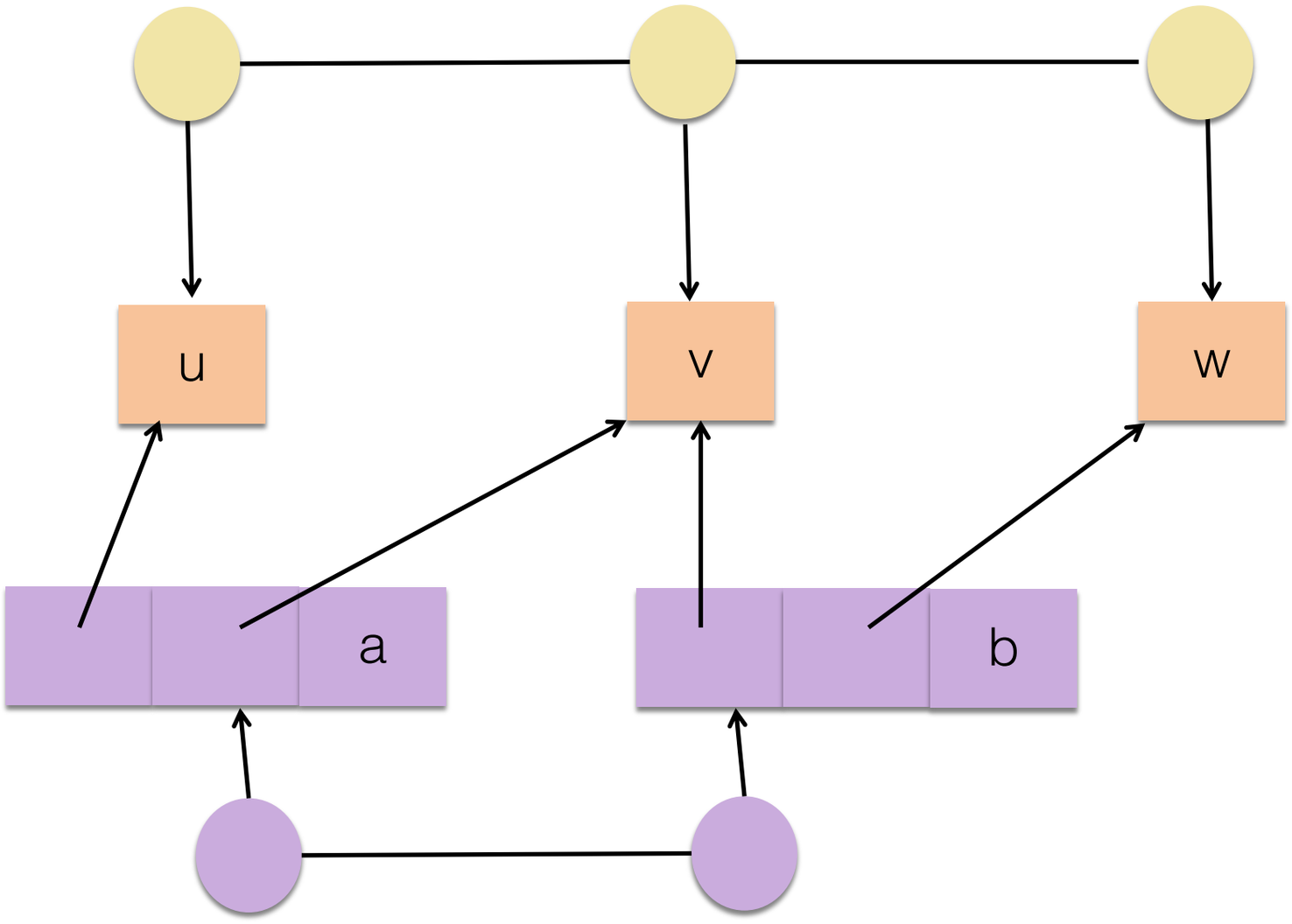
insertEdge(e, origin, dest)
O(1)



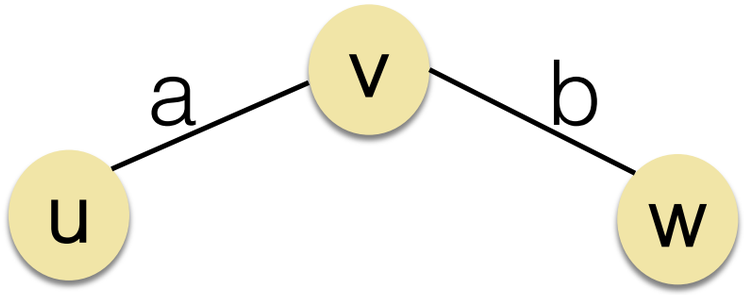
Edge List Structure



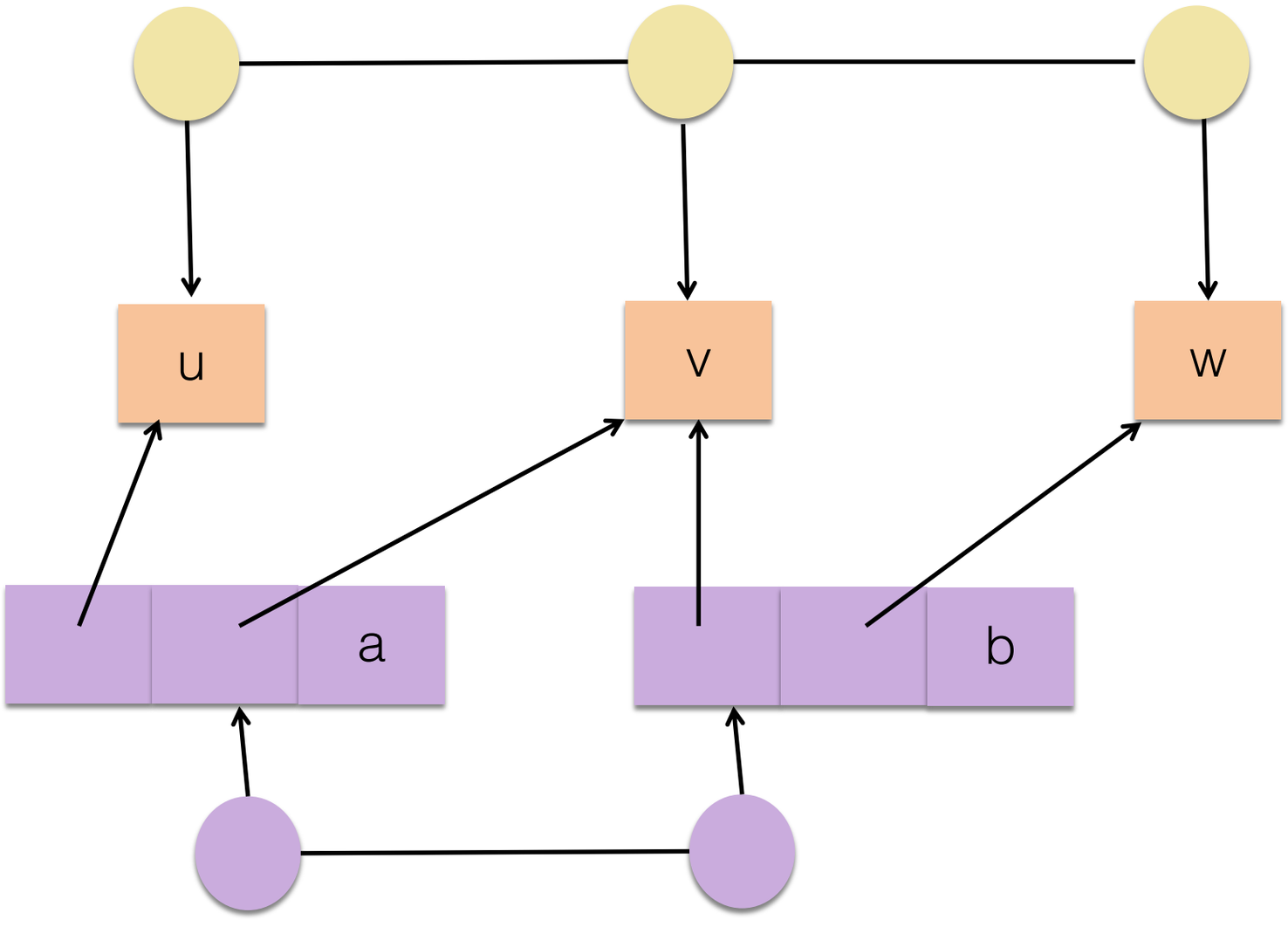
areAdjacent(v1, v2)
O(?)



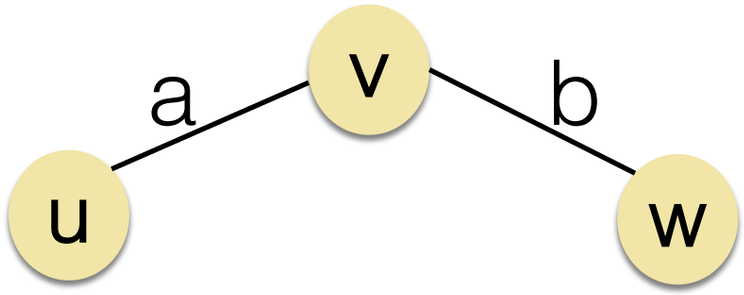
Edge List Structure



areAdjacent(v1, v2)
O(# of edges)

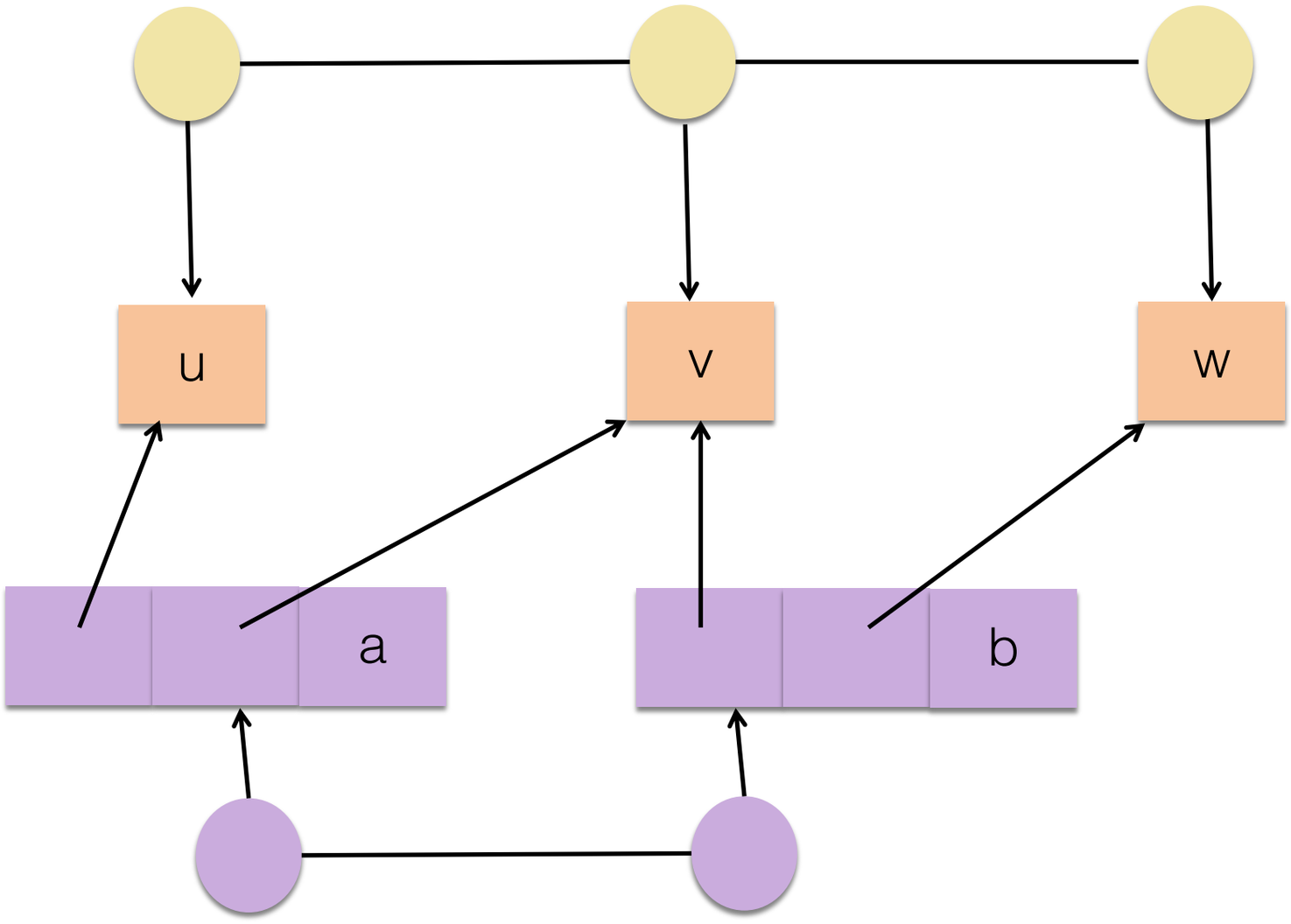


Edge List Structure

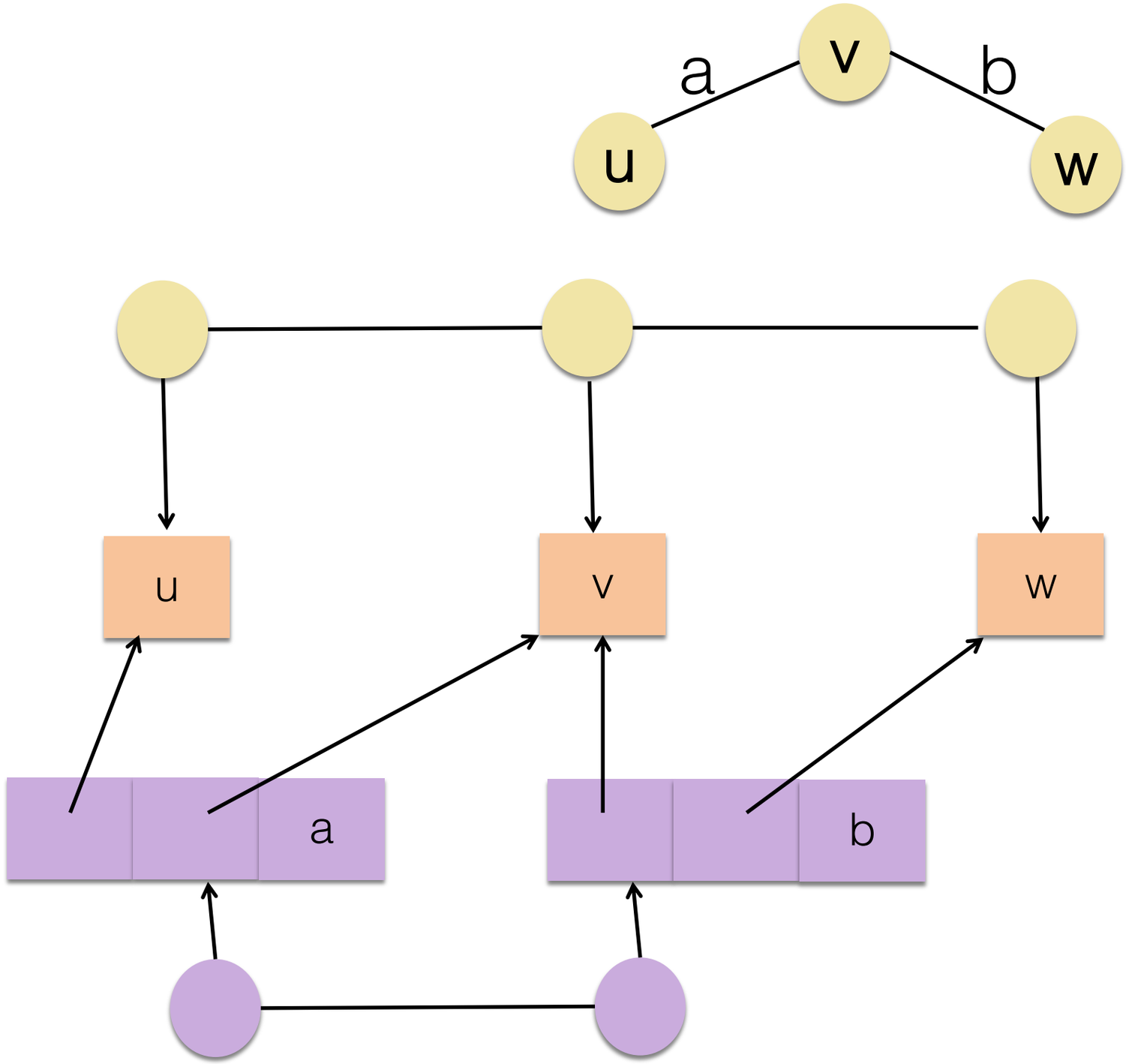


removeEdge(e)

O(# of edges)



Edge List Structure



Interesting one!

removeVertex(v)

O(?)

Edge List Structure

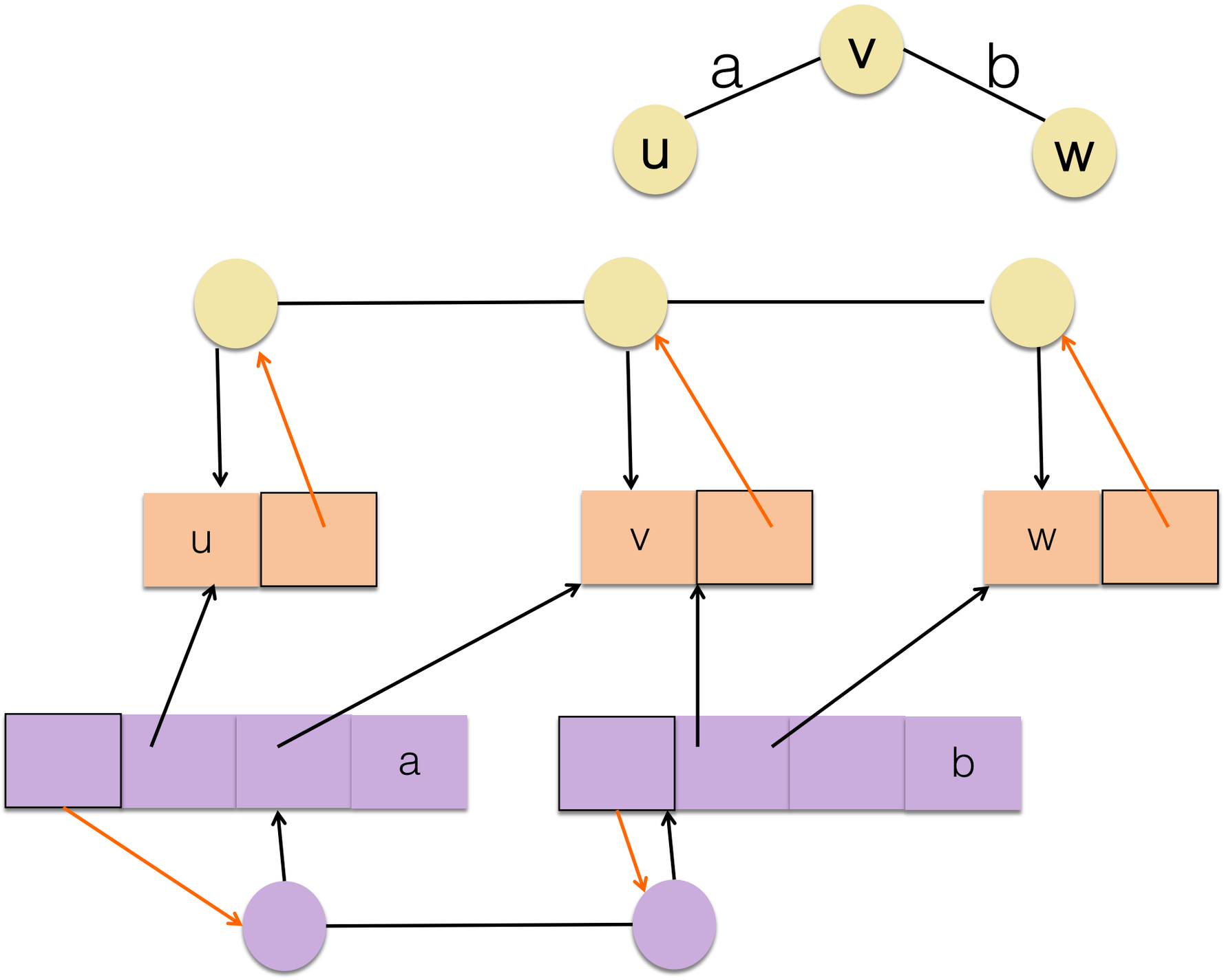
- Let's do one more change to improve the efficiency of removeVertex and removeEdge

- Let each vertex and edge object know where they are in their respective lists

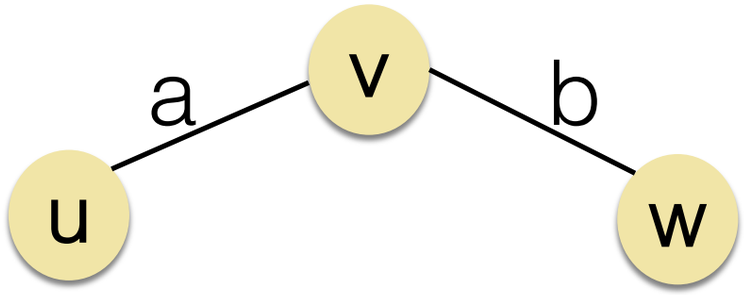
  - Vertex object: element, where am I in the vertex list

  - Edge Object: element, origin, destination, where am I in the edge list

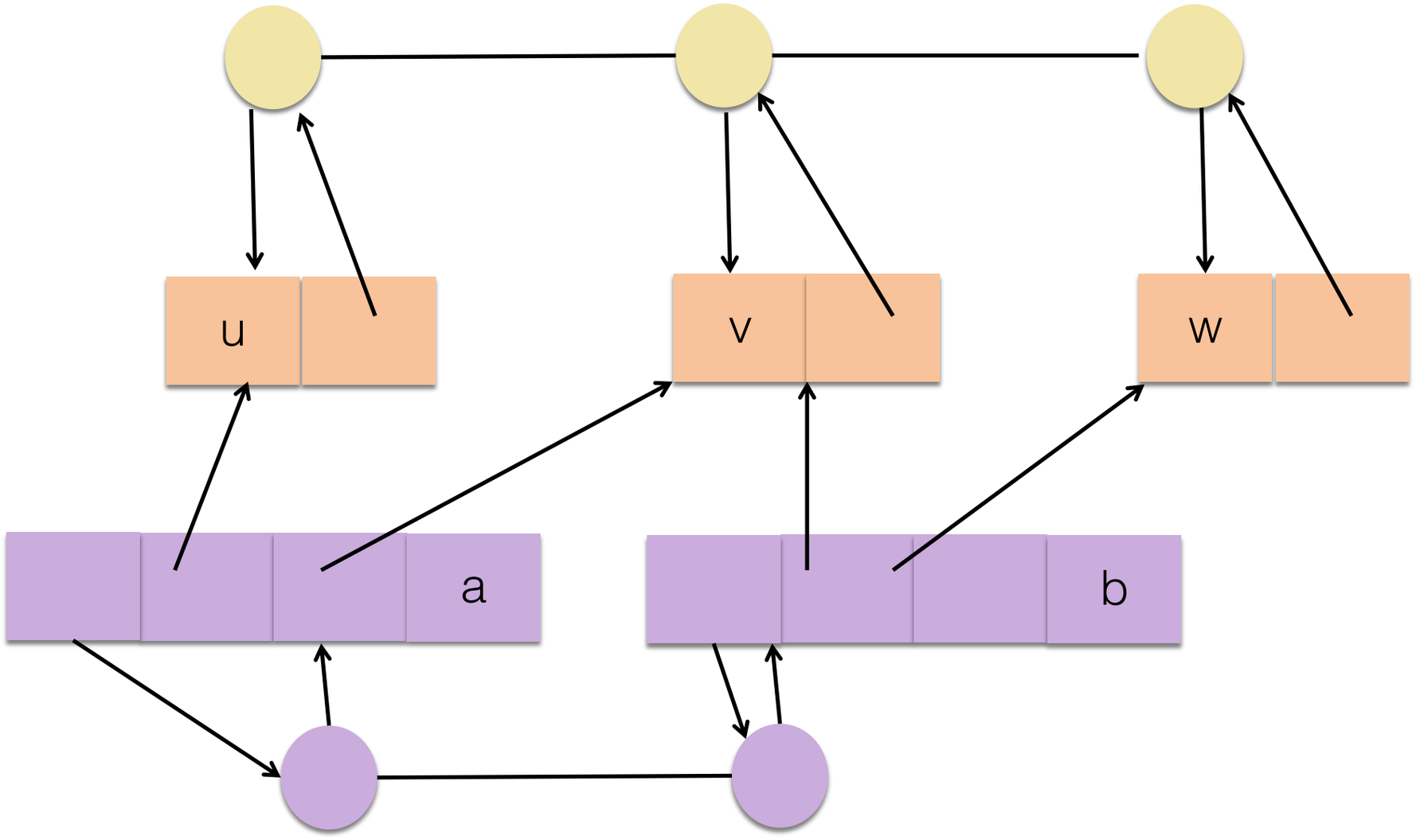
Edge List Structure



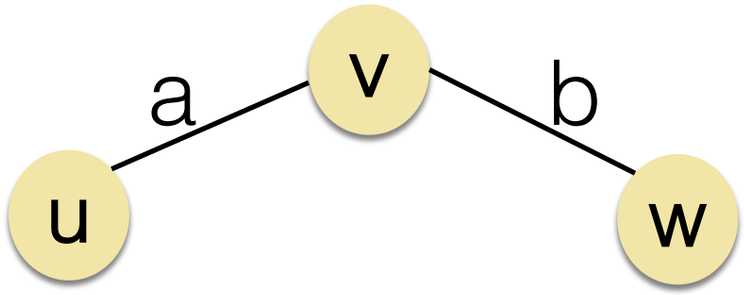
Edge List Structure



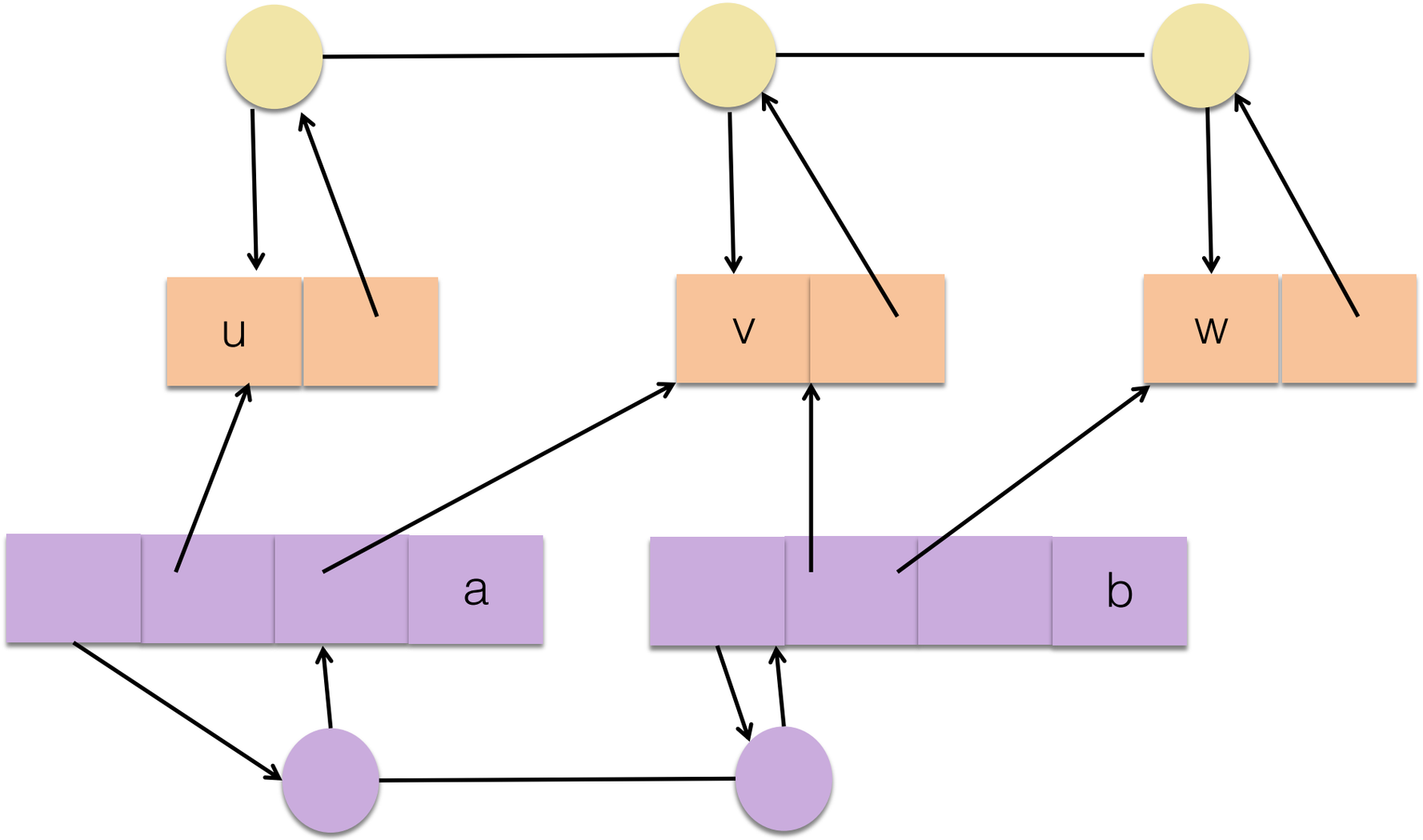
removeEdge(e)
O(1)



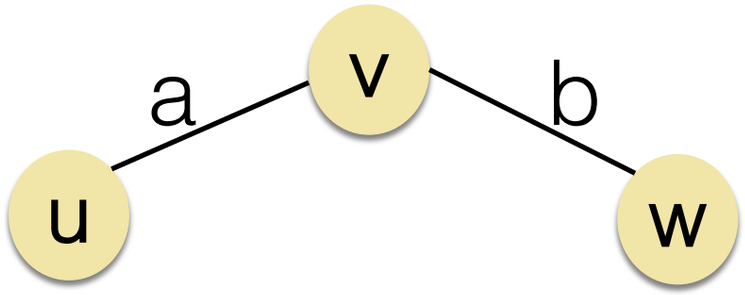
Edge List Structure



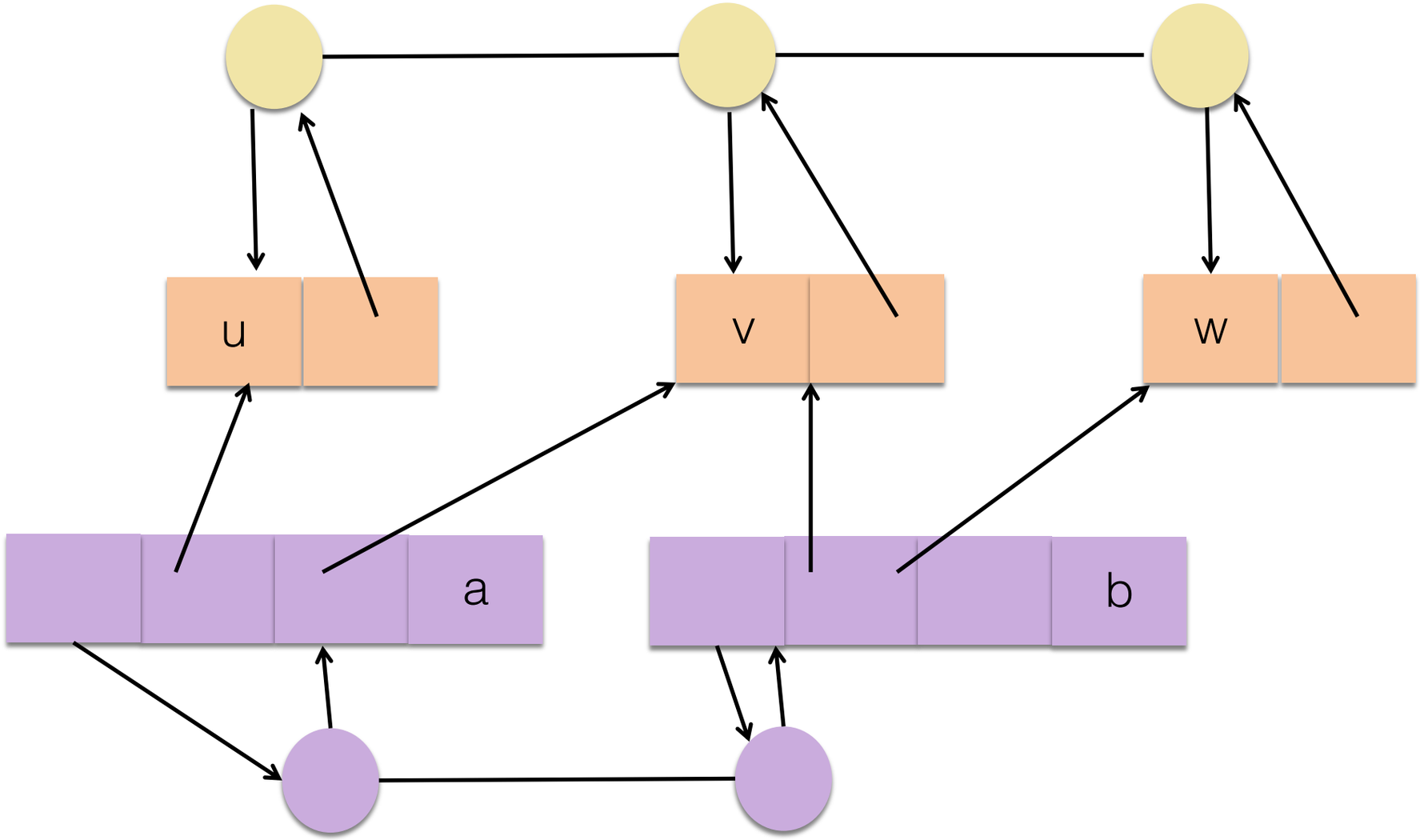
What about removeVertex(v)?



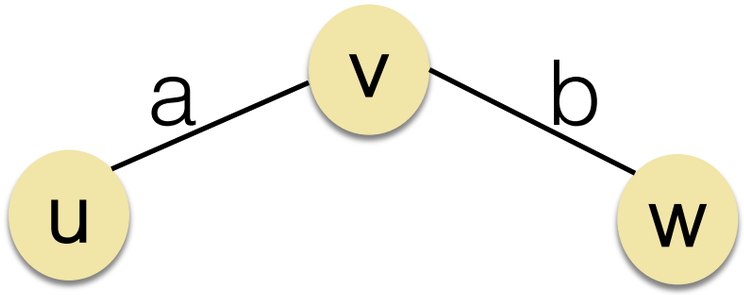
Edge List Structure



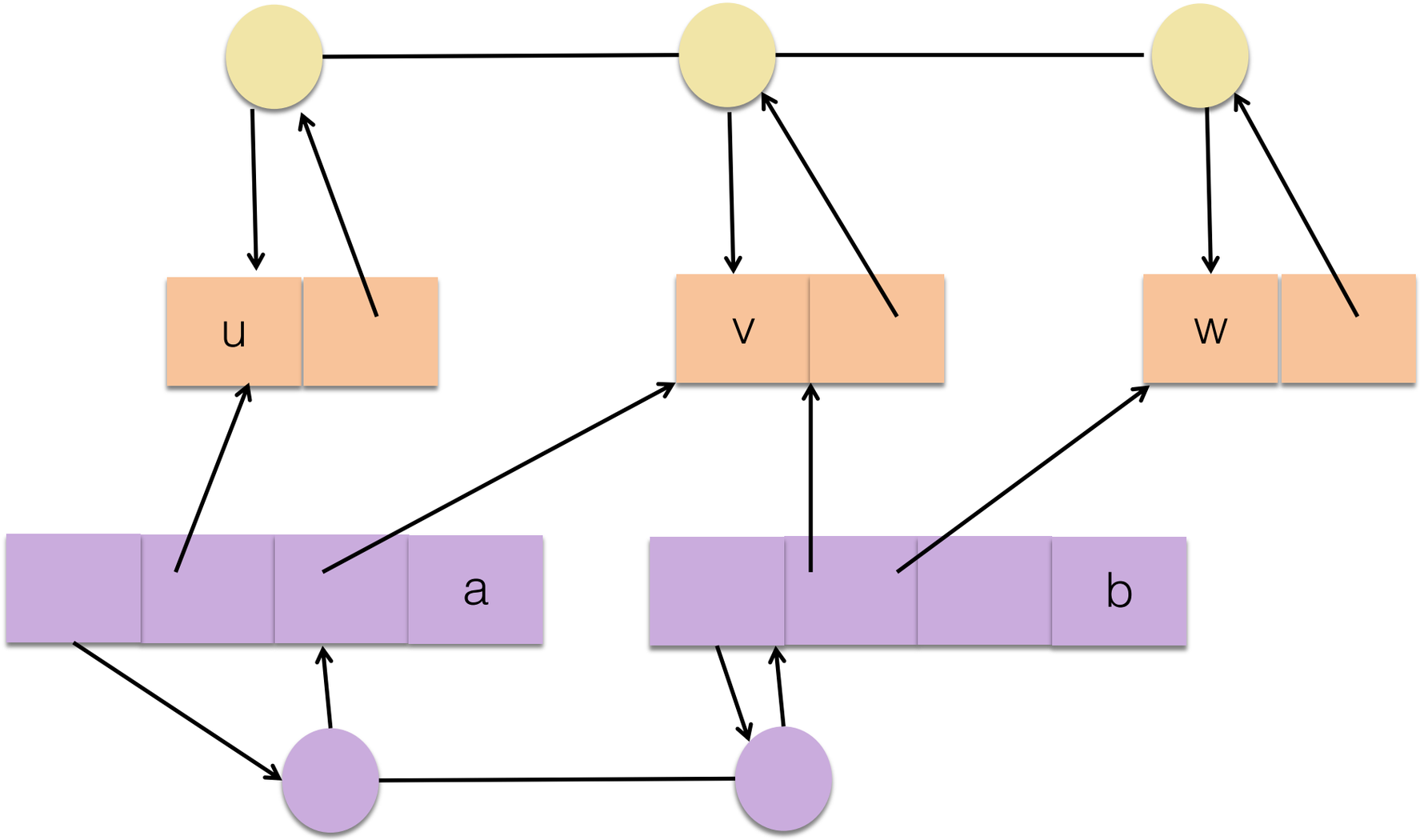
What about degree(v)?



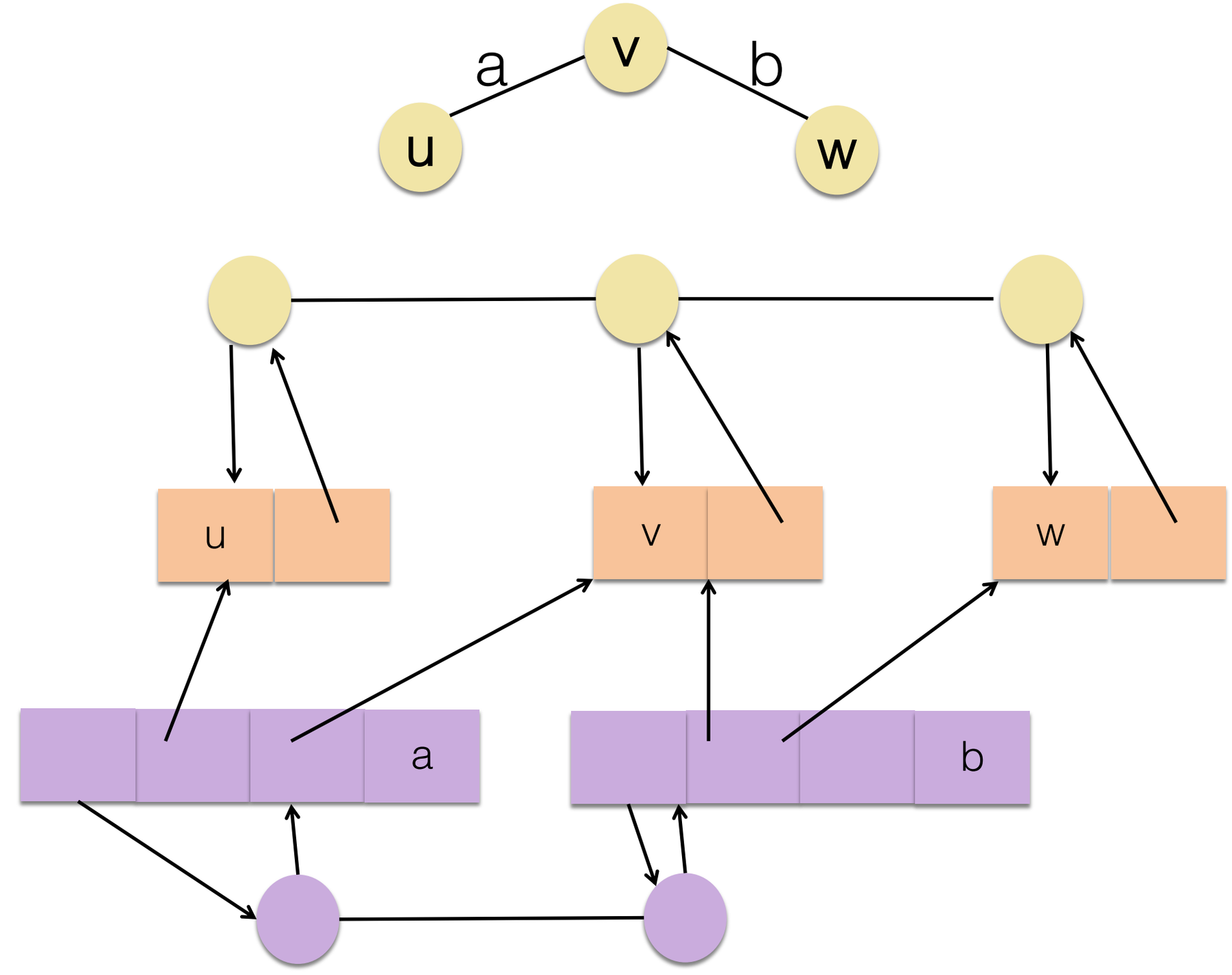
Edge List Structure



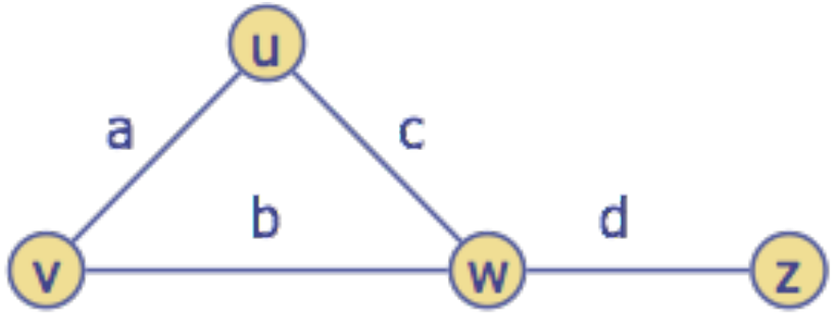
What about areAdjacent(v1, v2)?



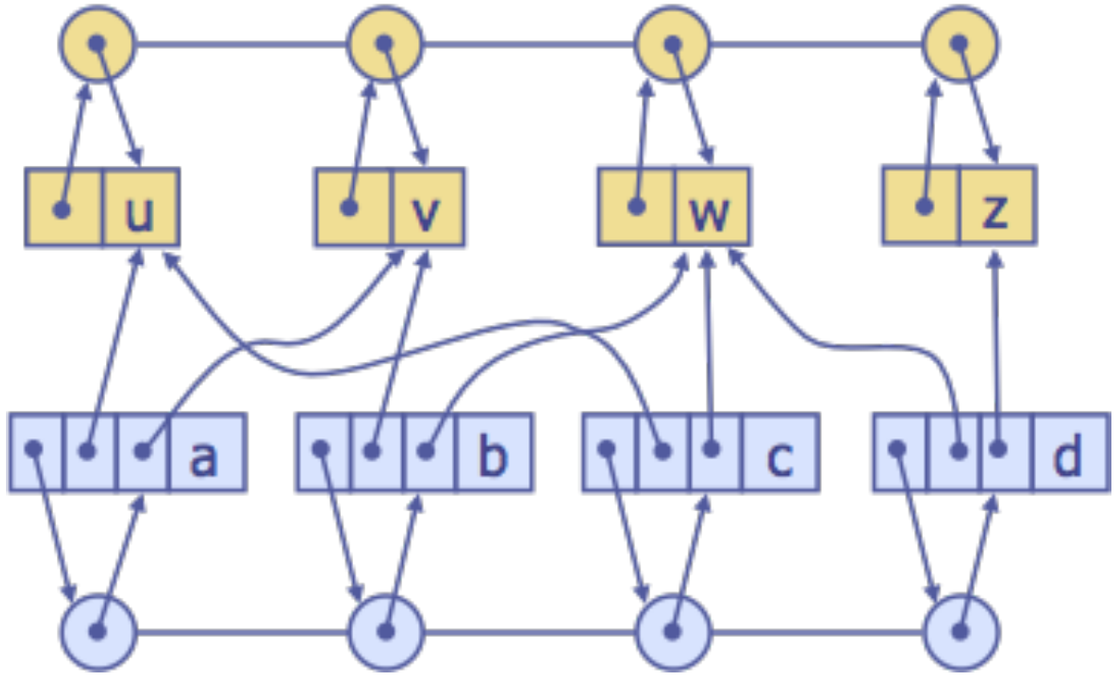
Summary: Edge List Structure



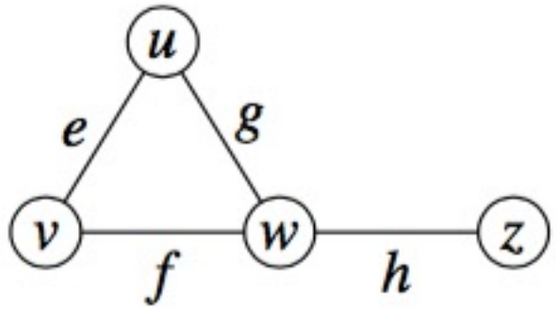
Example From Your Book



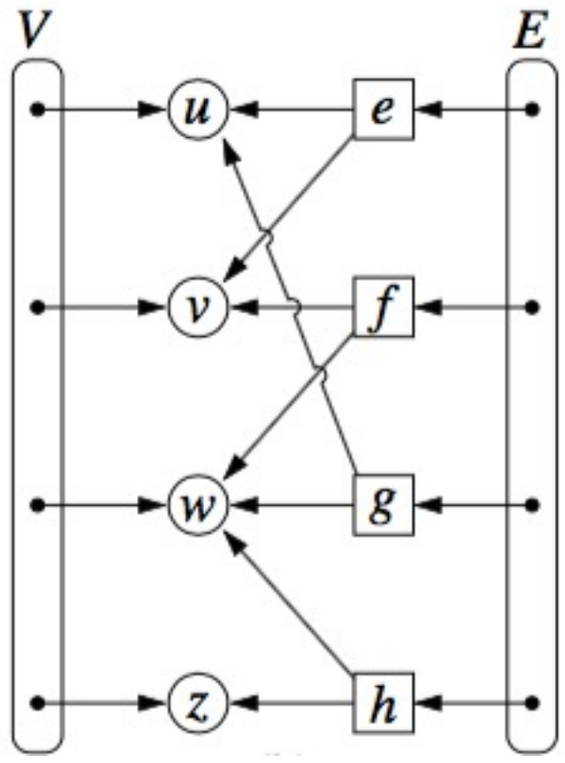
- **Goodrich (old version)**



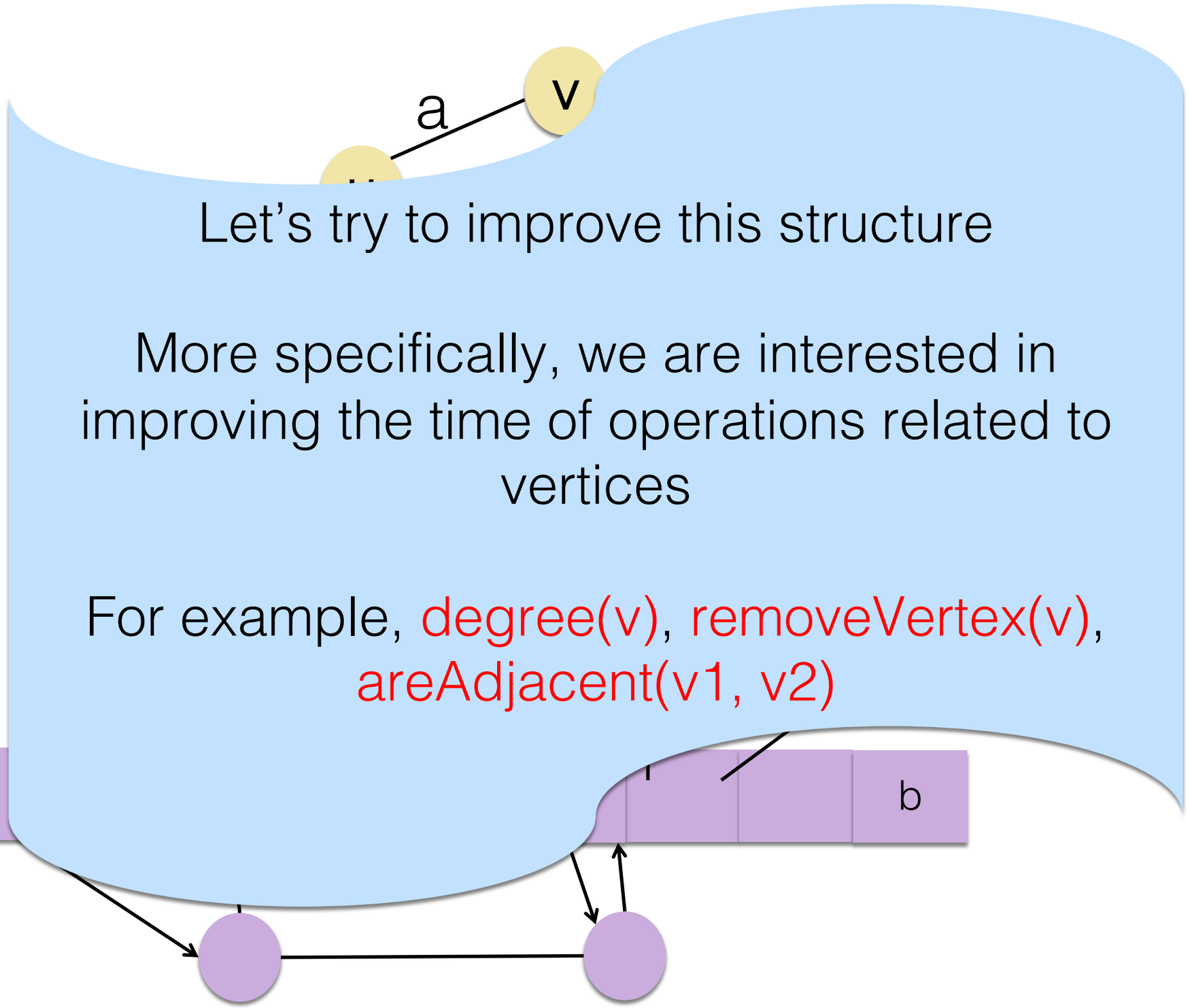
Example From Your Book



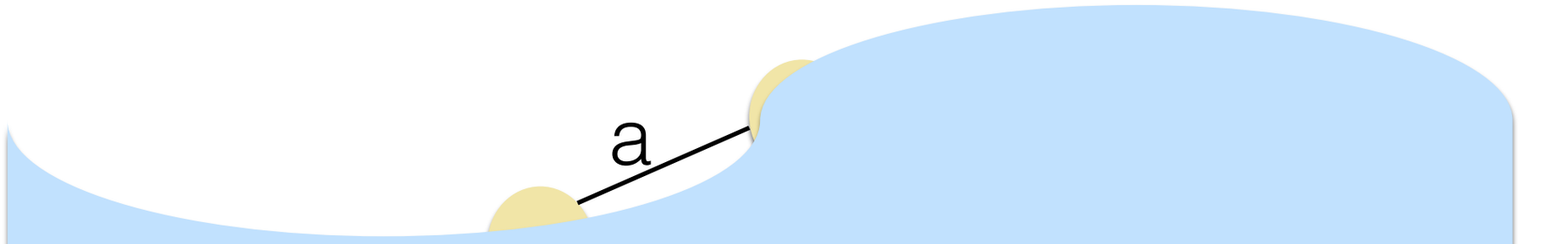
- **Goodrich (new version)**



Edge List Structure



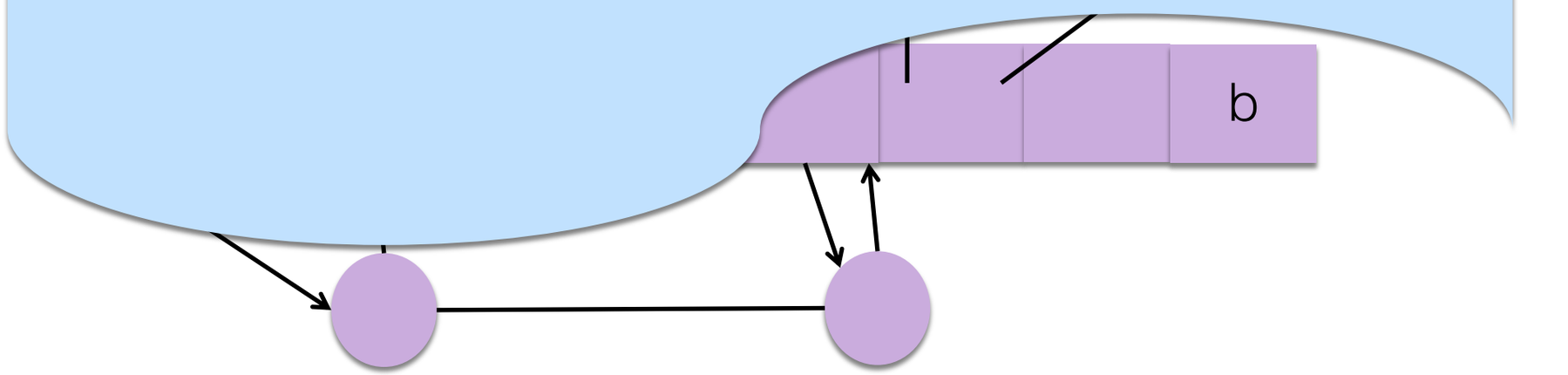
Improvement



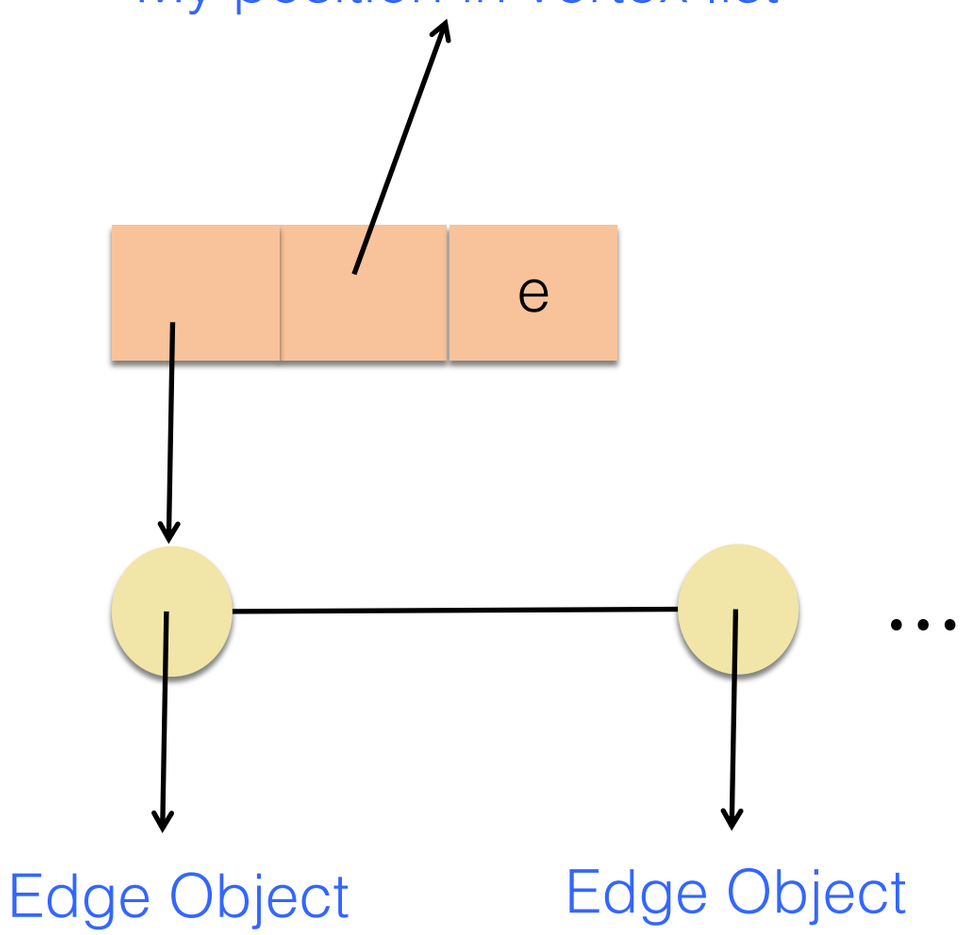
Vertex object is given more information

Each vertex now knows which edges are incident on it!

This information is stored as a LIST of pointers pointing to incident edges



Arbitrary Vertex Object with Two Incident Edges

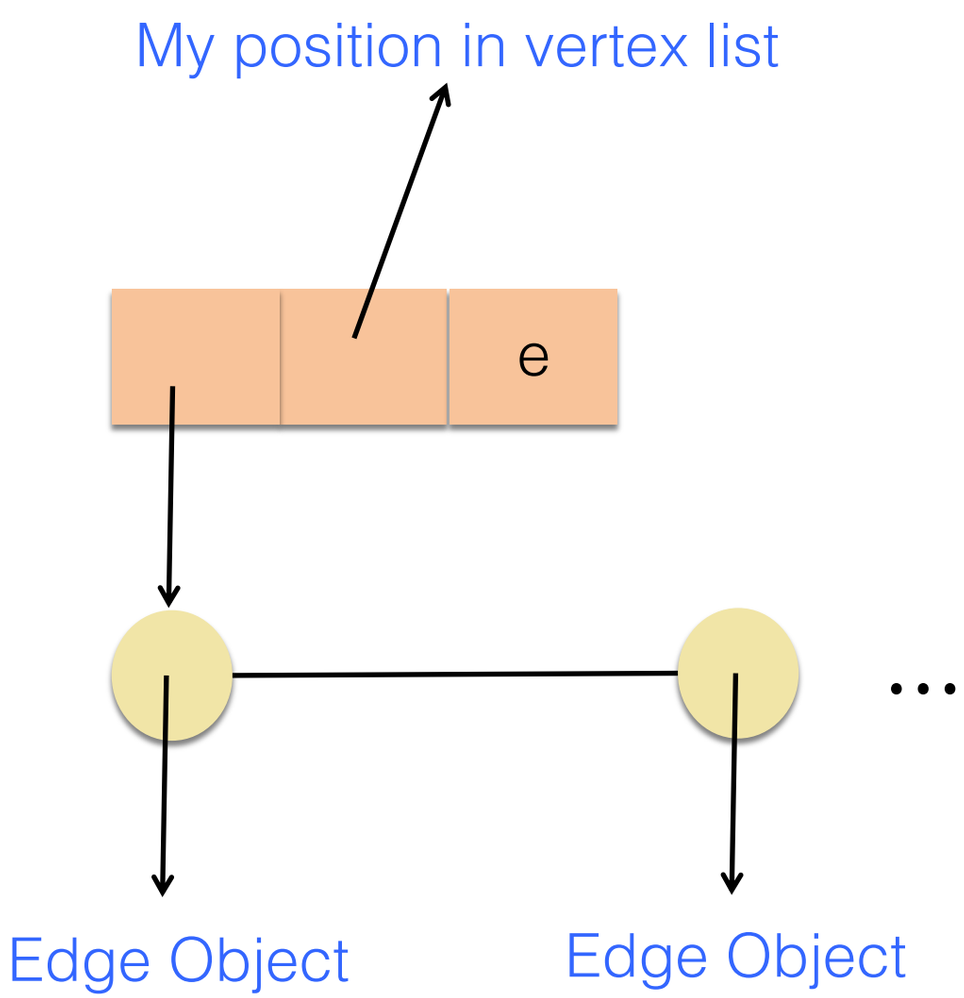


Vertex object is given more information

Each vertex now knows which edges are incident on it!

This information is stored as a LIST of pointers pointing to incident edges – called **Adjacency List** – why name it like that?

Let’s Analyze

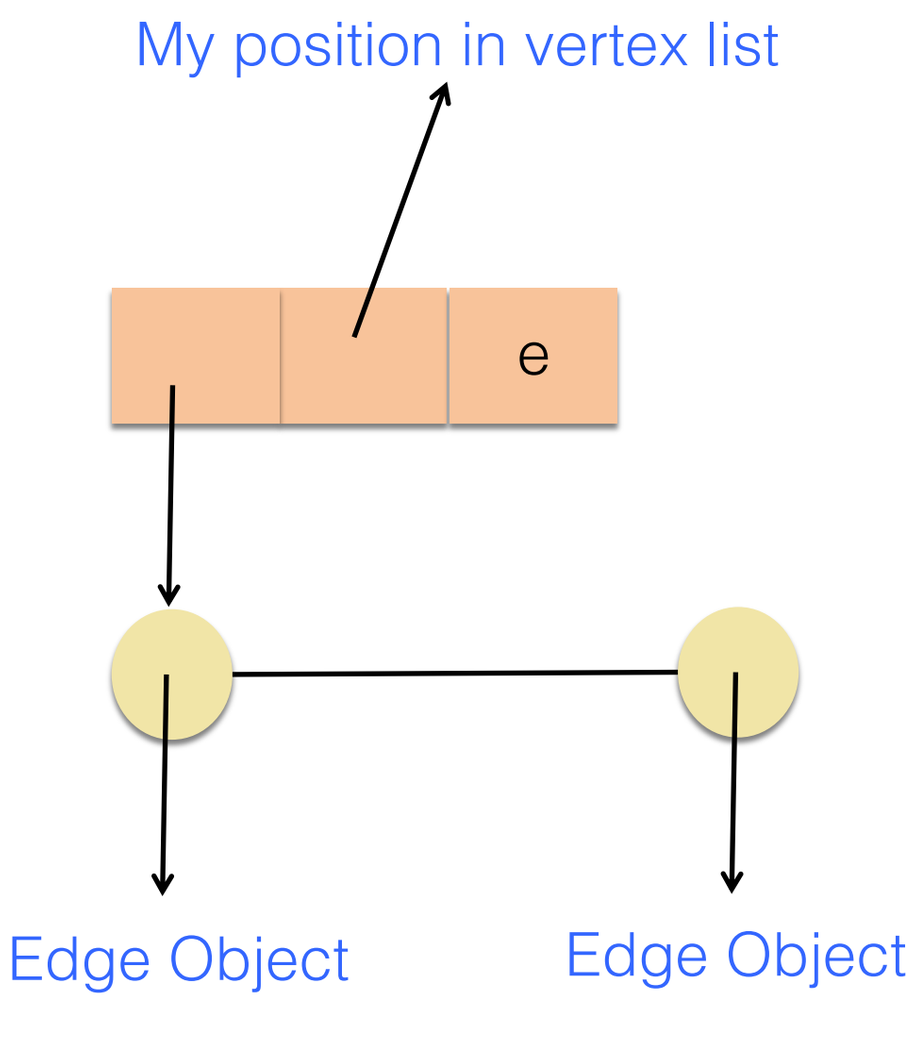


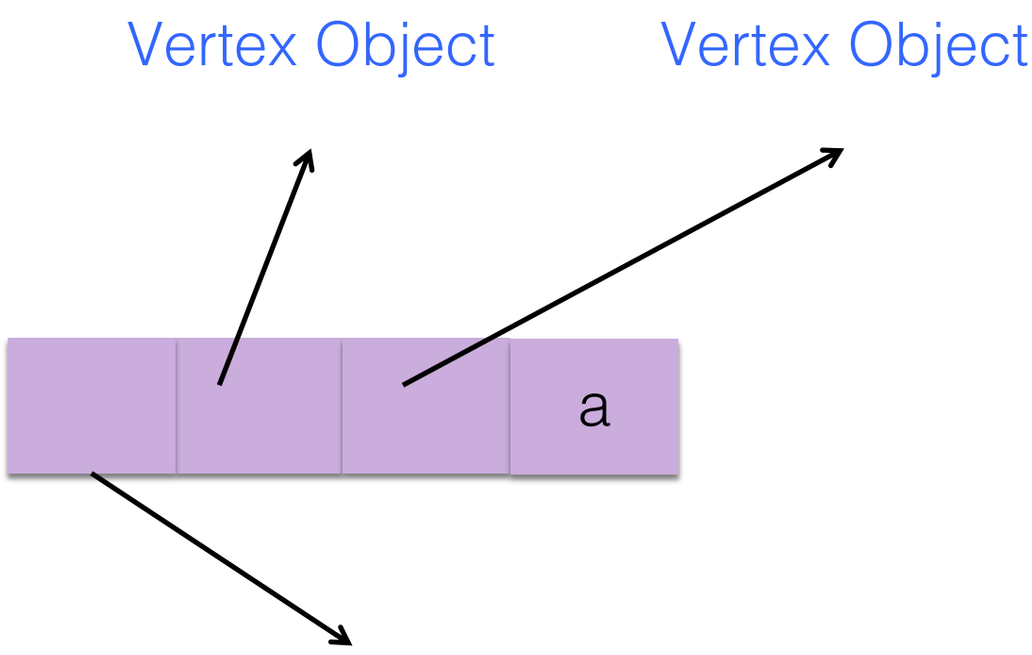
degree(v) ?

areAdjacent(v1, v2) ?

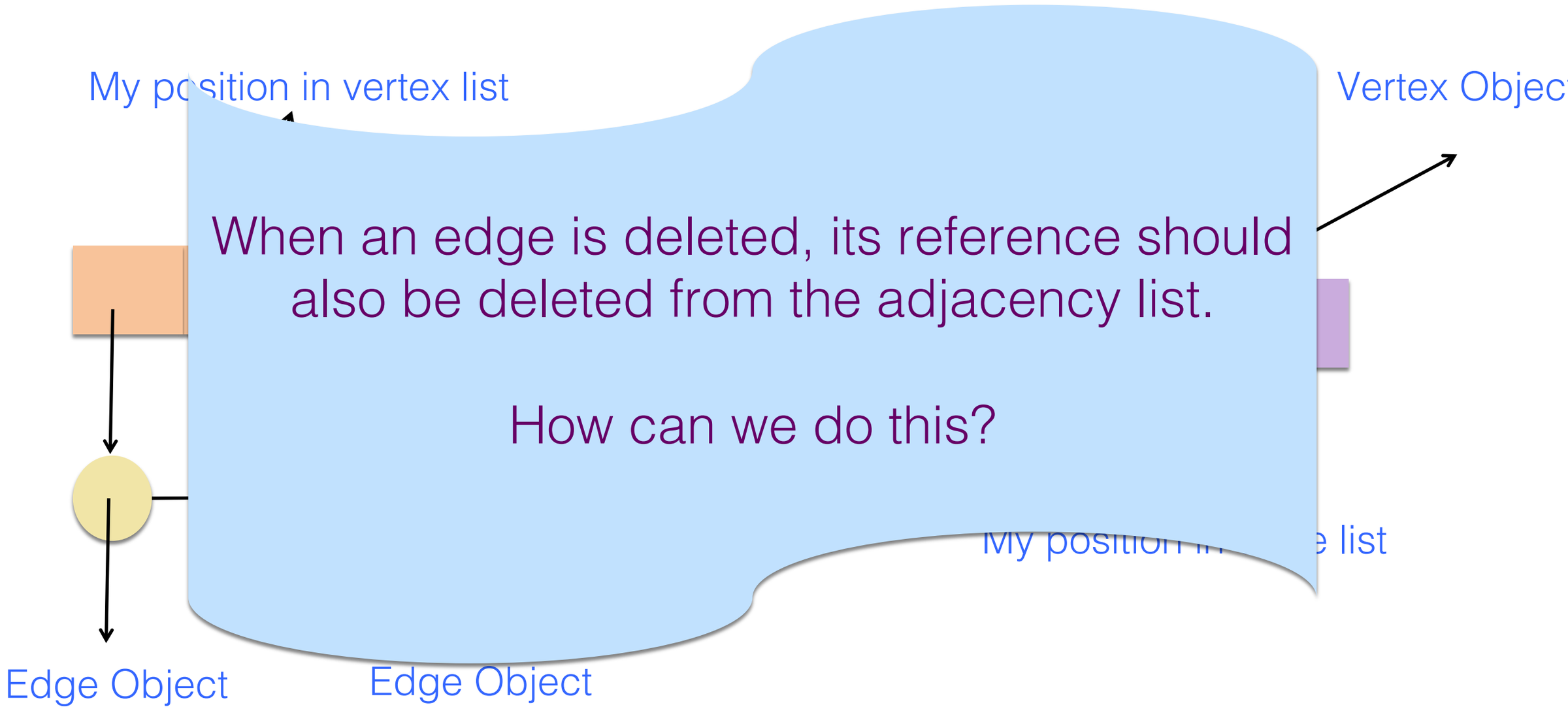
removeVertex(v) ?

New Vertex with Previous Edge Object

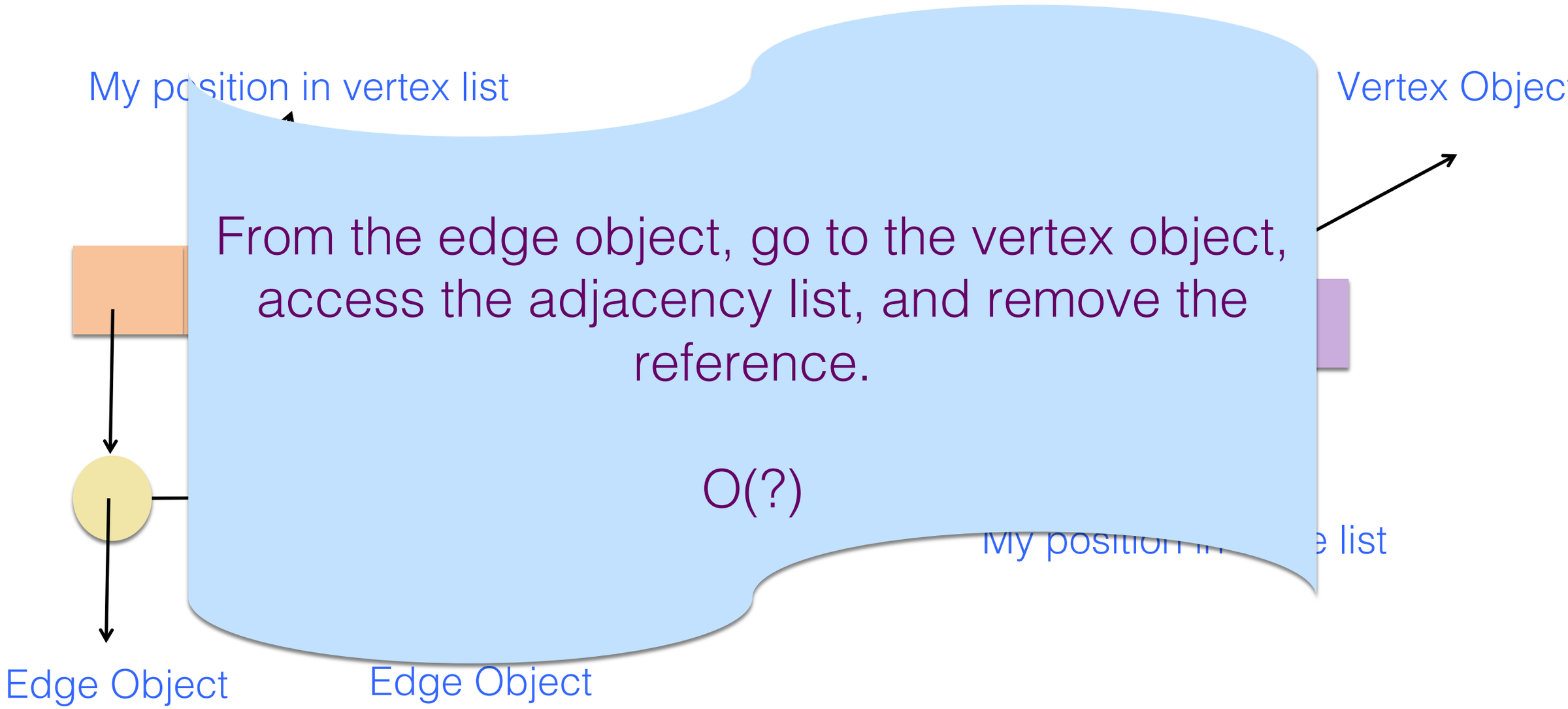




Edge Removal

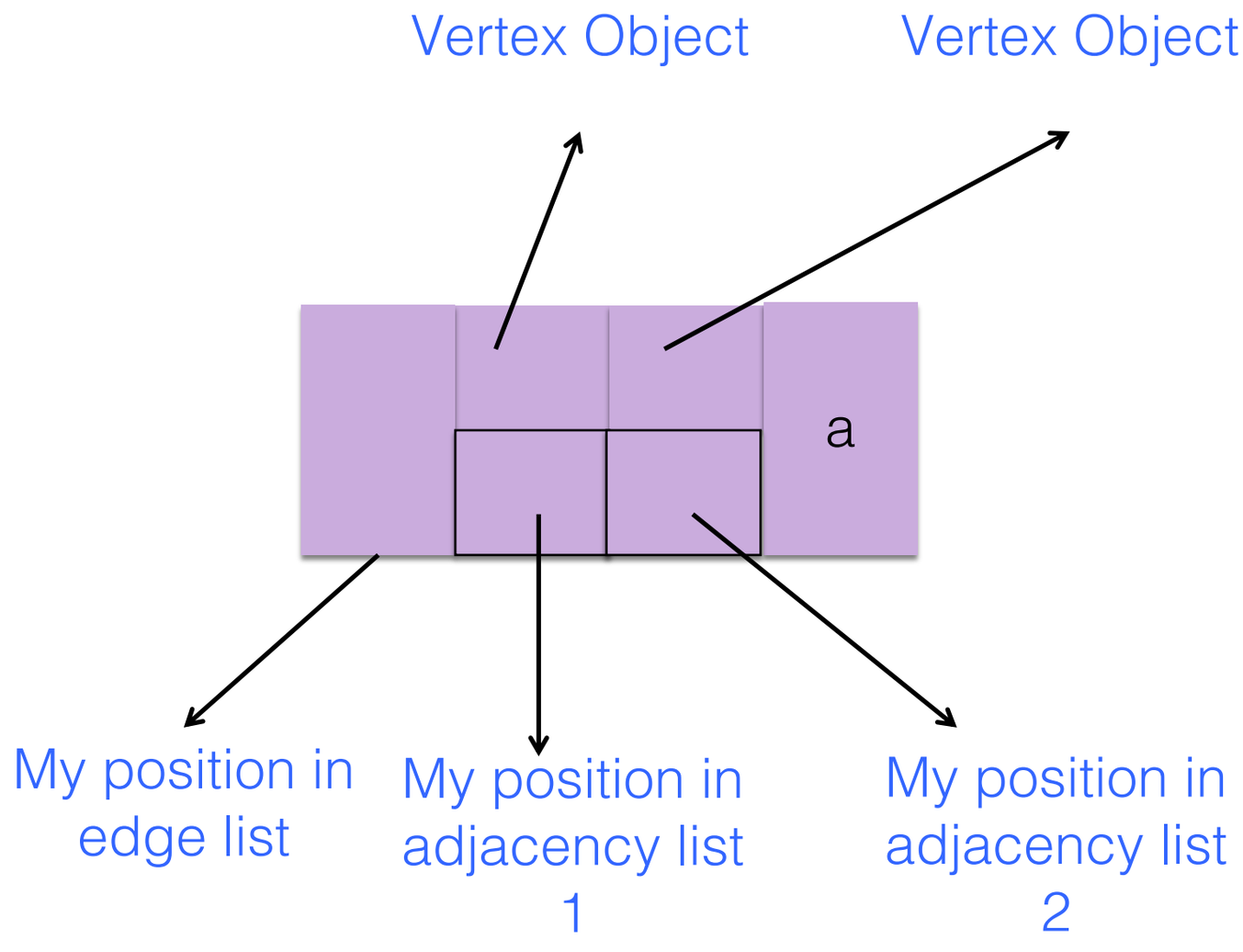


Removing Edge Reference from Adjacency List

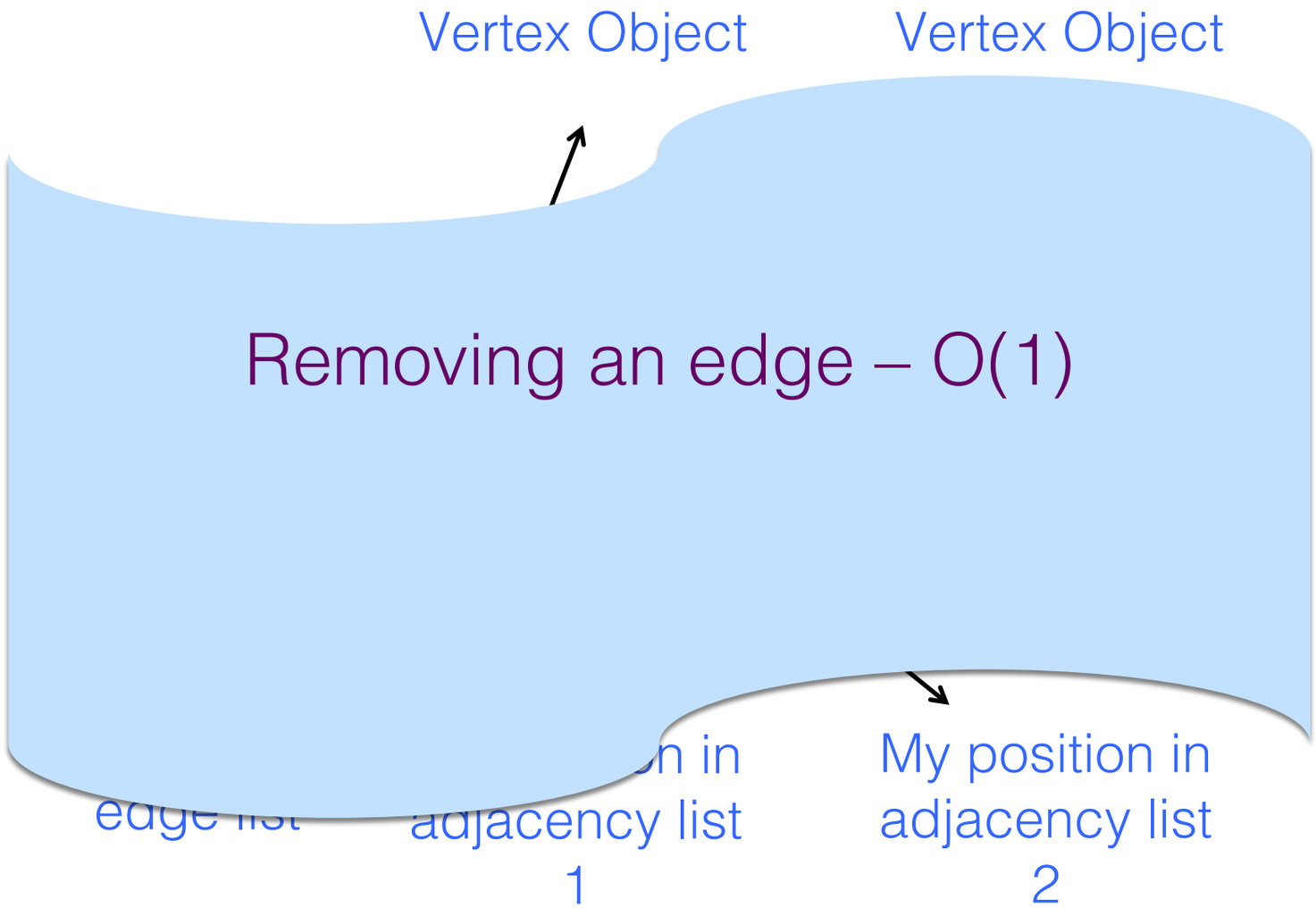


Removing Edge Reference from Adjacency List (2)

Updated Edge Object

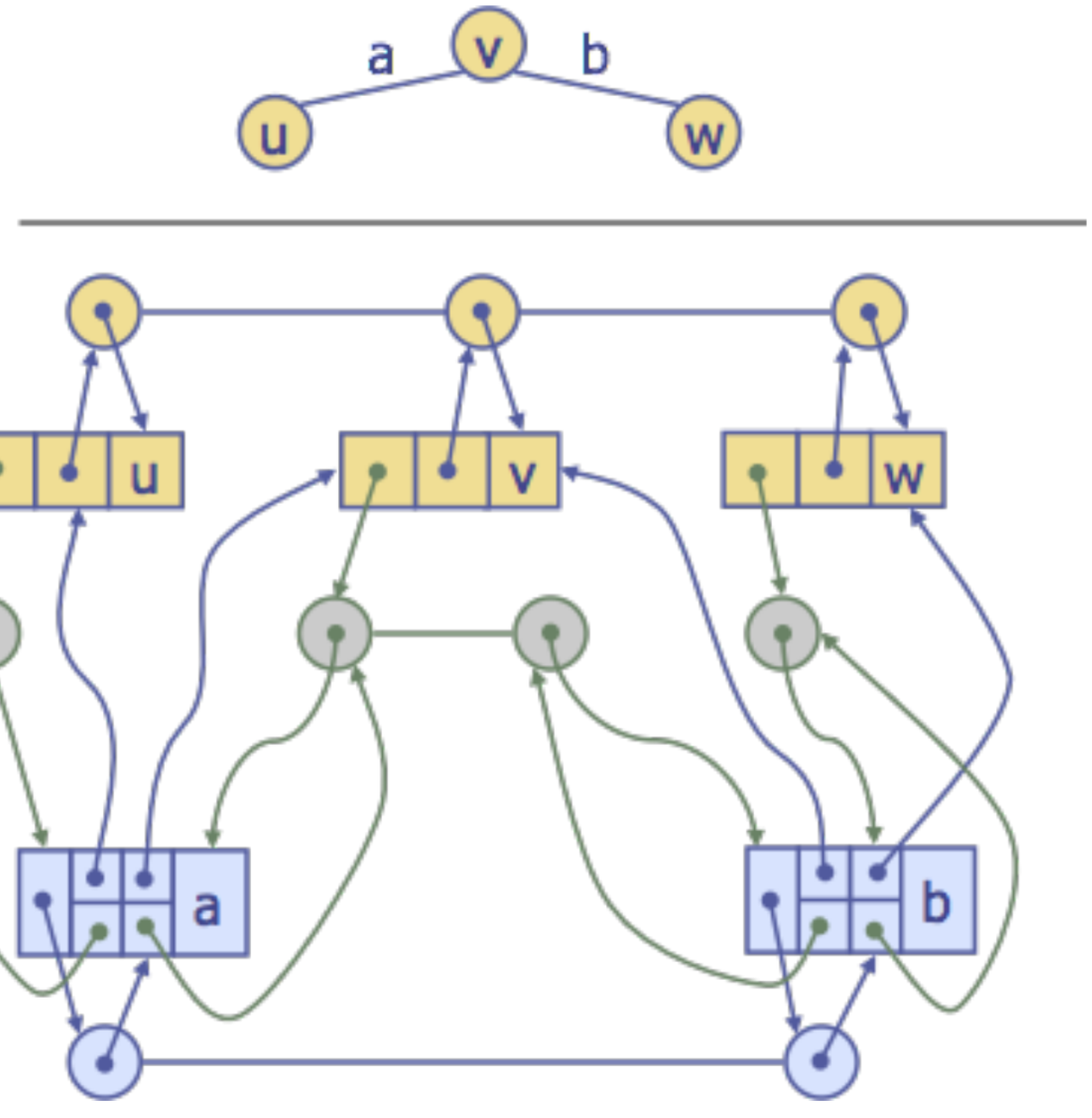


Updated Edge Object (2)

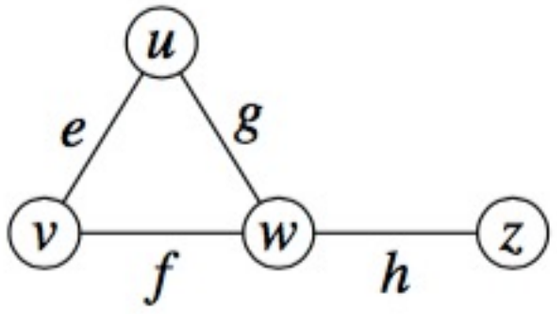


Adjacency List Structure

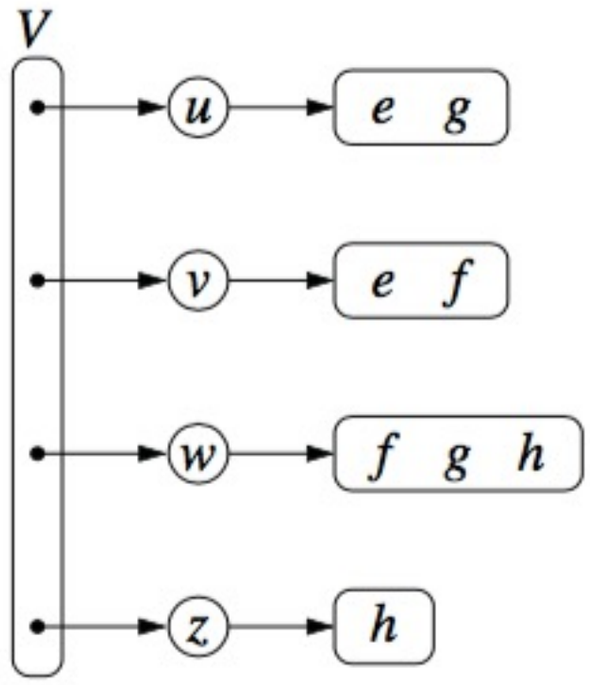
- **Complete structure, Example from the book (old)**



Adjacency List Structure



- **Incomplete structure, Example from the book (New)**

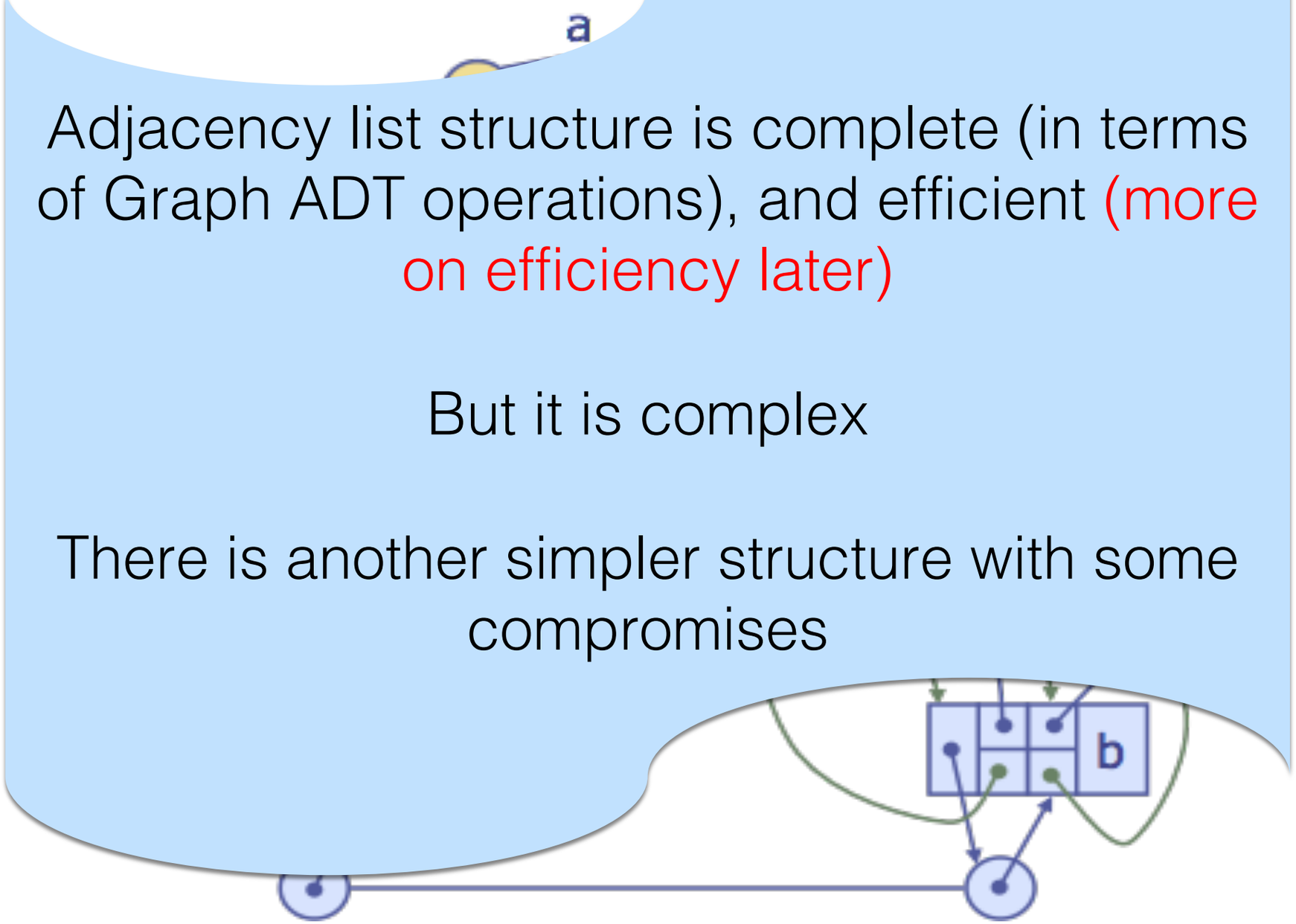


Adjacency List Structure (2)

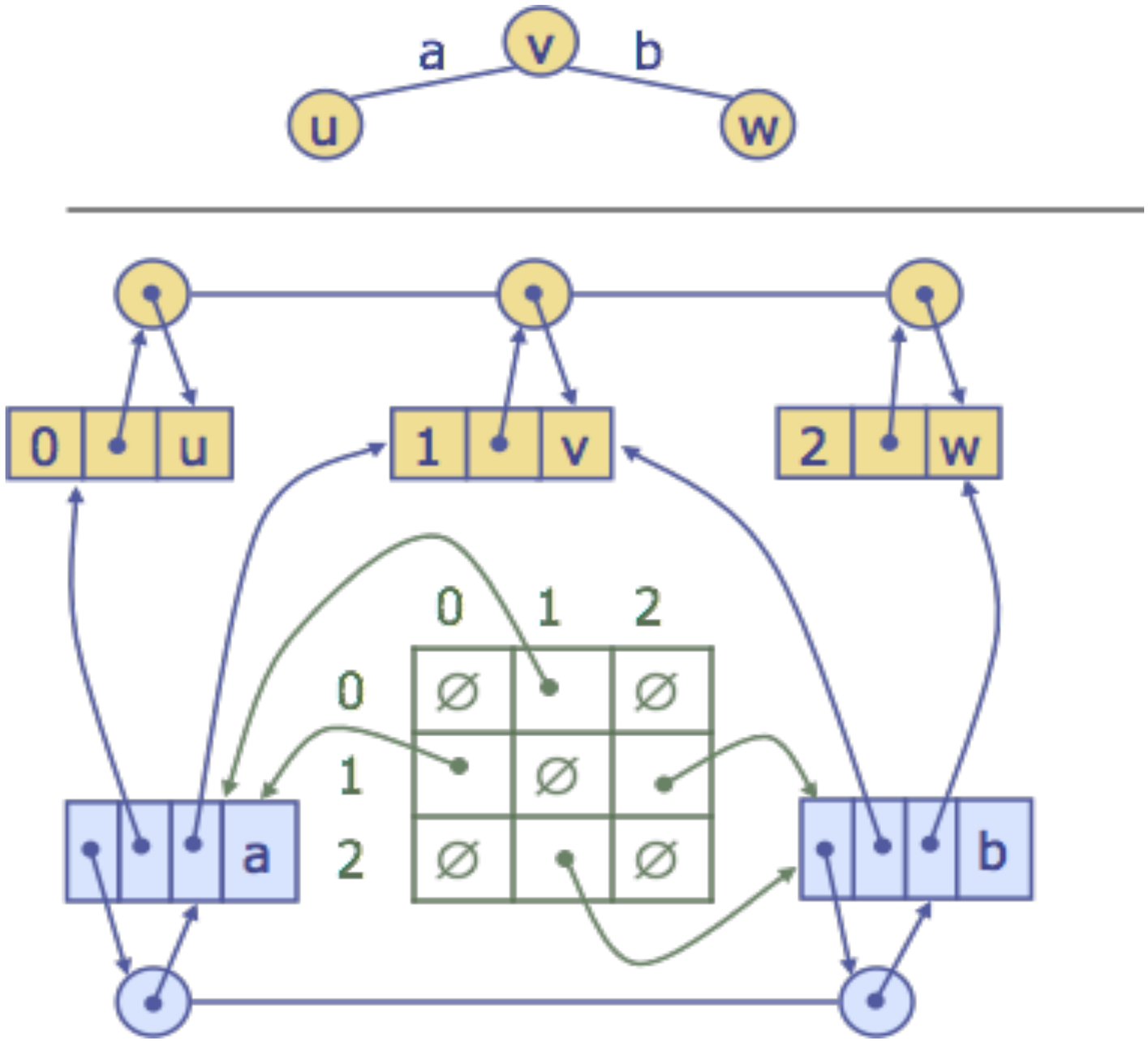
Adjacency list structure is complete (in terms of Graph ADT operations), and efficient (more on efficiency later)

But it is complex

There is another simpler structure with some compromises



Adjacency Matrix Structure

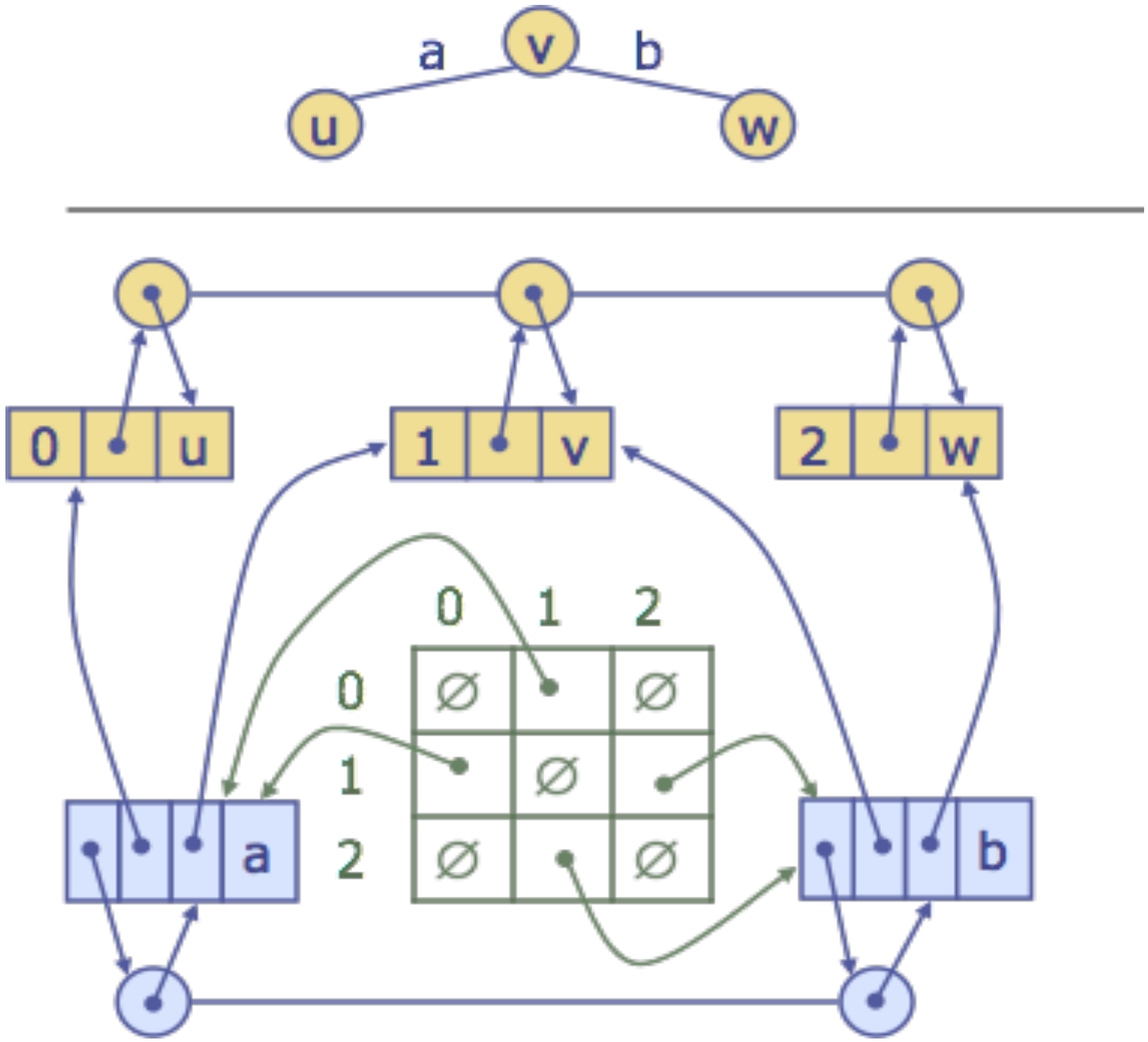


A nxn matrix is introduced – Adjacency Matrix

Each cell stores a reference to an edge object

Cell (0,1) refers to the edge between vertex0 and vertex1

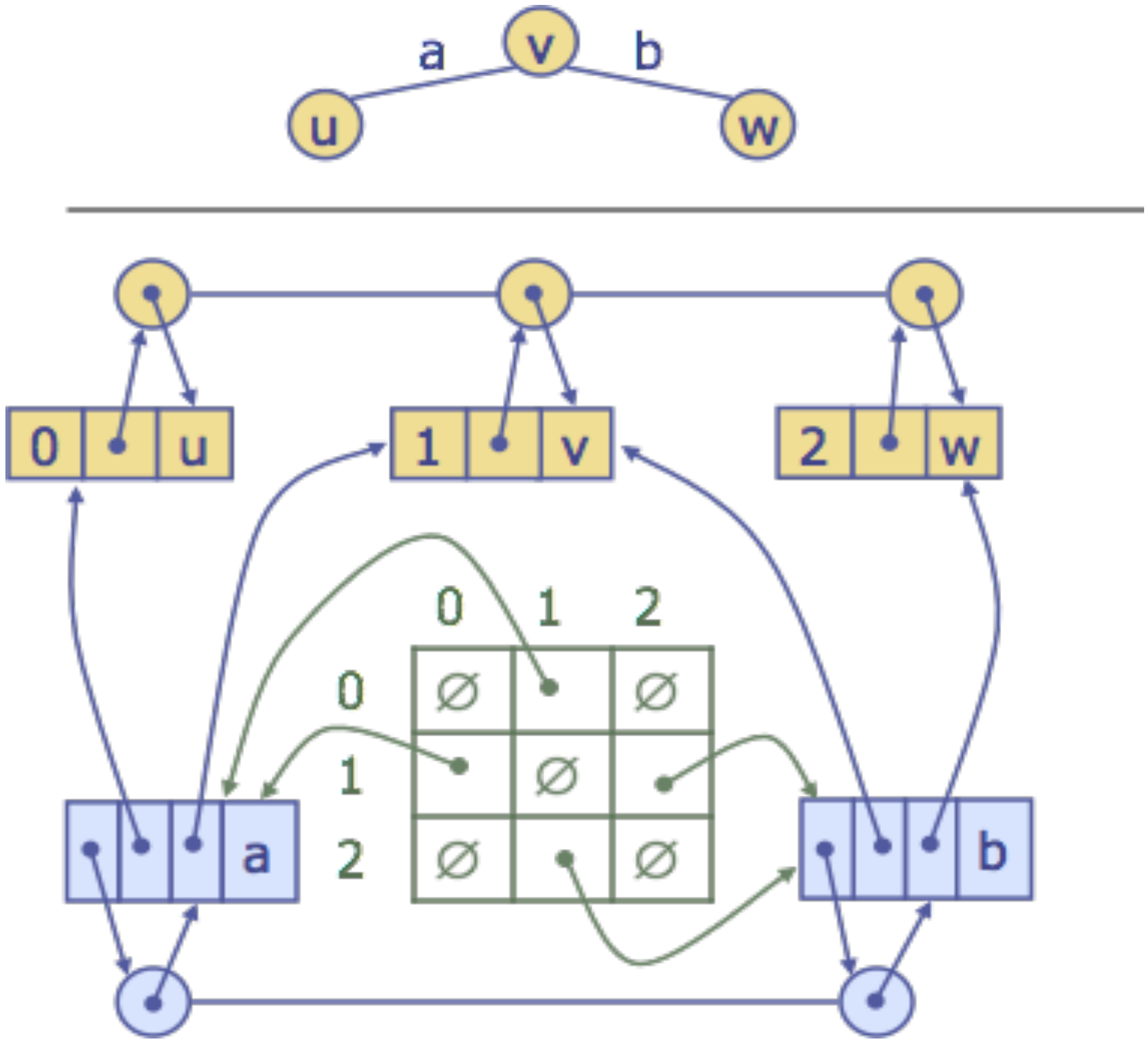
Adjacency Matrix Structure (2)



Edge object goes back to "Edge List" stage!

Only knows four things!

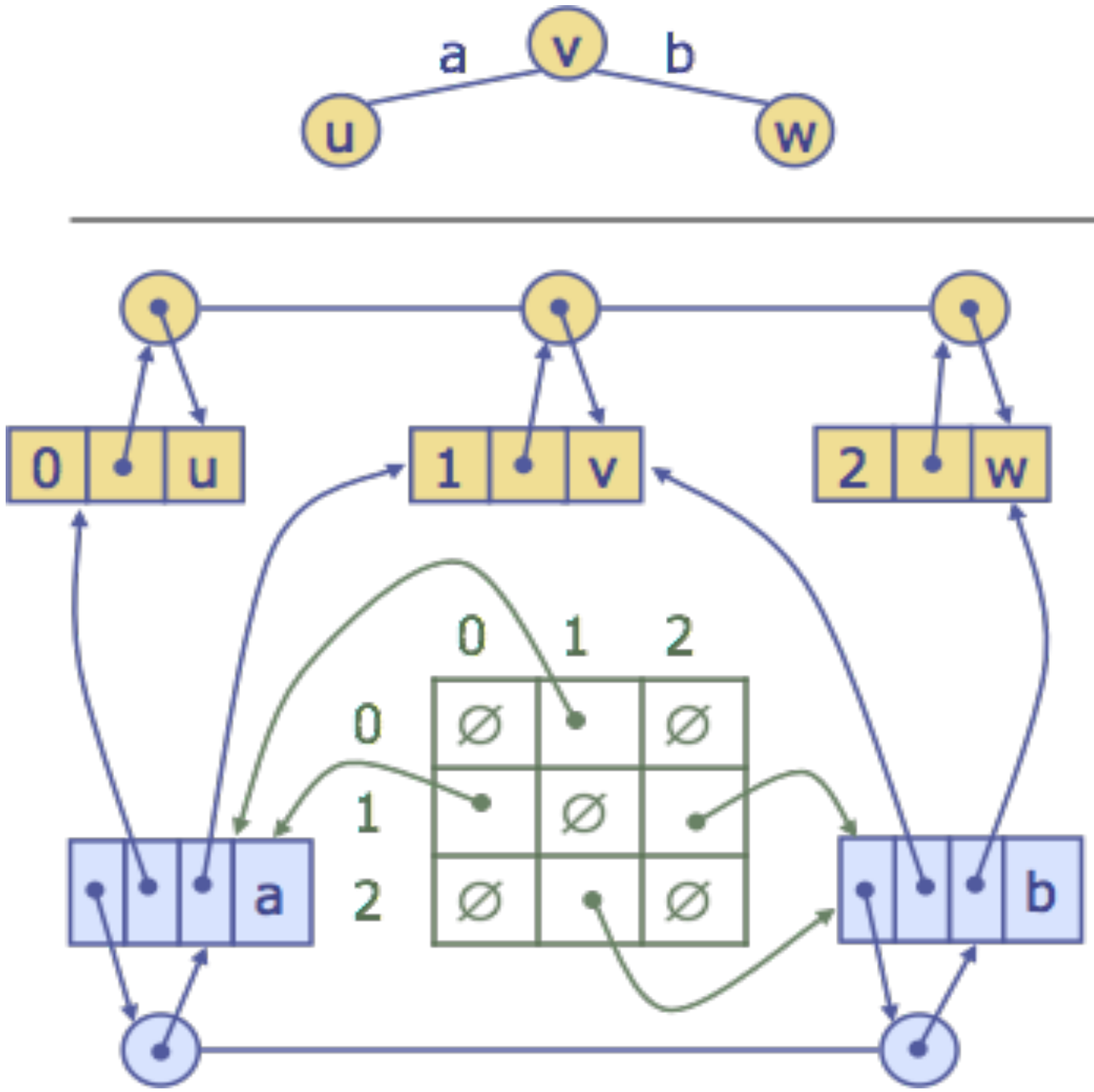
Adjacency Matrix Structure (3)



Finally, each vertex object now stores a number associated with it

It is used to refer to its position (index) in the adjacency matrix

Adjacency Matrix Structure (4)



incidentEdges(v) – O(?)

areAdjacent(v0, v1) – O(?)

insertVertex(v) – O(?)

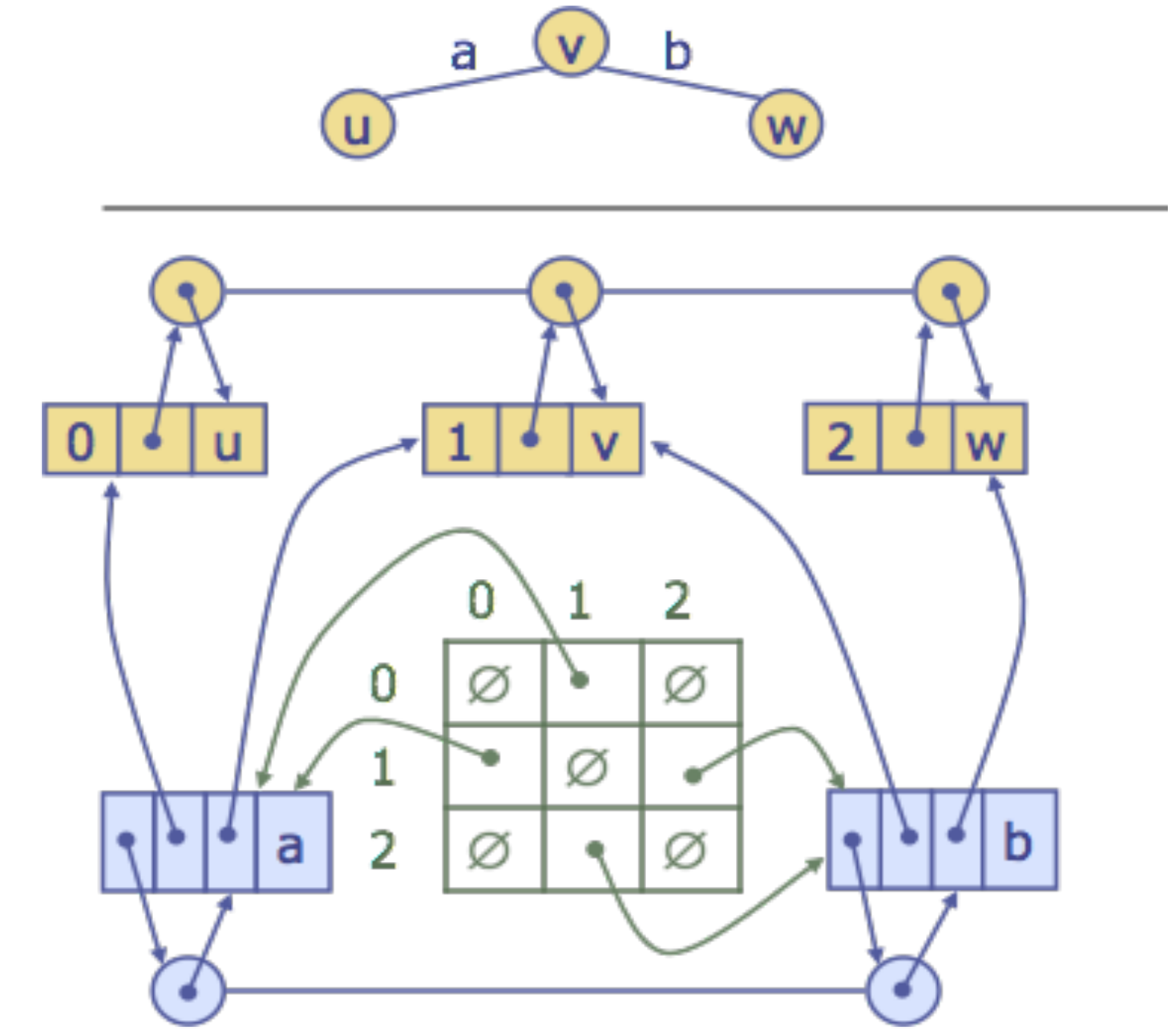
insertEdge(e, o, d) – O(?)

removeVertex(v) – O(?)

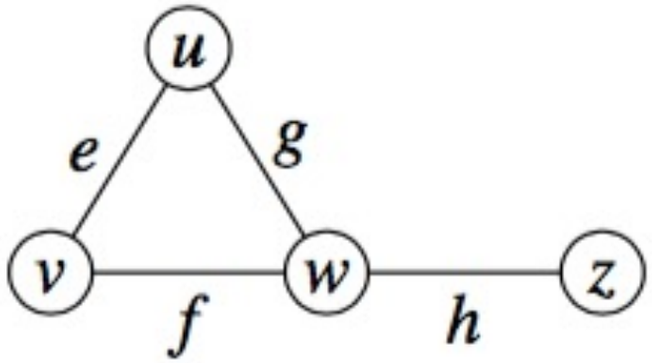
removeEdge(e) – O(?)

Adjacency Matrix Structure (5)

- **Complete structure, Example from the book (old)**



Adjacency Matrix Structure (5)



- **Incomplete structure, Example from the book (new)**

		0	1	2	3
u	0		e	g	
v	1	e		f	
w	2	g	f		h
z	3			h	

Graph Representations

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

Remember: m = n(n-1)/2 in worst case

Did we achieve today's Objectives?

- Graphs
- Graph ADT (What is it and why we need it)
- Graph Representations (ways to represent Graphs)

Data Structures & Algorithms in Java, Ch:14