# Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Kazan, Russia

# Graphs
# Graph Representations

# 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?

- For general kinds of data storage problems

  Don’t need graphs

- But for some problems, **graphs are indispensable**

# 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

- For example

  **Road networks**

  **Internet**

  **Molecules in chemistry**

  **Social networks**

  **Individual tasks necessary to complete a project**

- In all these cases, the shape of a graph arises from the specific real-world situation

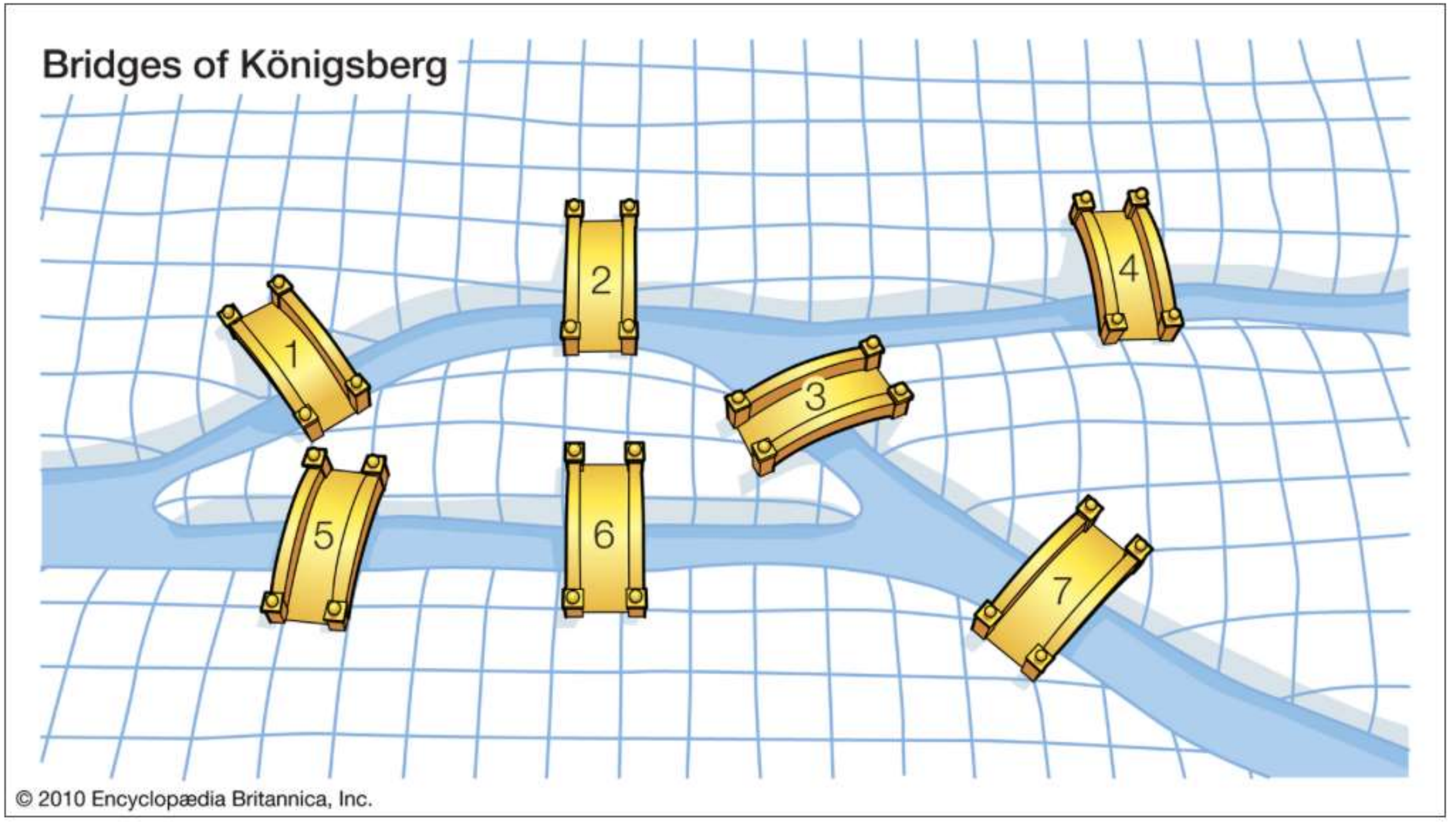
# Graphs

- The key to resolving problems related to such 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 existing graph algorithms

# 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

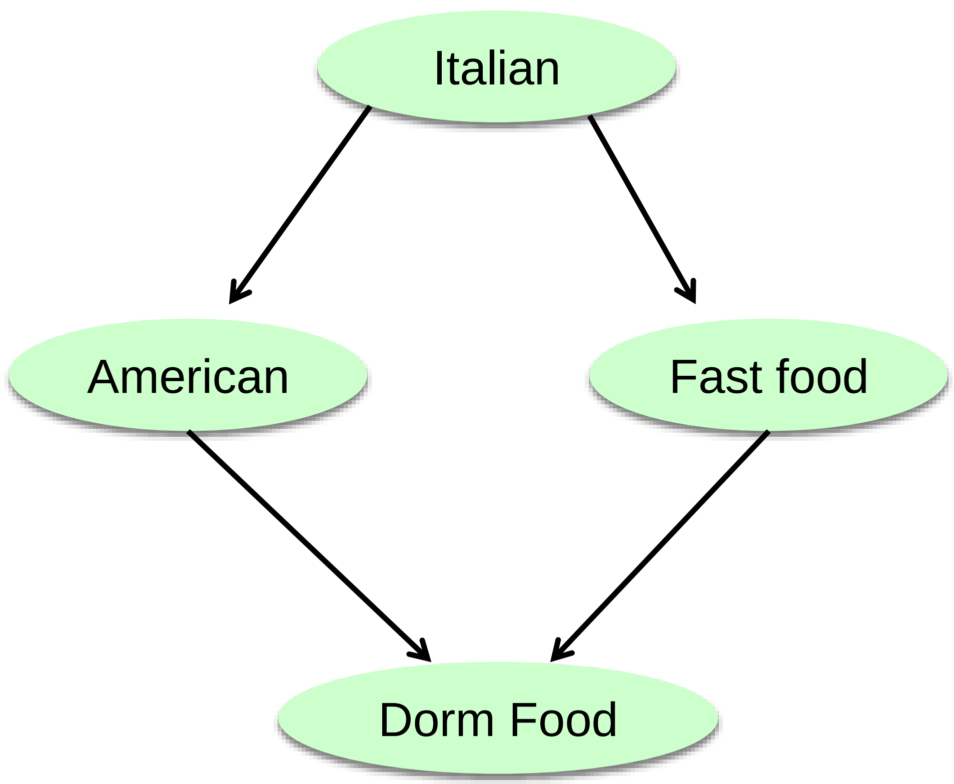
- Useful if we want to pair objects such that none of them is “first” or “second”

# Ordered Pair

- Collection of two objects a and b in order

- (a, b)

- For example, a graph where each node represents a food




Two ordered pairs (a0, b0) and (a1, b1) are equal if a0=a1 and b0=b1

# Graphs

- A graph $G = (V, E)$ where

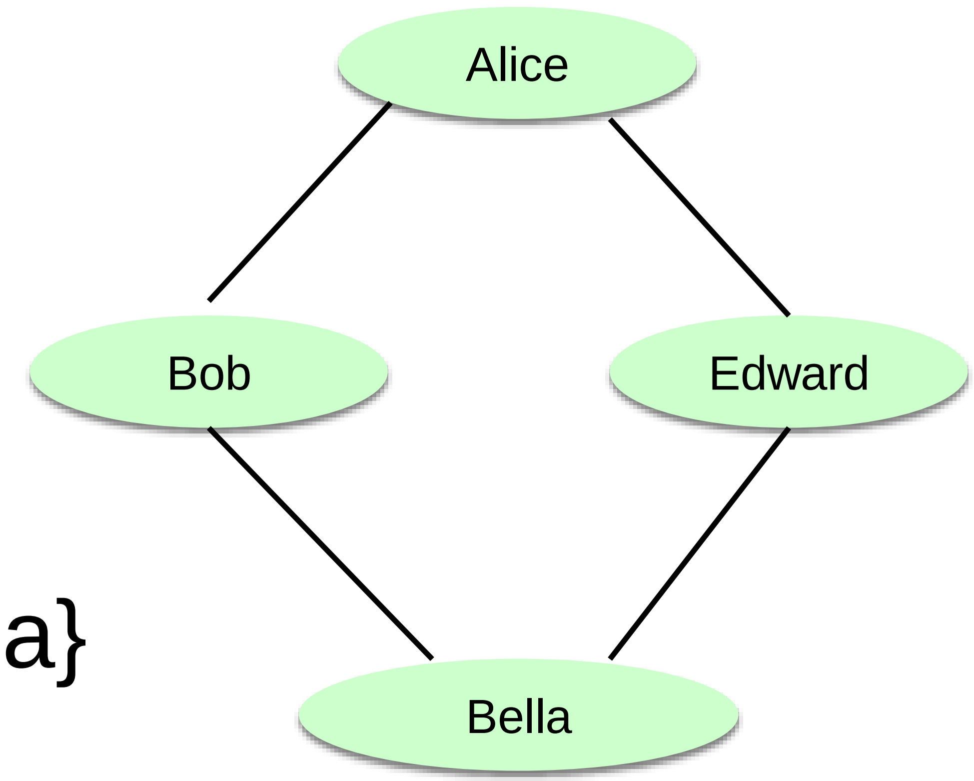
  $V$ is a set of vertices, and

  $E$ 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

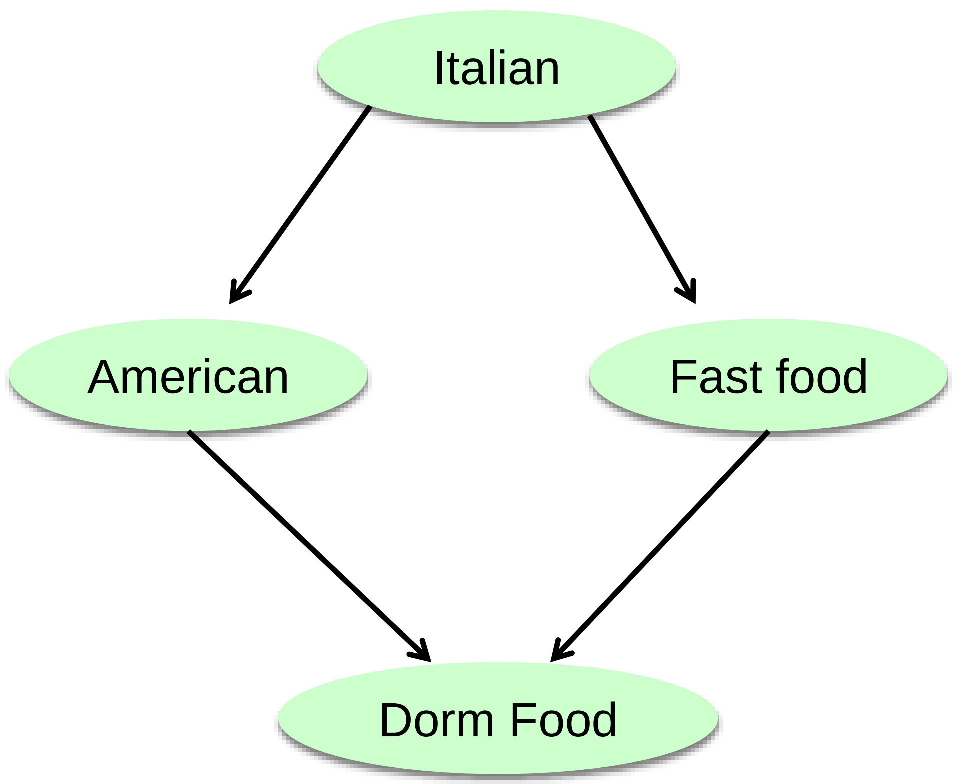
- A graph G = (V,E), where
- E is a set of unordered pairs




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

# Directed Graphs

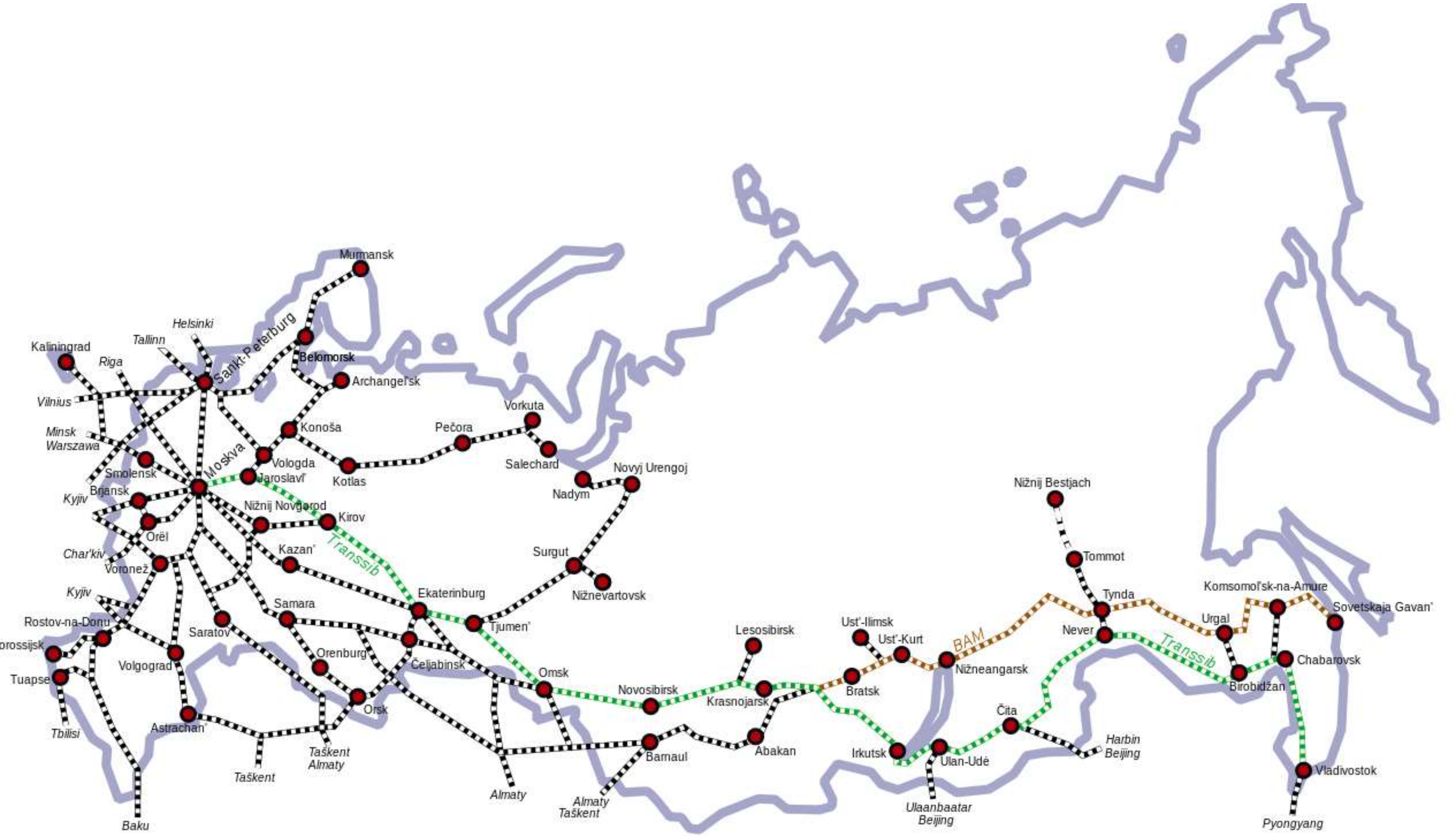
- A graph G = (V,E), where
- E is a set of ordered pairs




- V = { Italian, American,

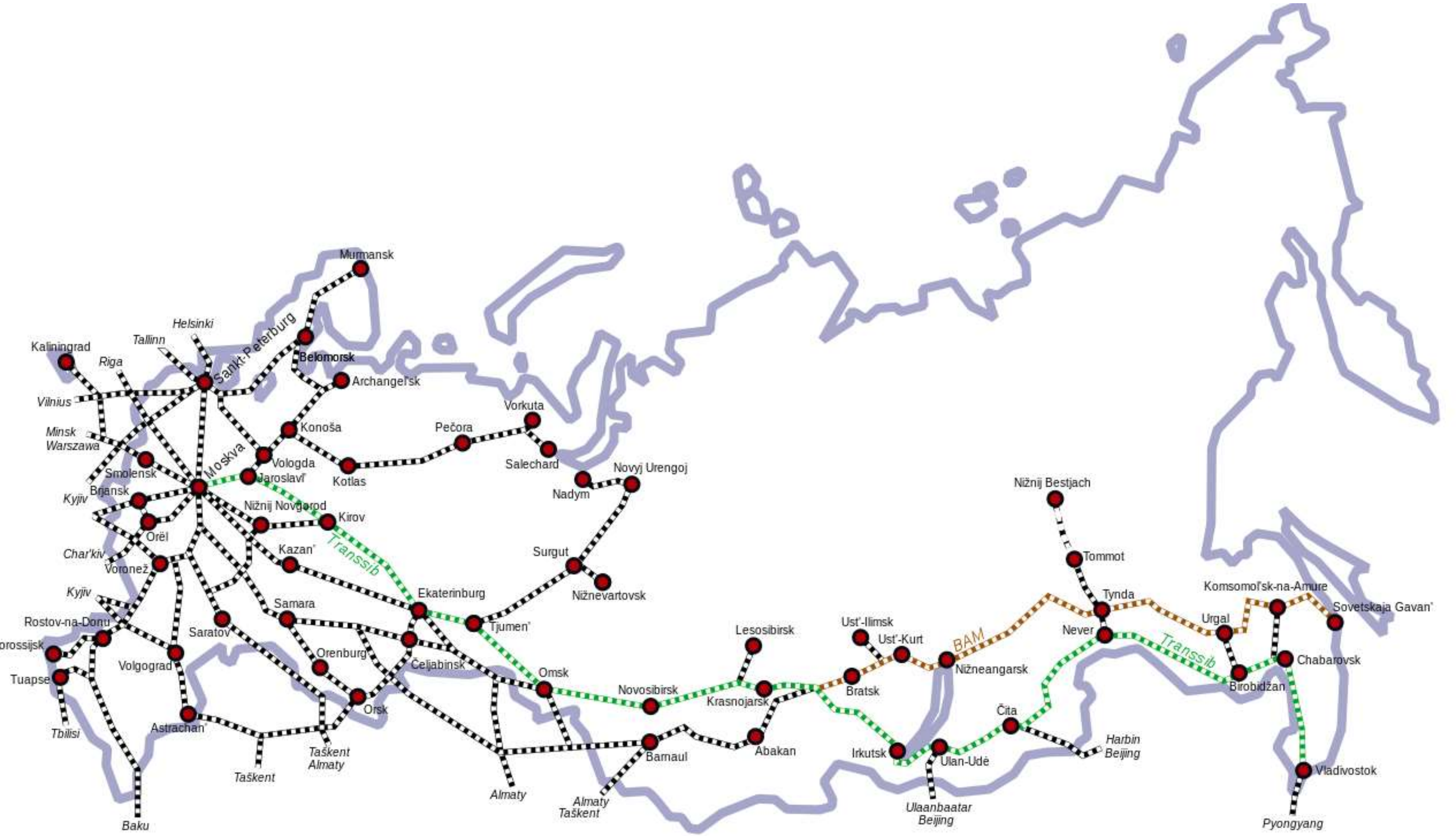
  Fast food, Dormfood}

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

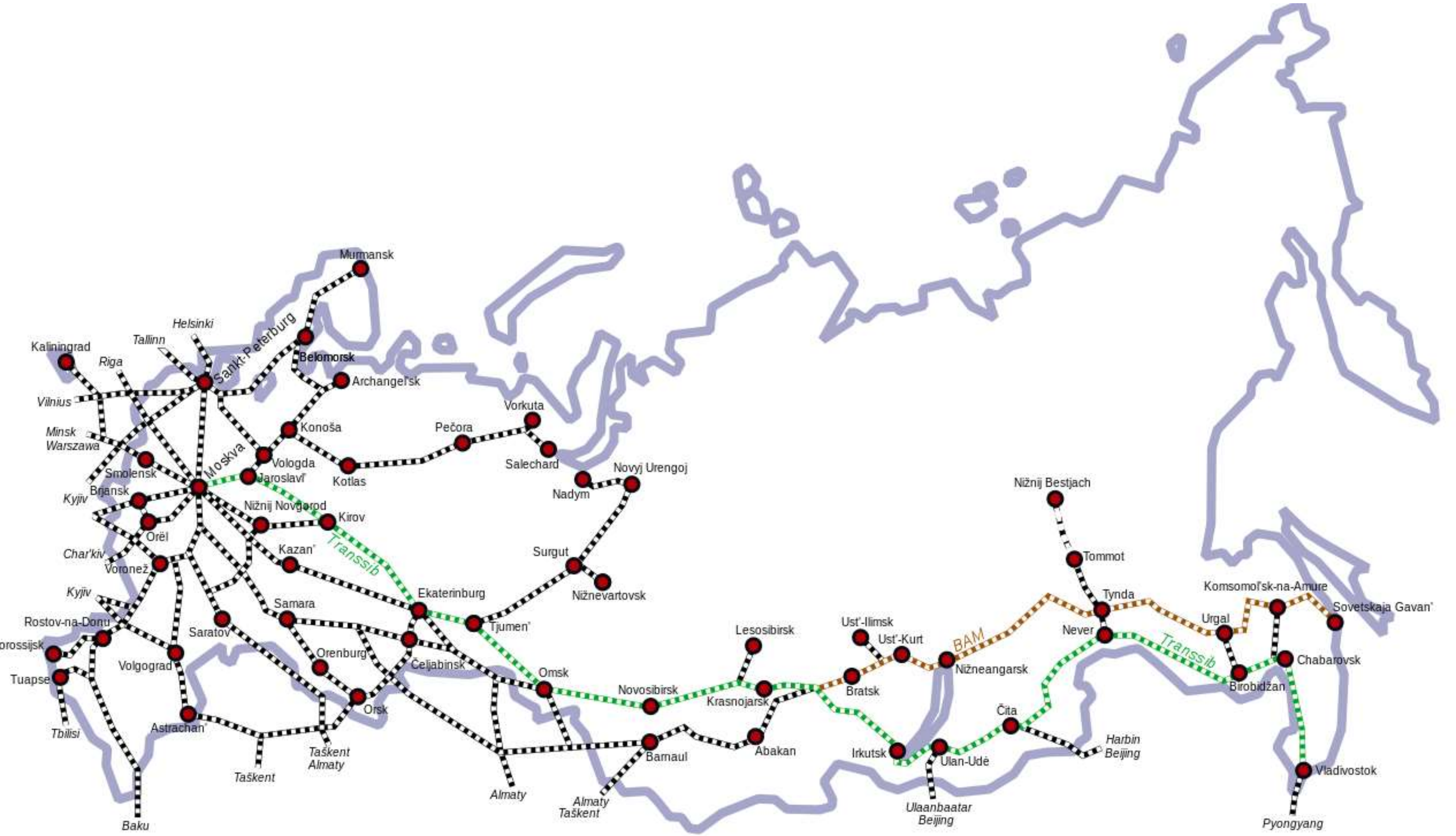



- **Adjacent vertices:** two vertices in a graph that are connected by an edge – Kazan and Moscow

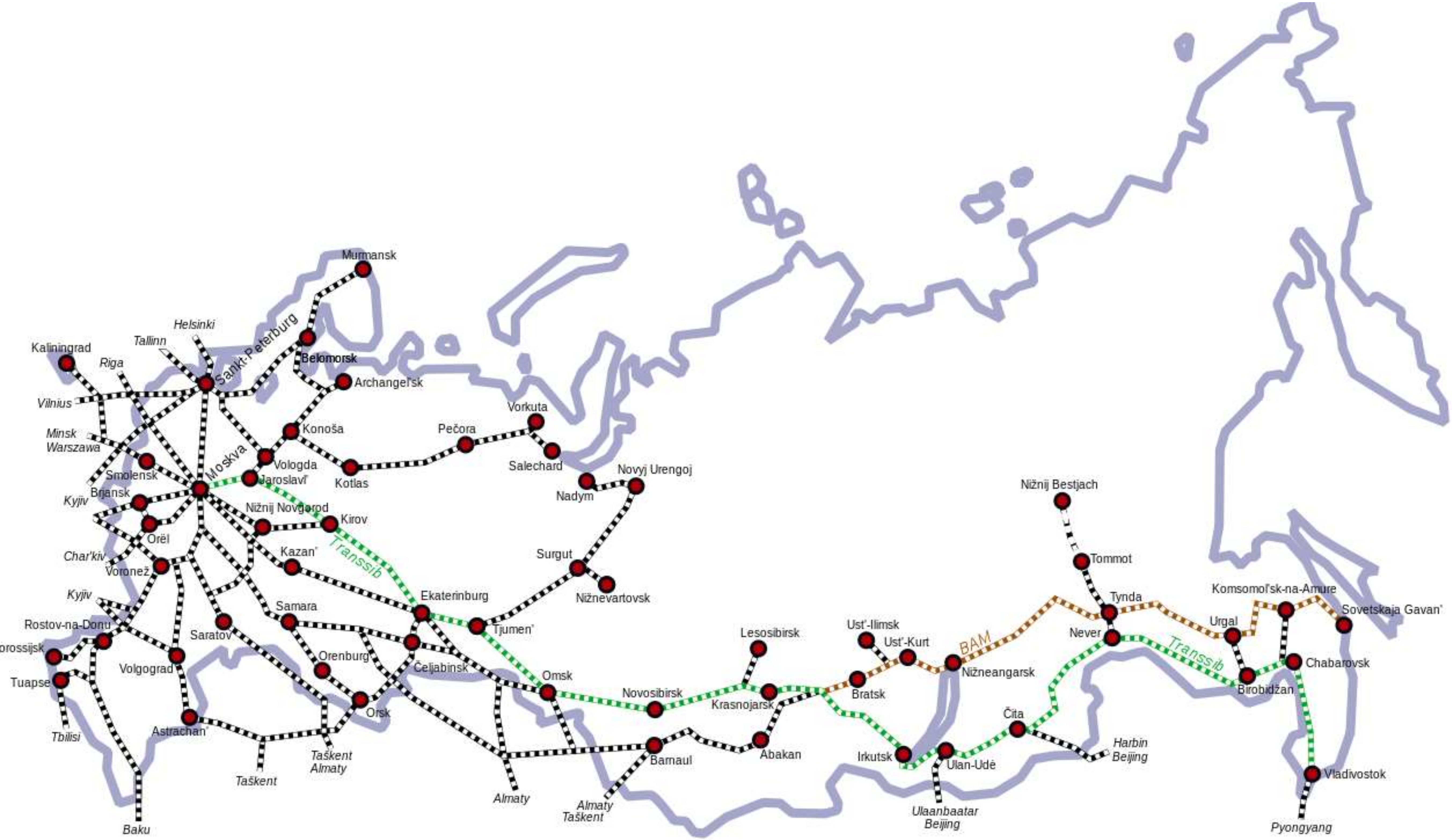

- **Path:** a sequence of vertices that connects two vertices – (Kazan, Moscow, Saint Petersburg)



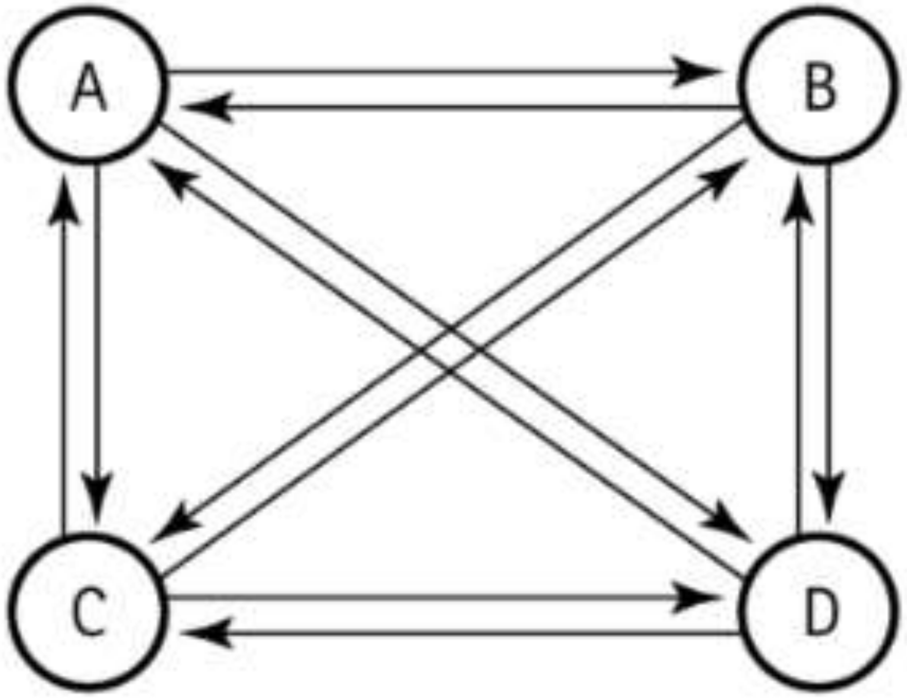

- **Simple Path:** A path with no repeated vertices
  *For example,*
  (Kazan, Moscow, Kazan, Ekaterinburg) is not a simple path



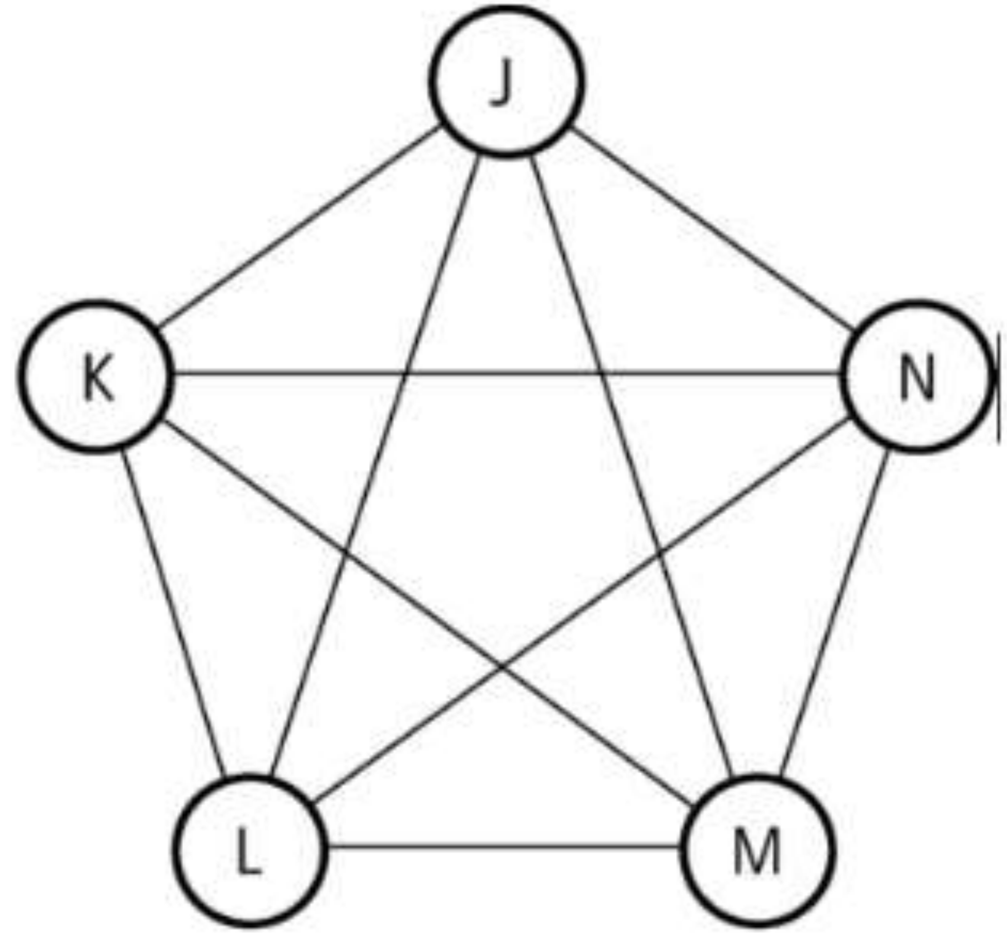

- **Cycle:** A path that starts and ends at the same vertex– (Kazan, Ekaterinburg, Kirov, Yaroslavl, Moscow, Kazan)




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

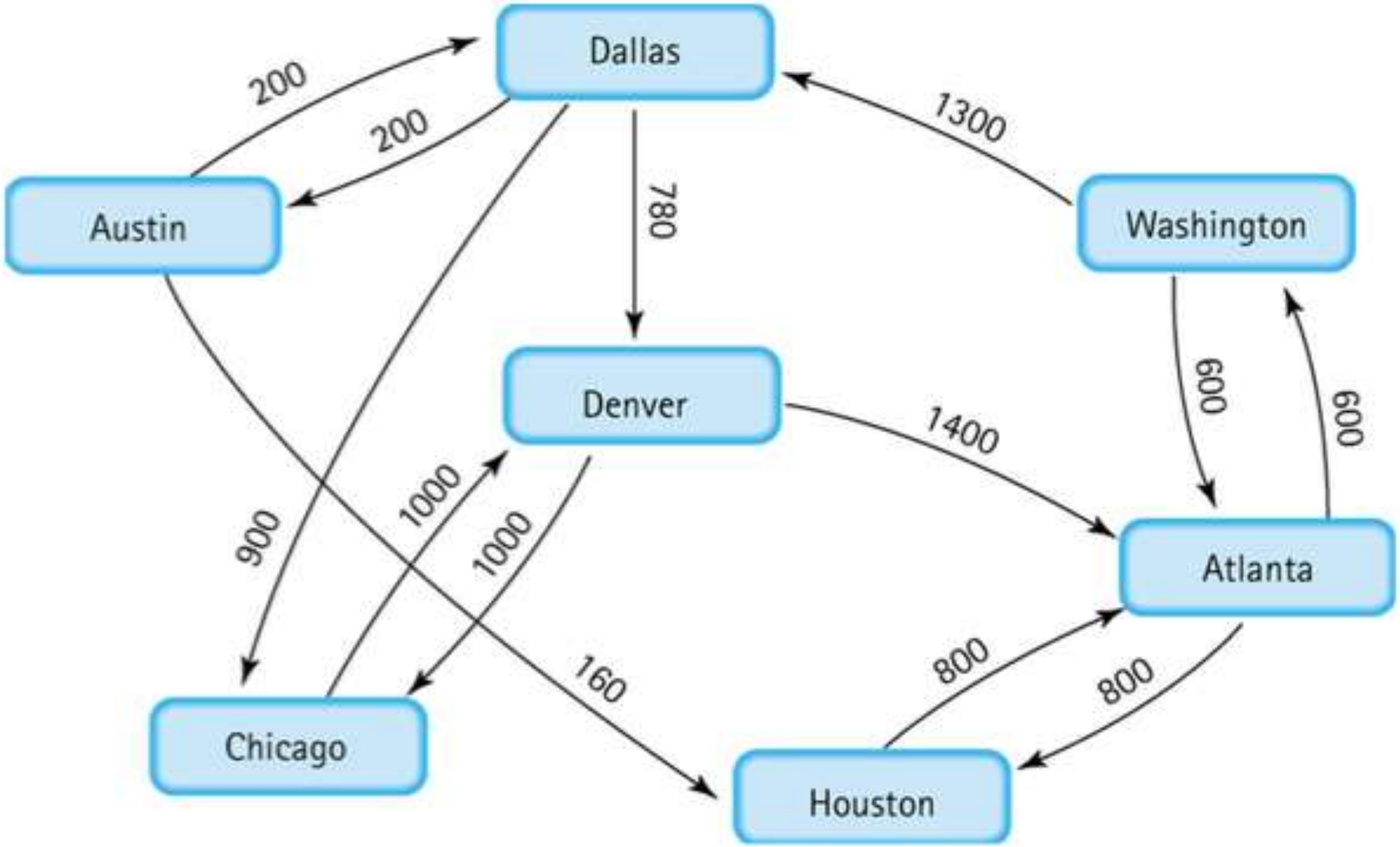



A complete directed graph $G$




A complete undirected graph $G$

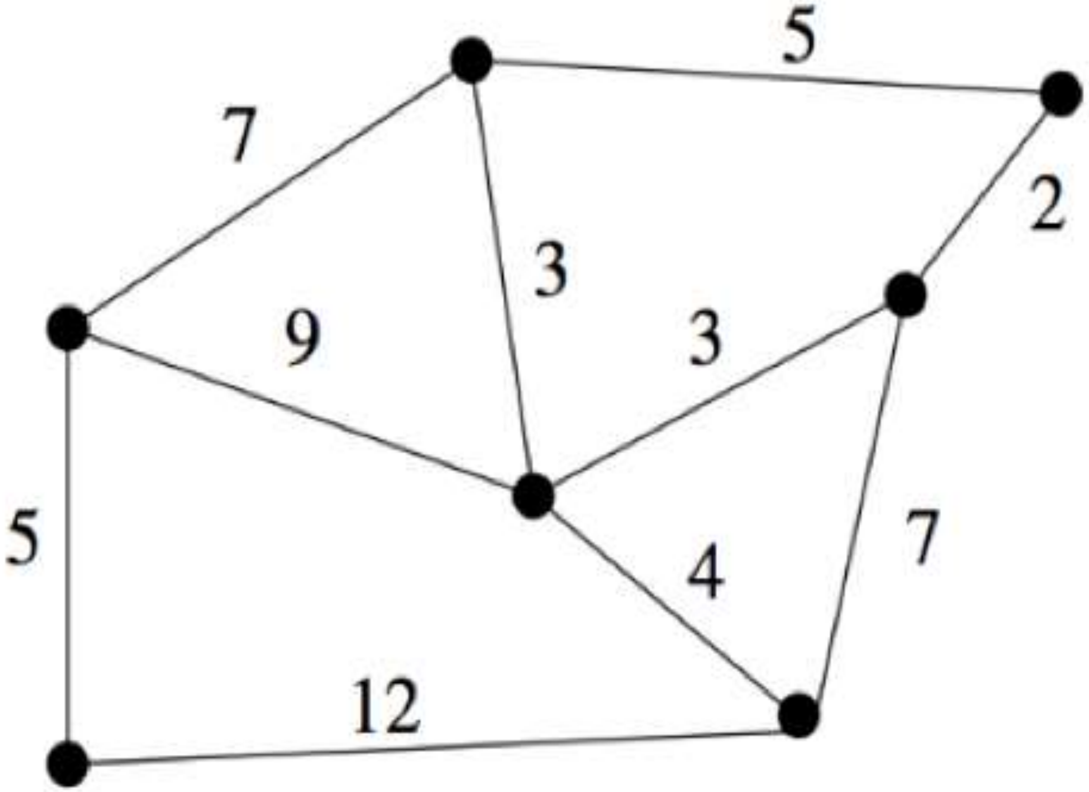
- **Complete:** A graph in which every vertex is directly connected to every other vertex




A weighted graph $G$

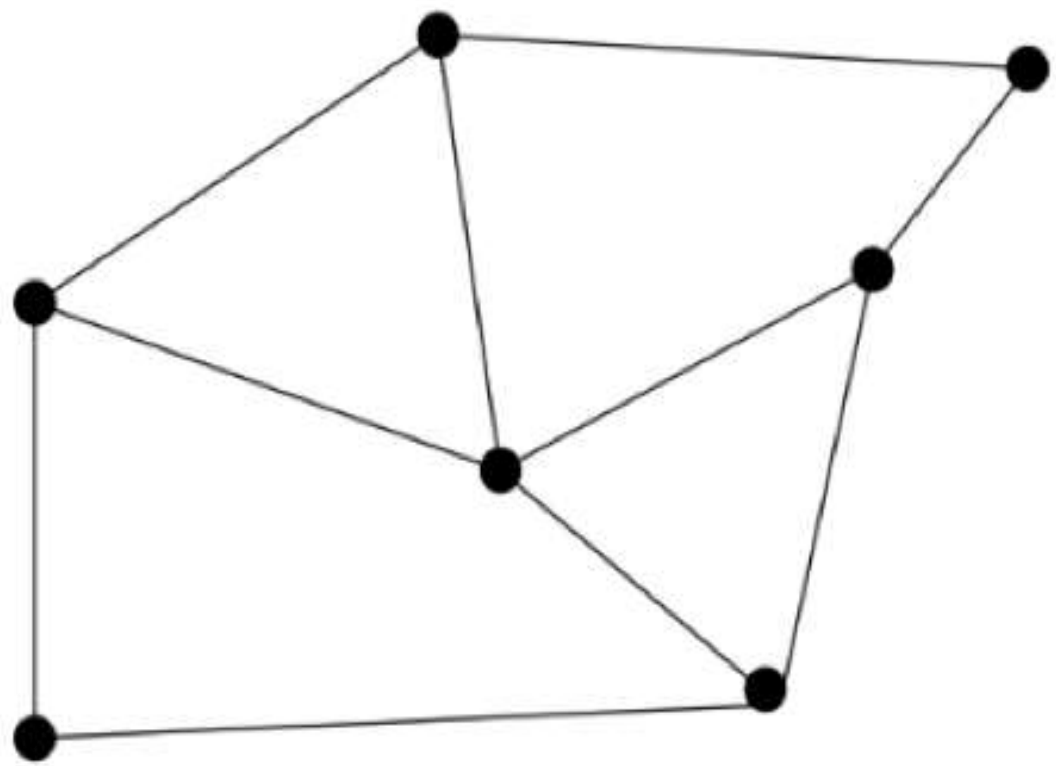
- **Weighted graph:** a graph in which each edge carries a value (weight)

# Shortest Path



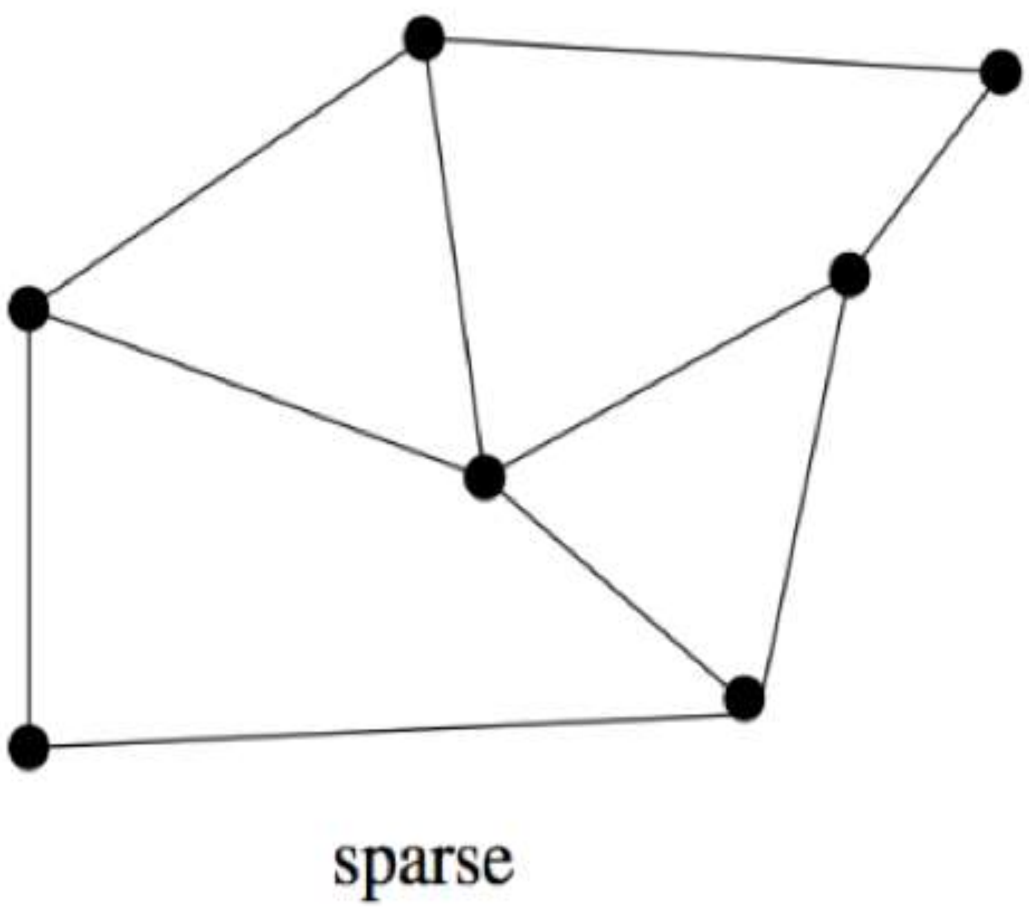

unweighted

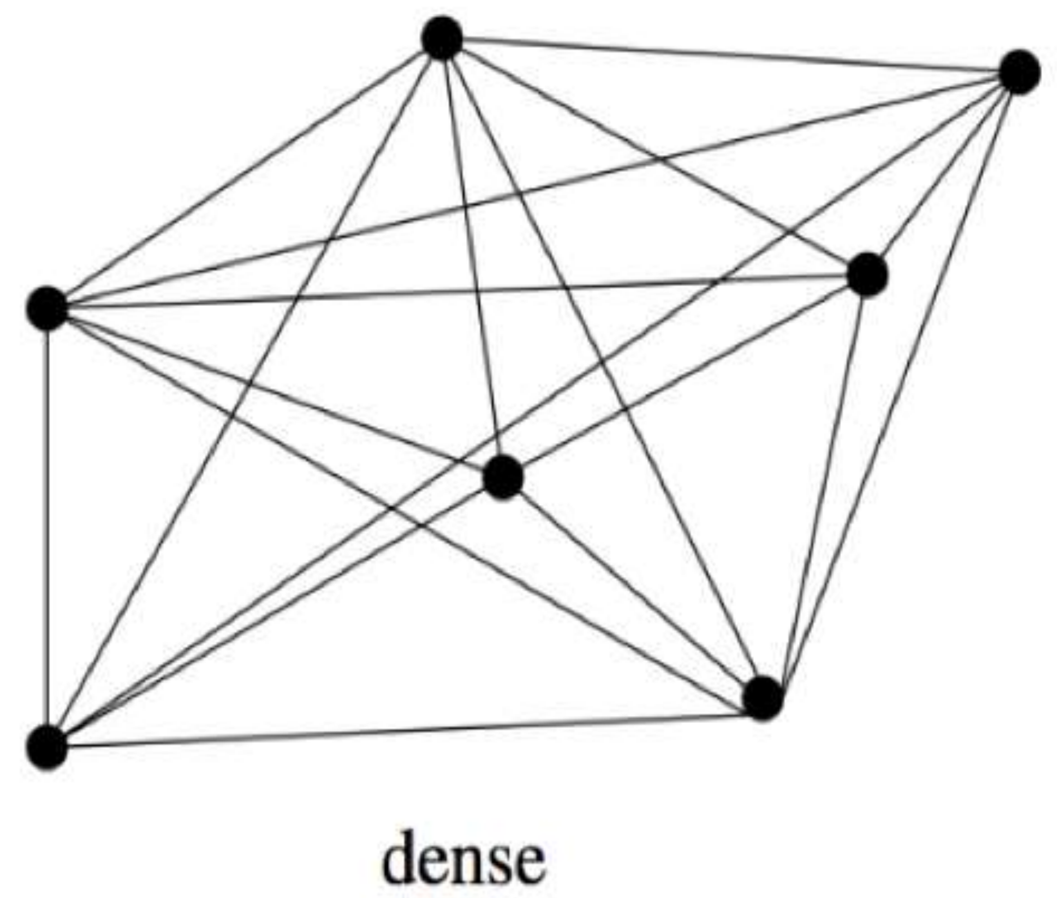
weighted



- 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




- 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 will learn more about graphs, their properties, theorems and proofs associated with those properties in “Discrete Math” course, next semester
- For now, let’s focus on how to implement them as an abstract data type

# Main Methods of the Graph ADT

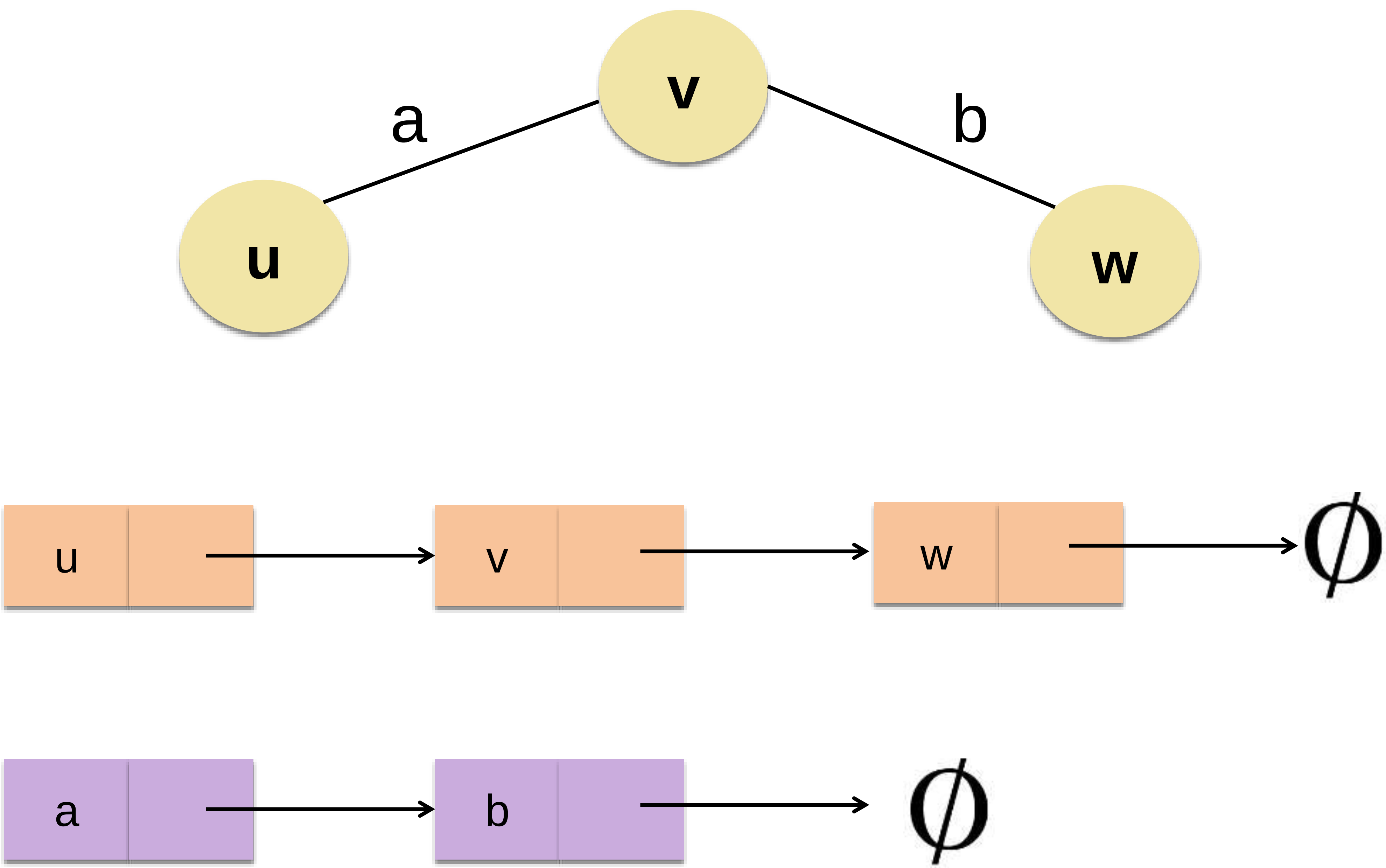
- endVertices(e): an array of the two endvertices of e
- opposite(v, e): the vertex opposite of v on e
- areAdjacent(v, w): true iff v and w are adjacent
- degree(v): # of incident edges
- insertVertex(o): insert a vertex storing element o
- insertEdge(v, w, o): insert an edge (v,w) storing element o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

# Graph Representations

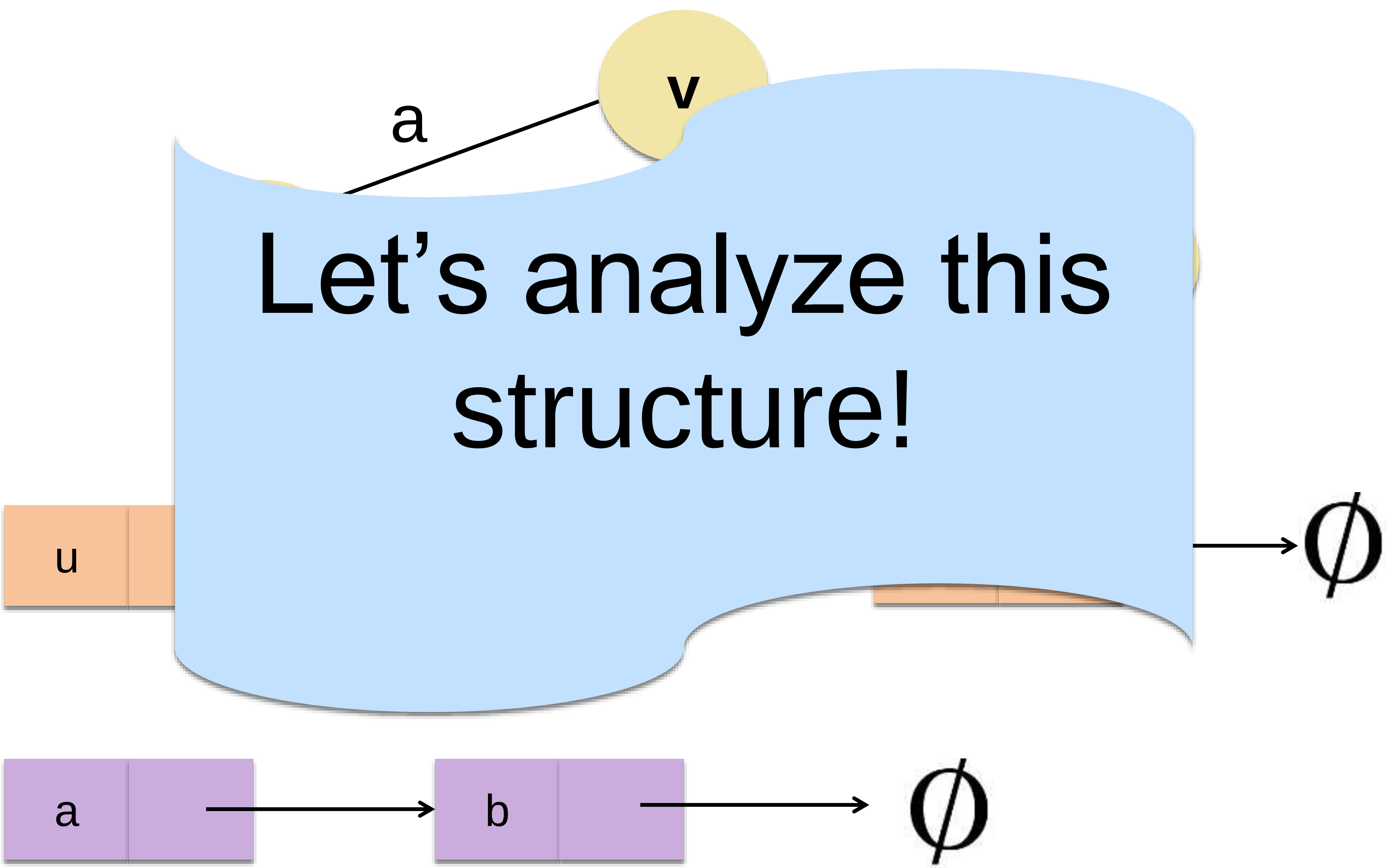
# Using Linked List (1)

- 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

# Using Linked List (2)

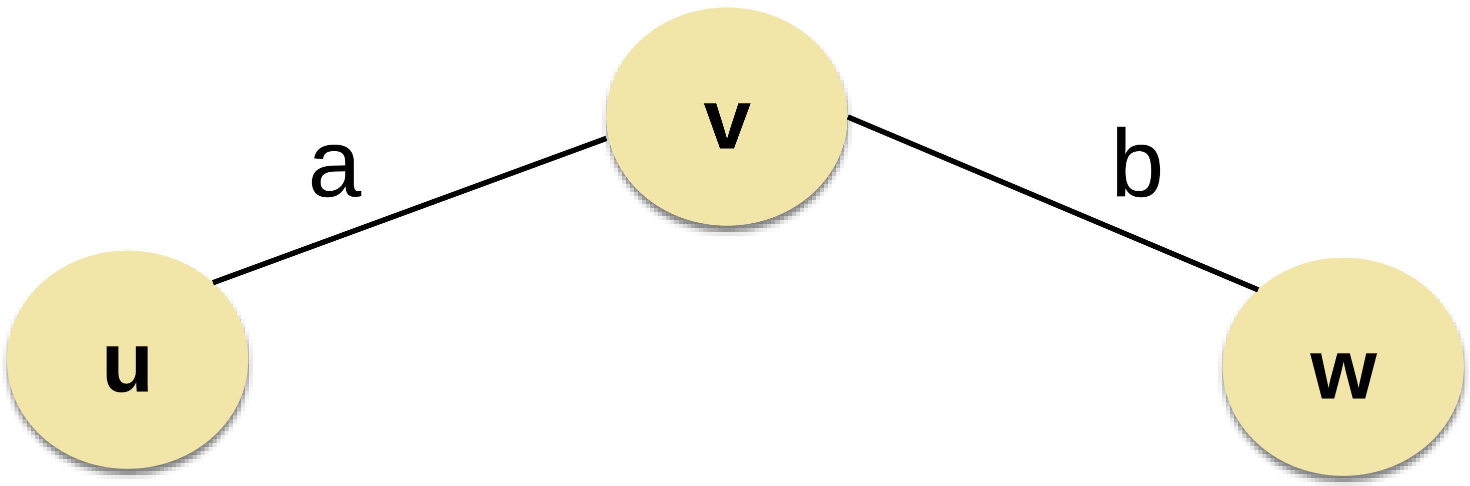


# Using Linked List (3)



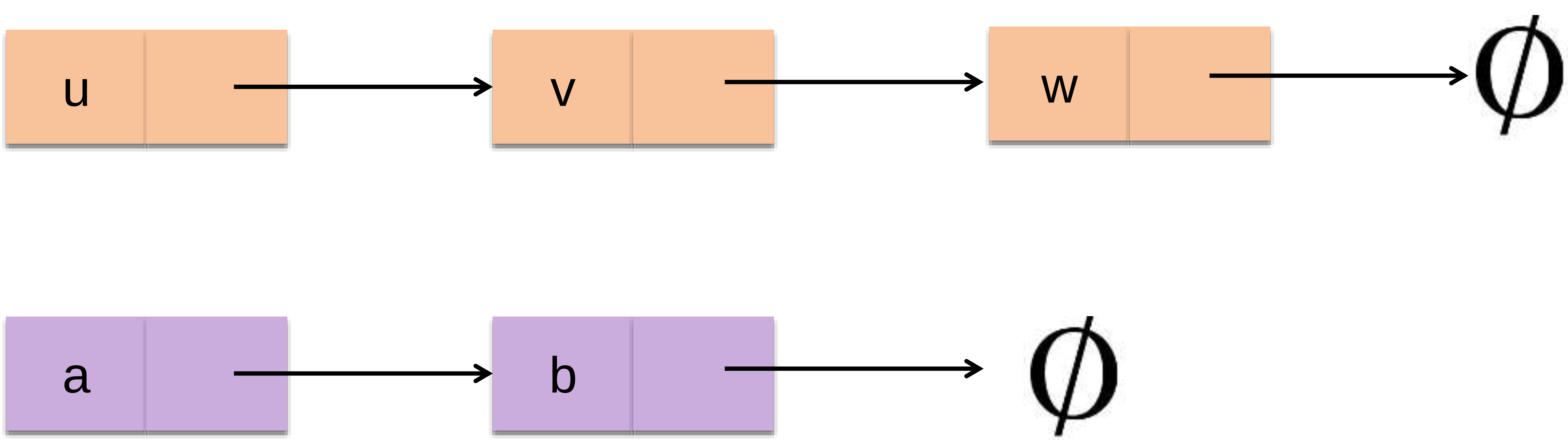

Let's analyze this structure!

# Using Linked List (4)

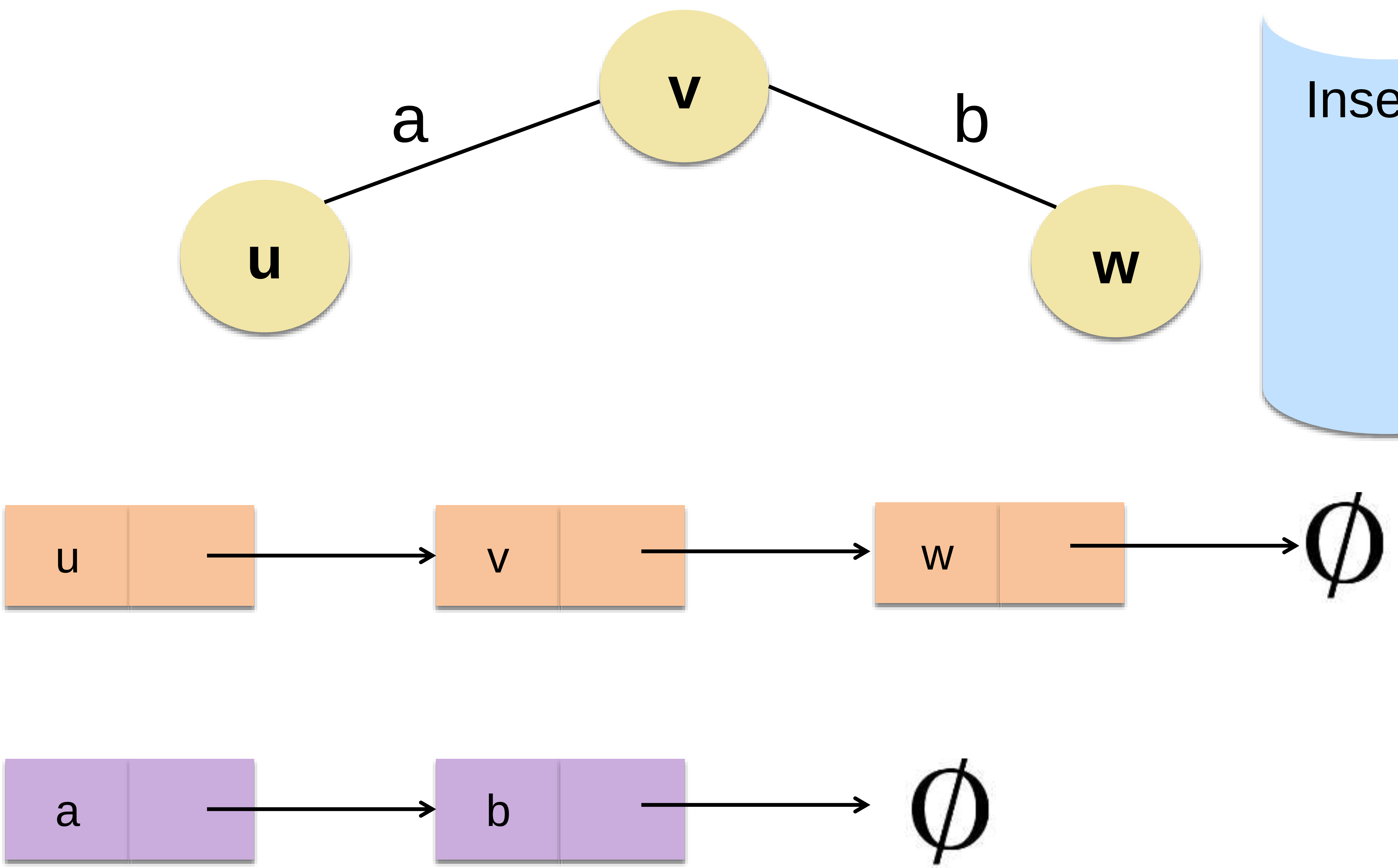



Insertion of a new vertex

O(1)



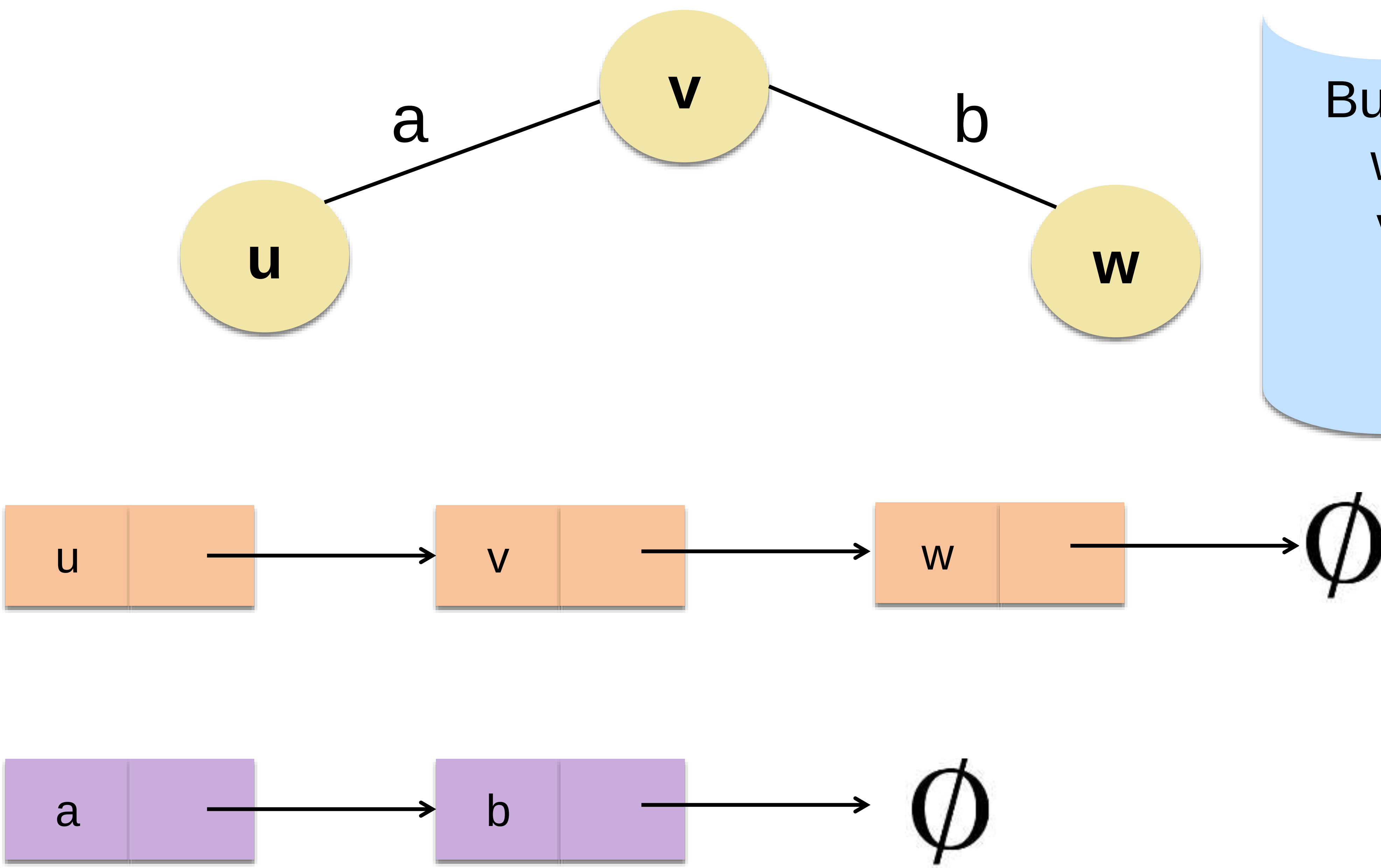
# Using Linked List (5)




Insertion of a new edge

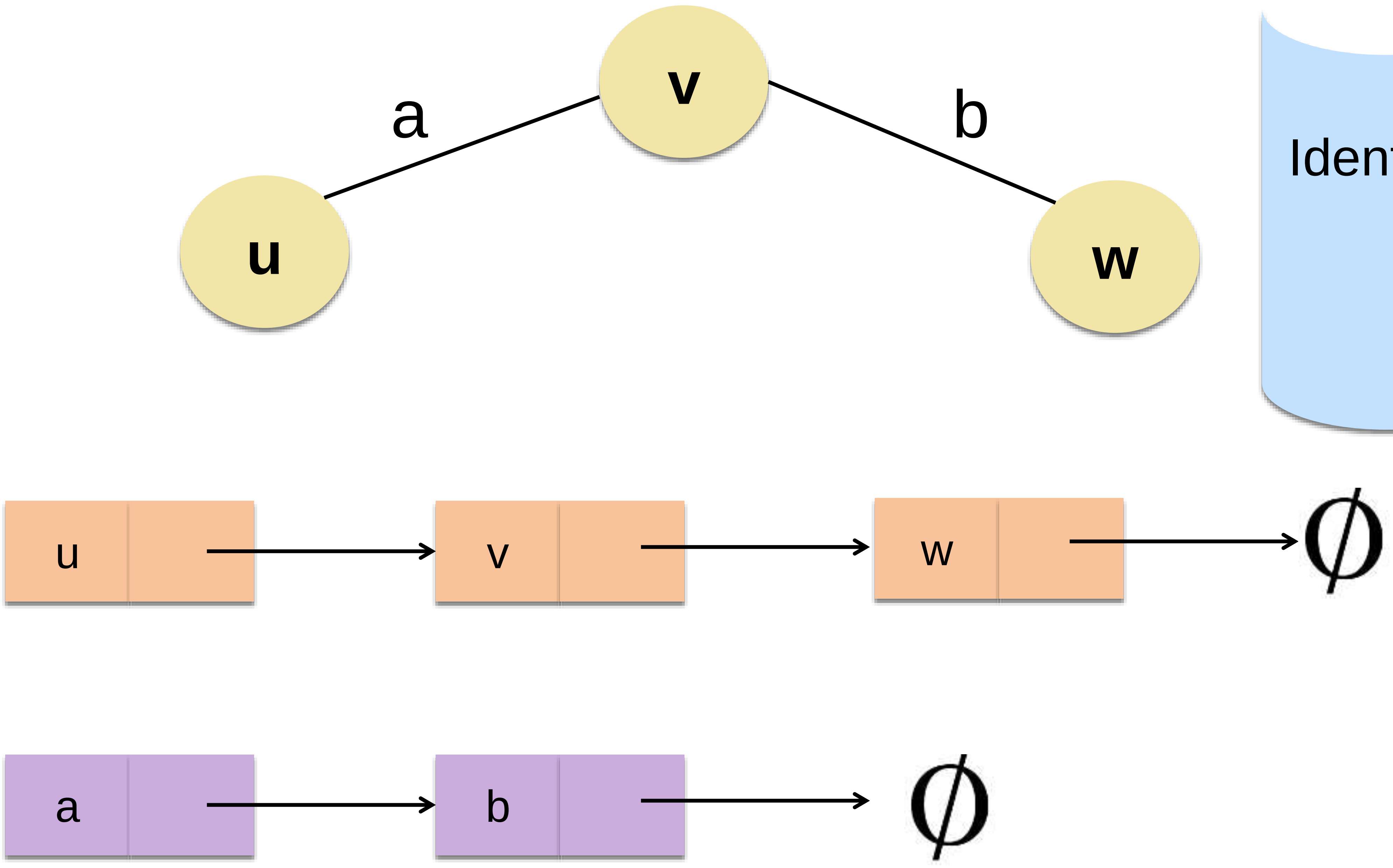
O(1)

# Using Linked List (6)



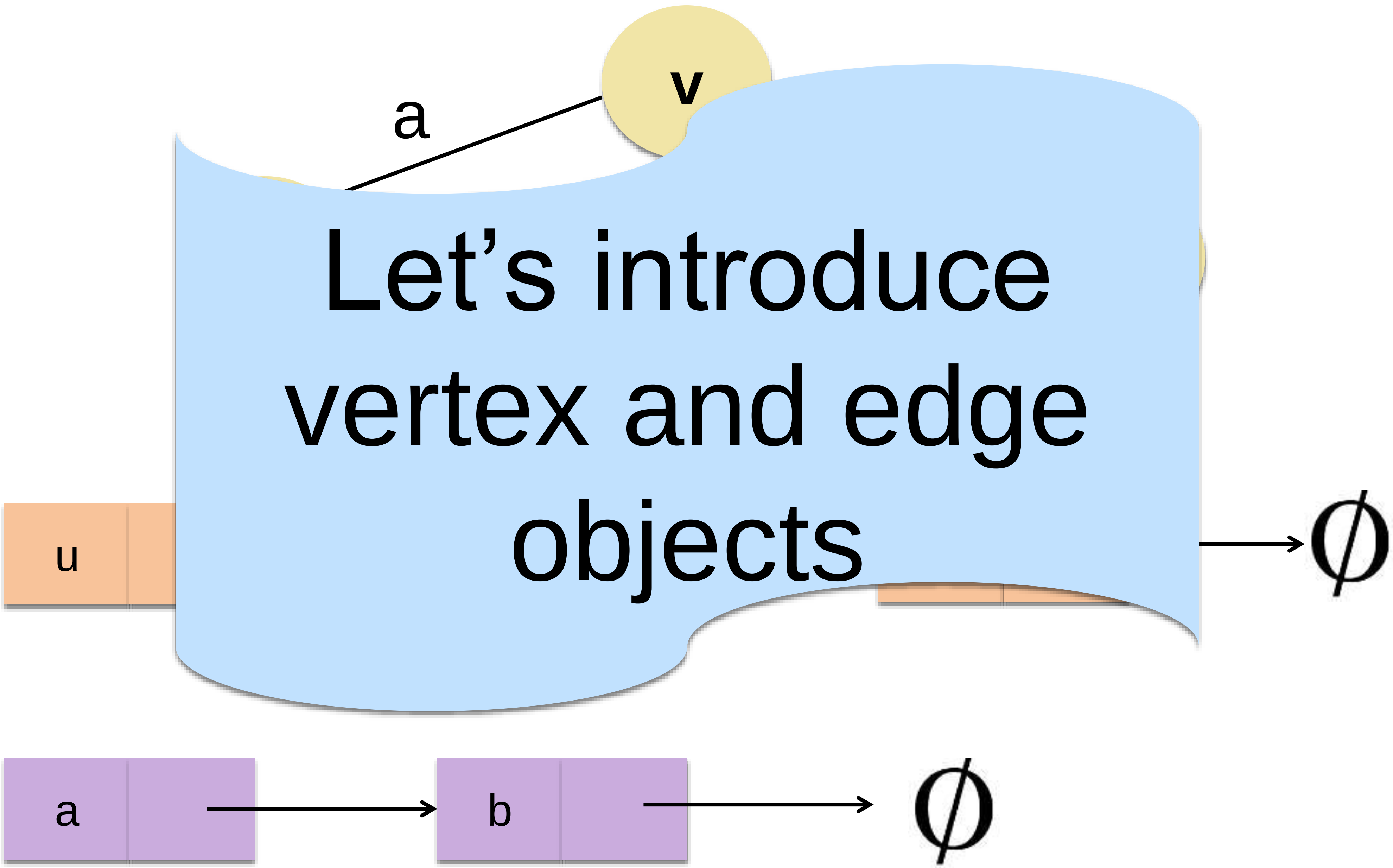

But can you find whether two vertices are adjacent?

# Using Linked List (7)



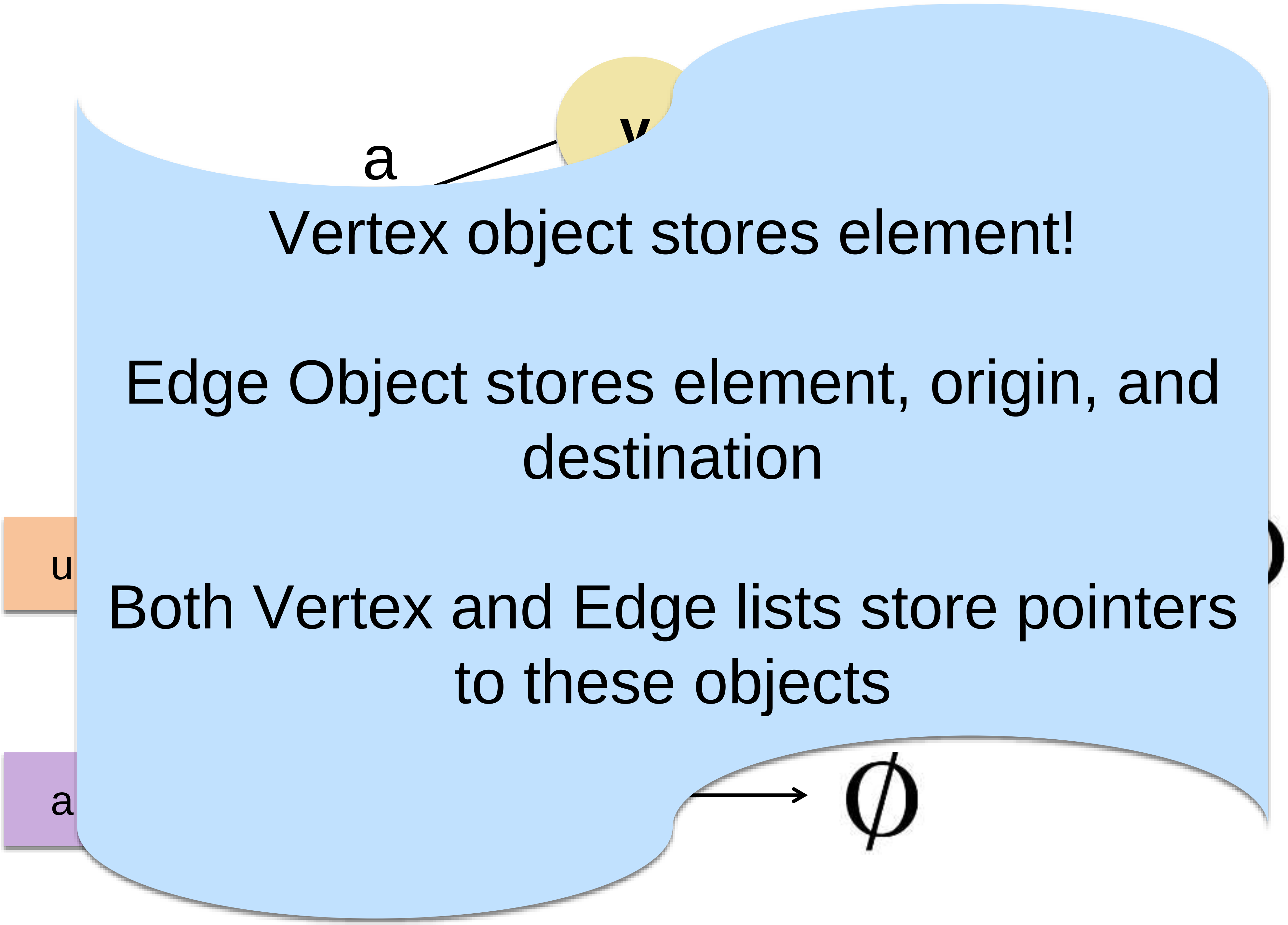

Identify some more problems!

# Using Linked List (8)




Let's introduce vertex and edge objects

# Using Linked List (9)

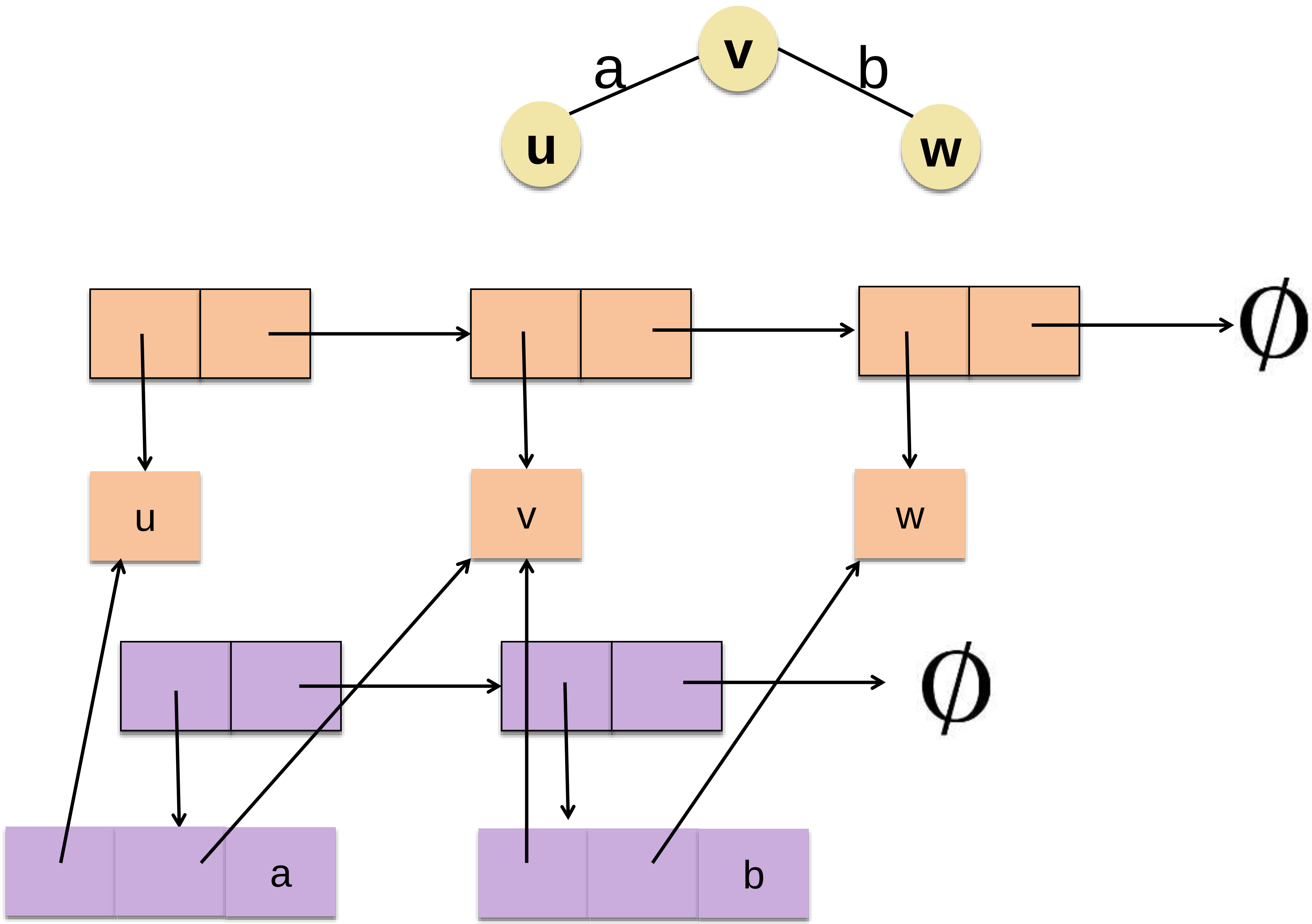



Vertex object stores element!

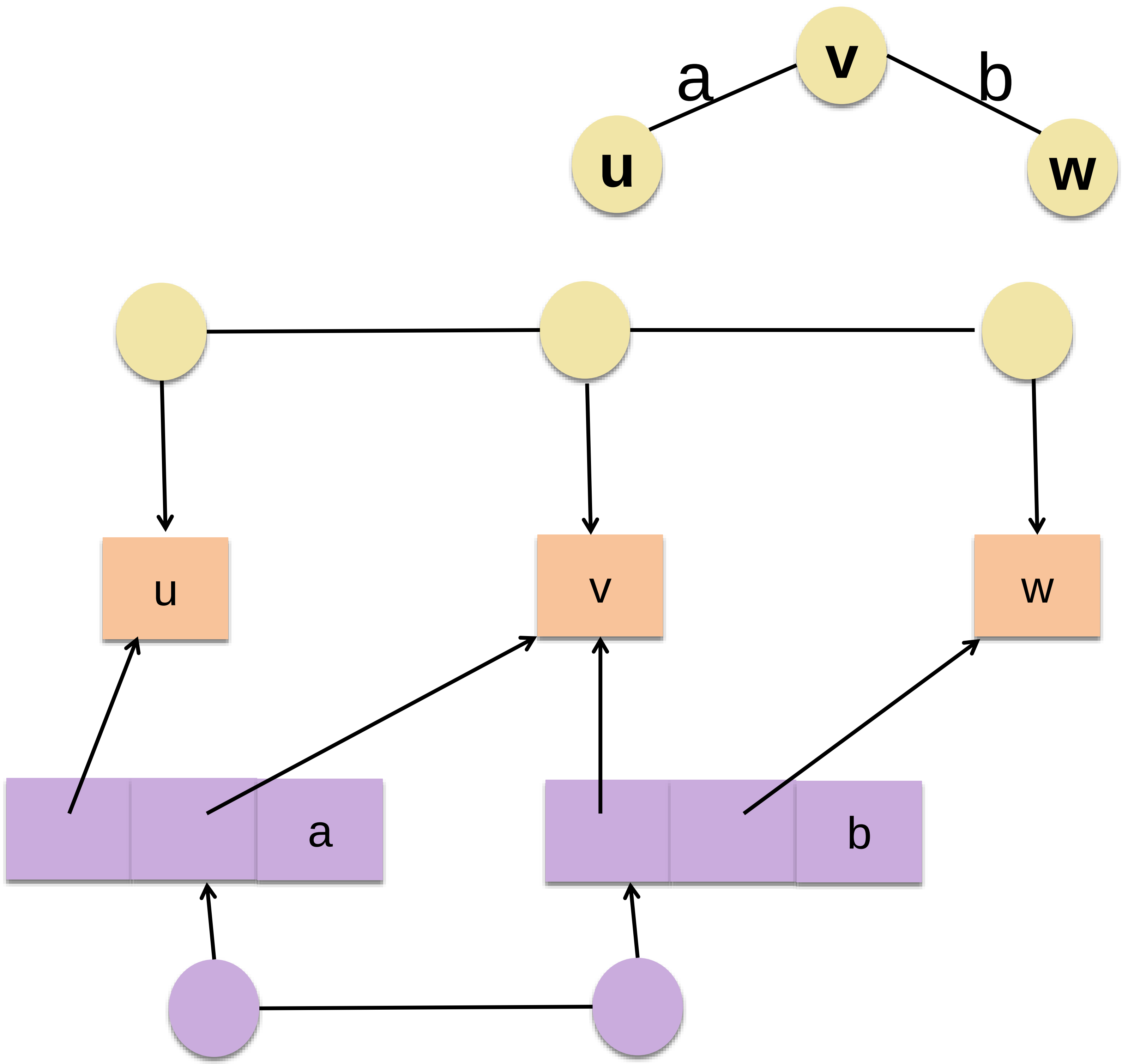
Edge Object stores element, origin, and destination

Both Vertex and Edge lists store pointers to these objects

# Using Linked List (10)

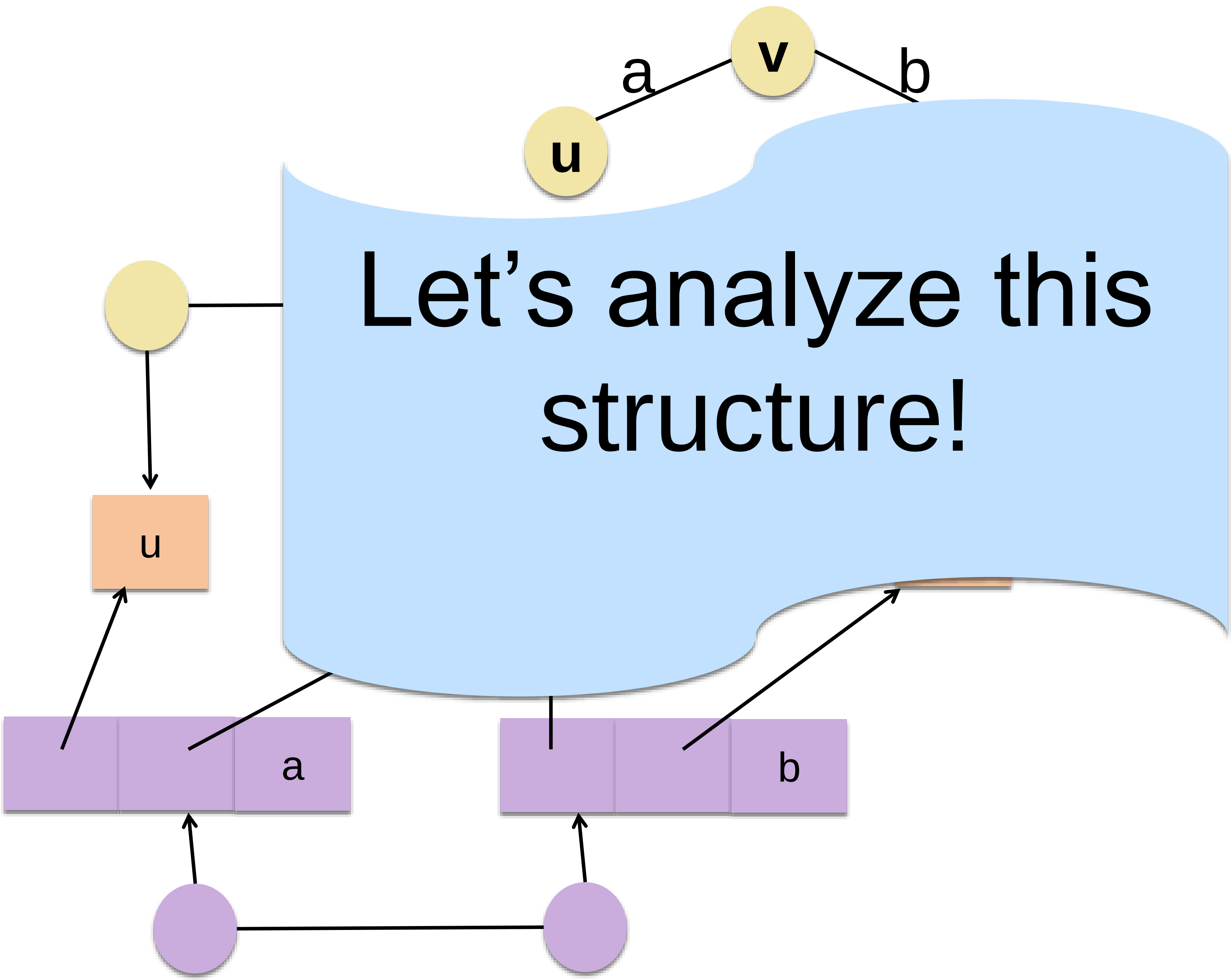


# Using Linked List (11)

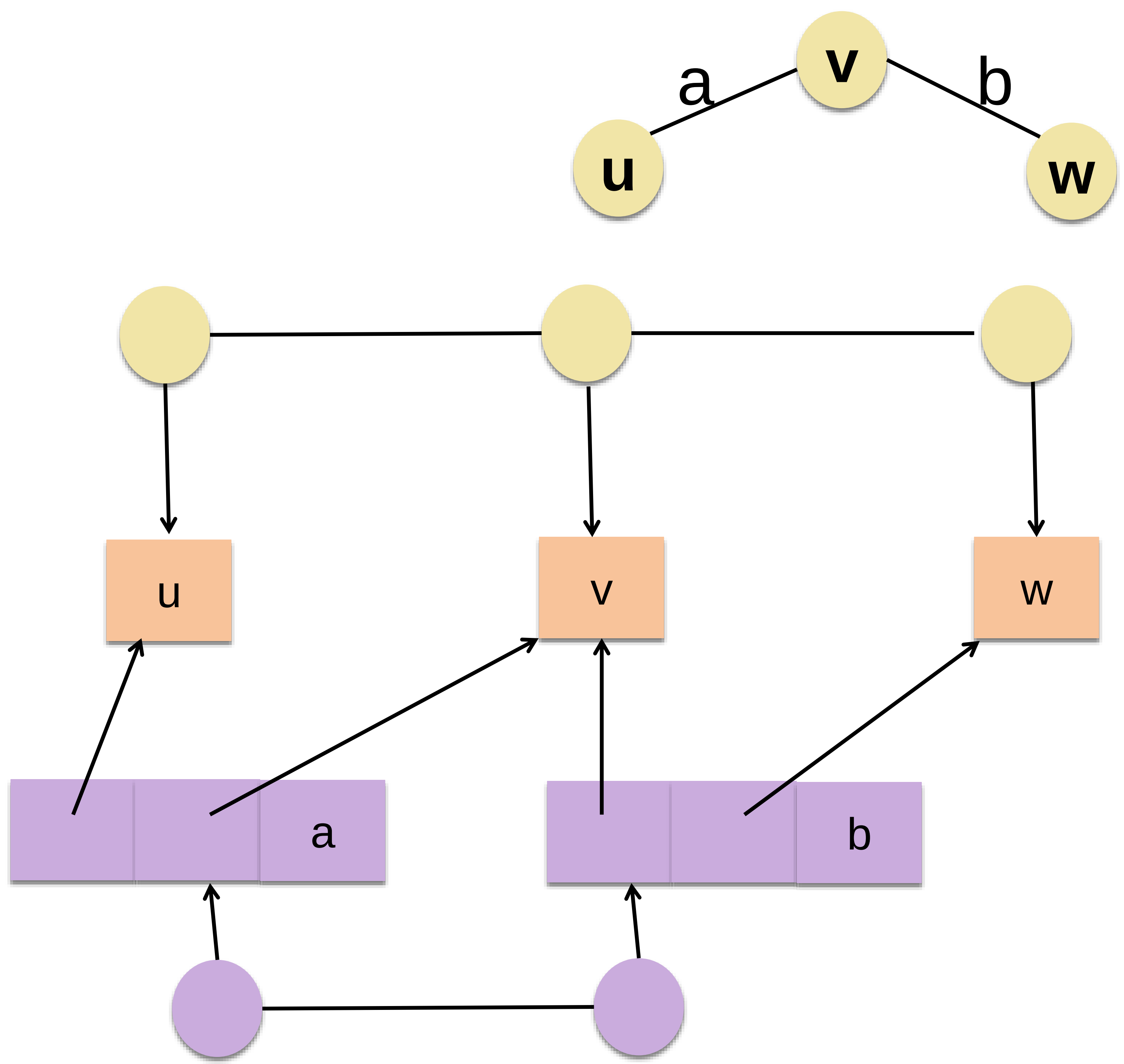



Simplified

# Using Linked List (12)

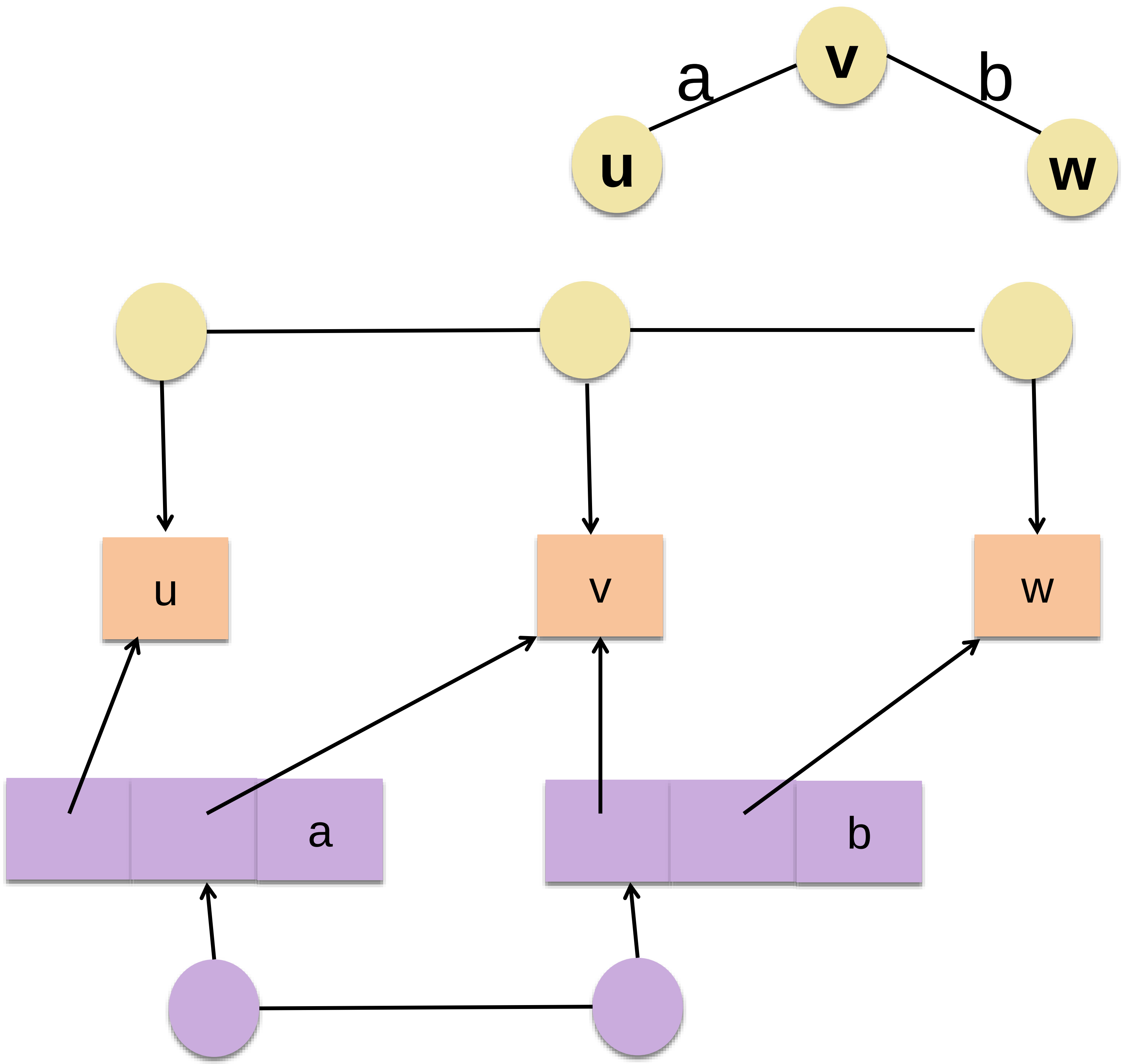


# Using Linked List (13)



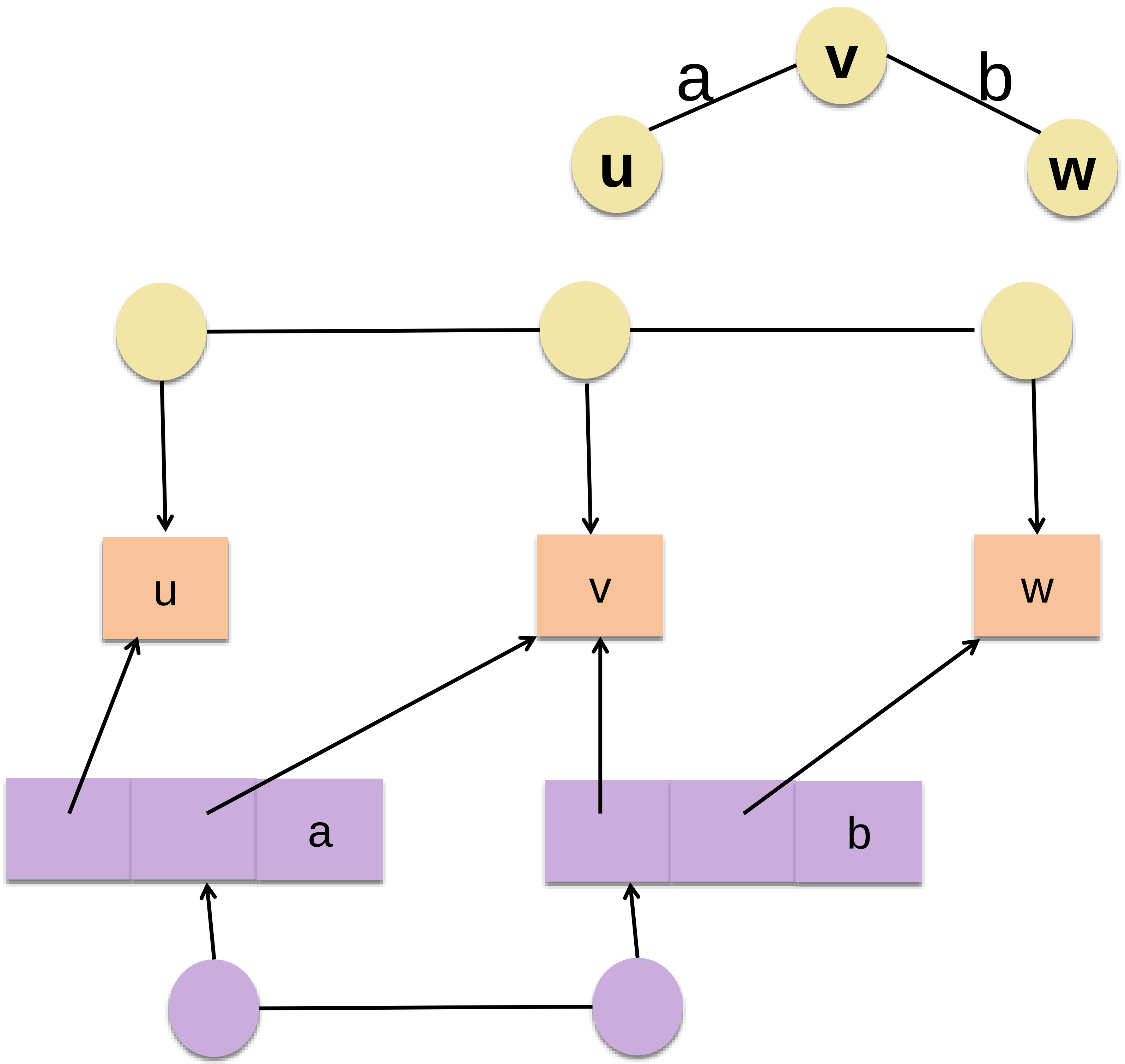

insertVertex(v)
O(1)

# Using Linked List (14)



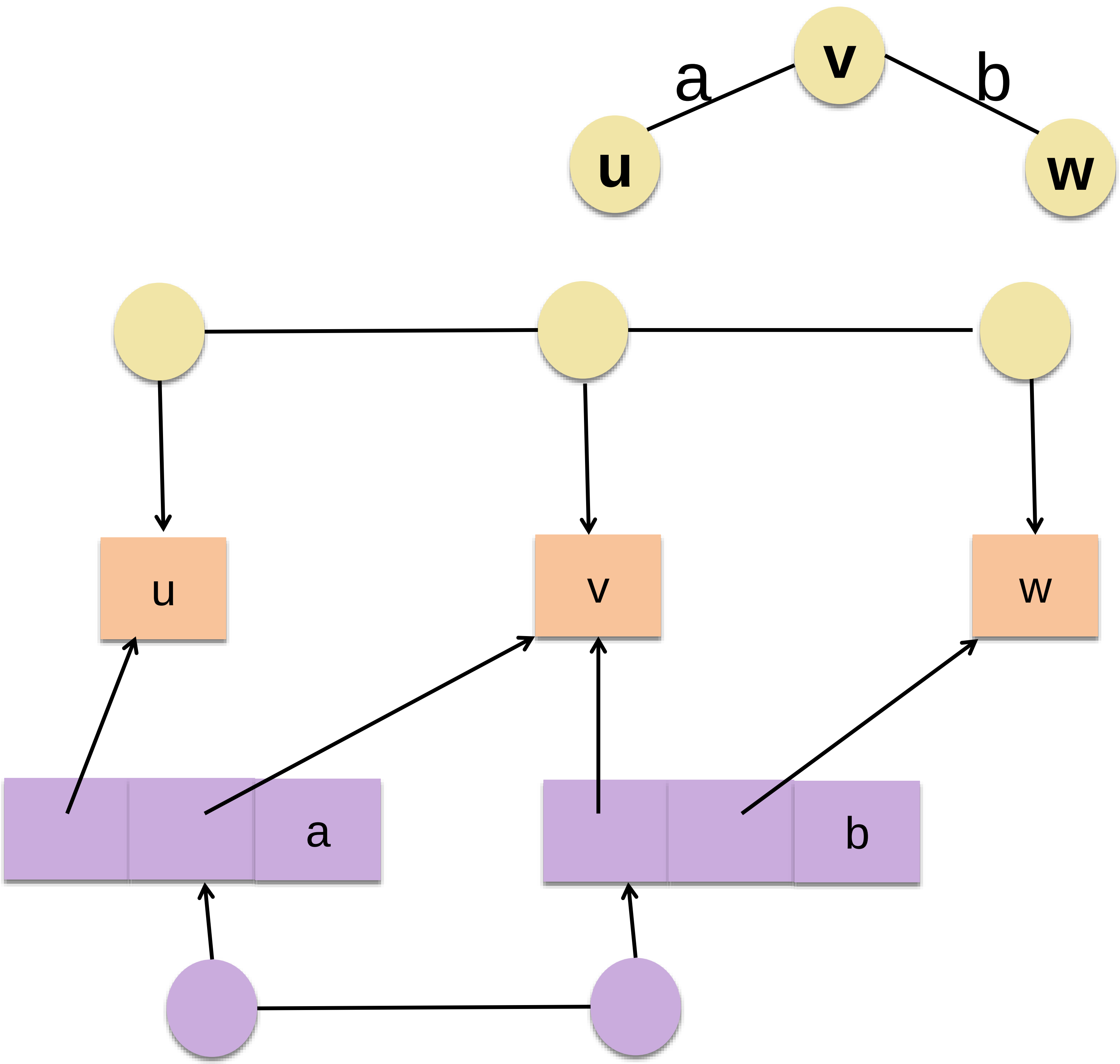

insertEdge(e, origin, dest)
O(1)

# Using Linked List (15)



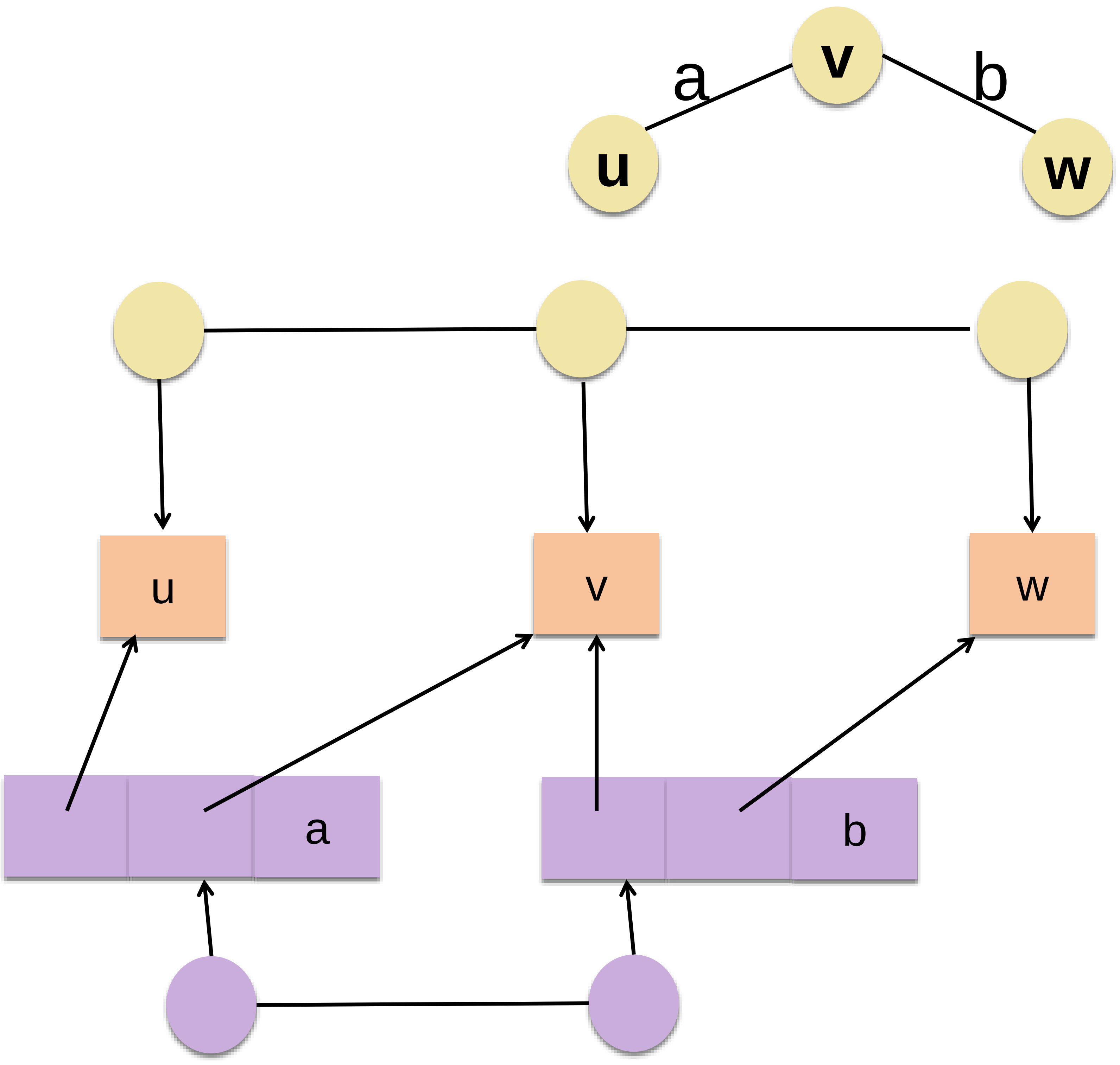

areAdjacent(v1, v2)
O(# of edges)

# Using Linked List (16)




removeEdge(e)
O(# of edges)

# Using Linked List (17)




Interesting one!
removeVertex(v)

O(?)

# Using Linked List (19)

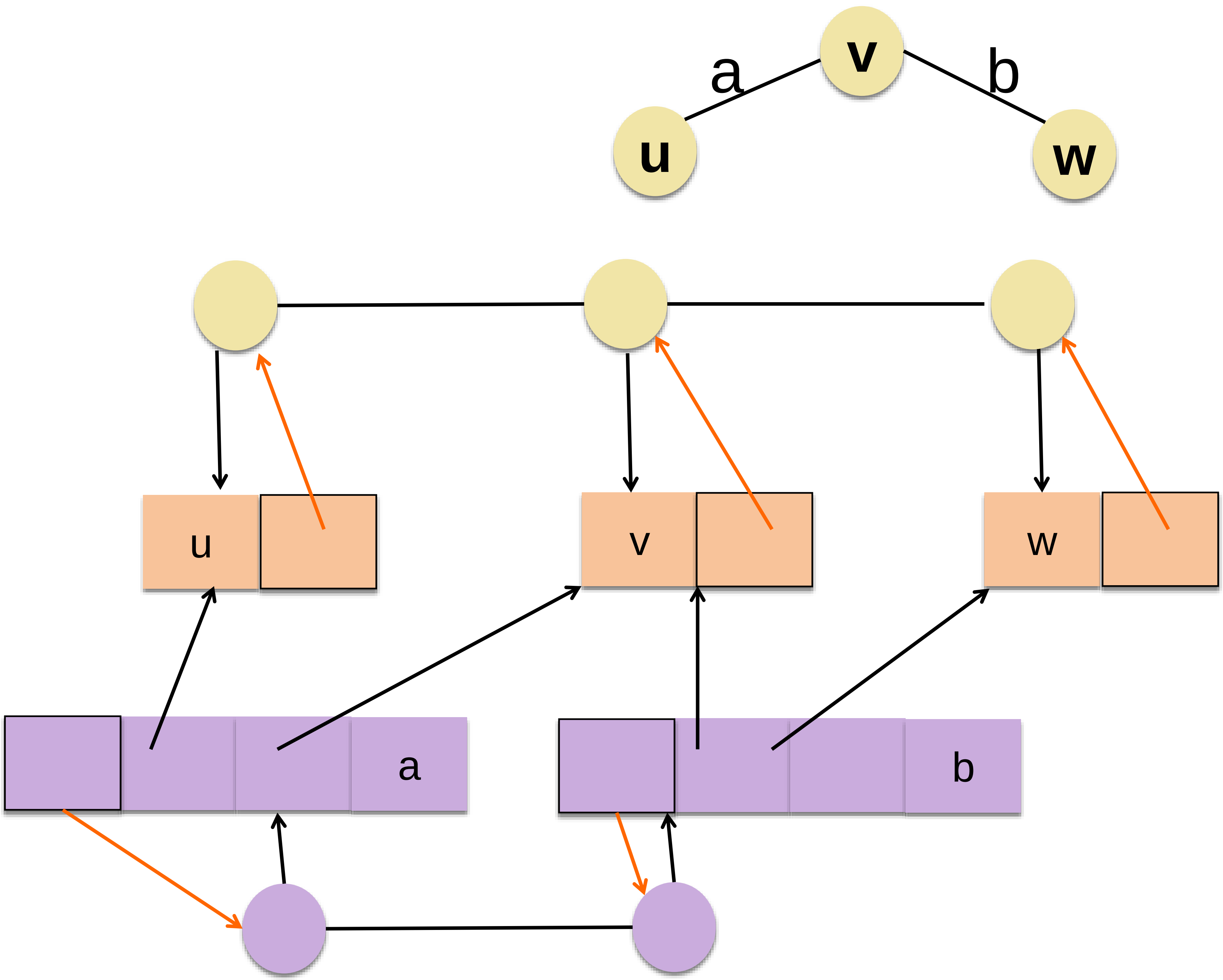
- Let’s do one more change to improve the efficiency of removeVertex and removeEdge

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

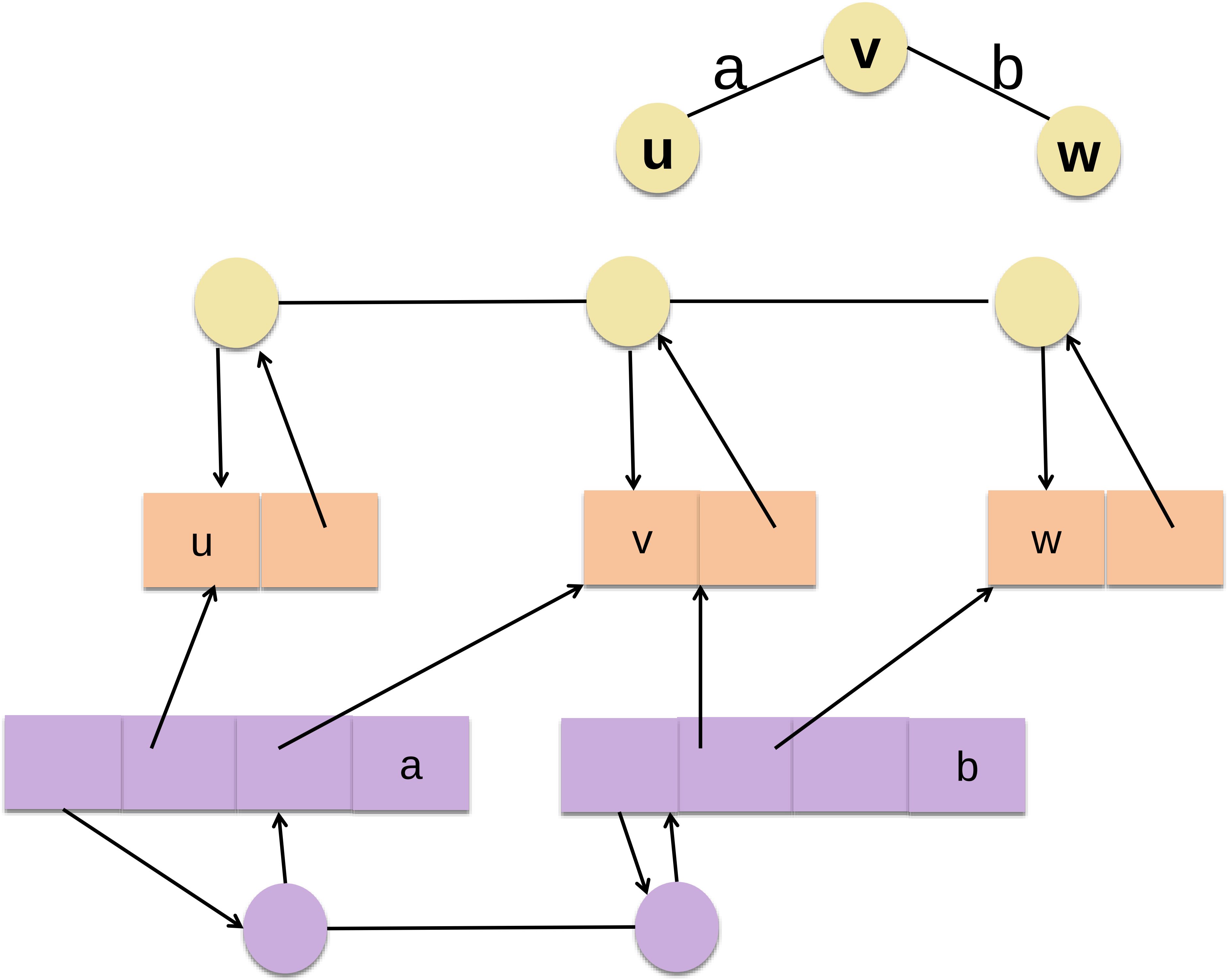
    - Vertex object: element, where am I in the vertex list

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

# Using Linked List (20)

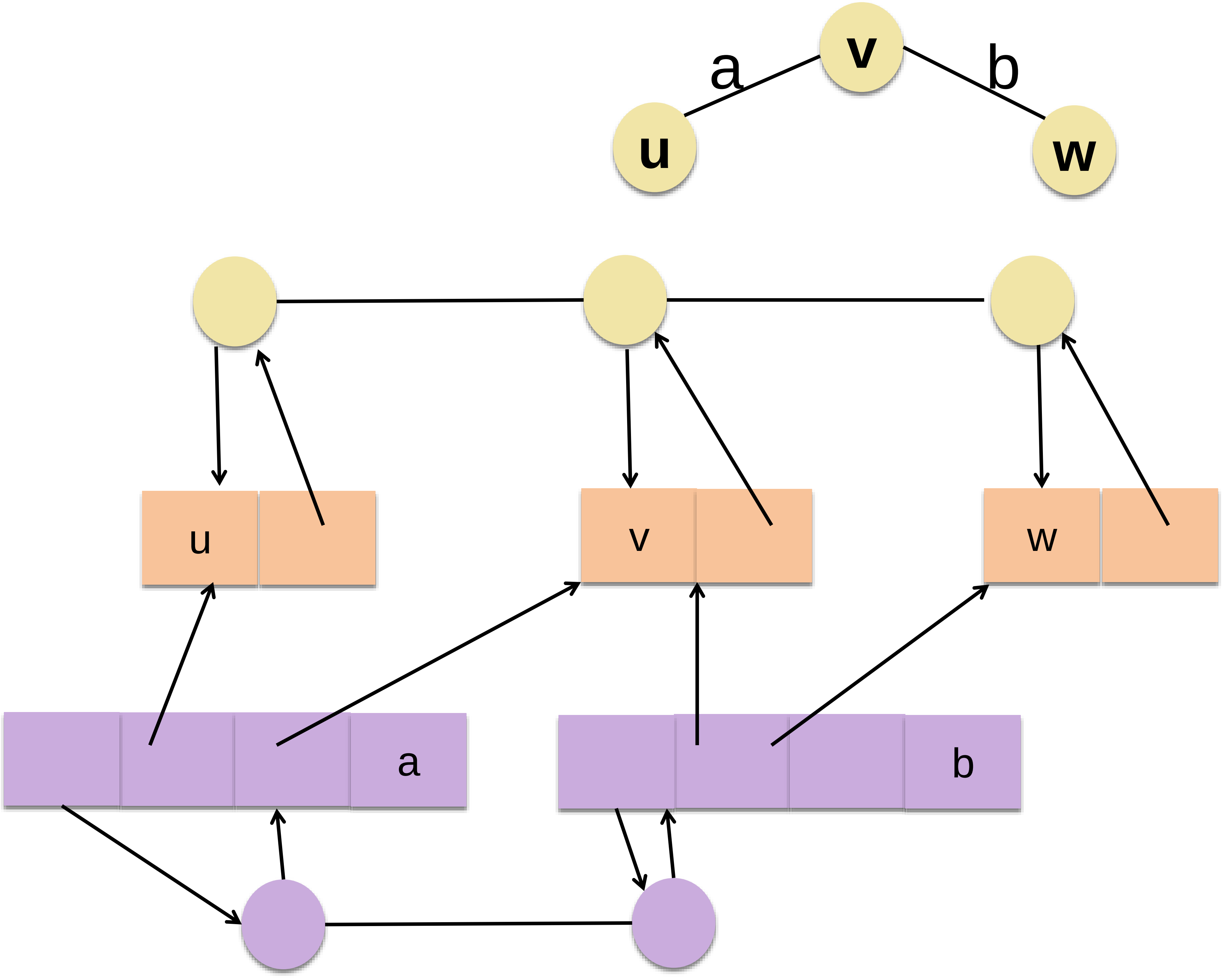


# Using Linked List (21)



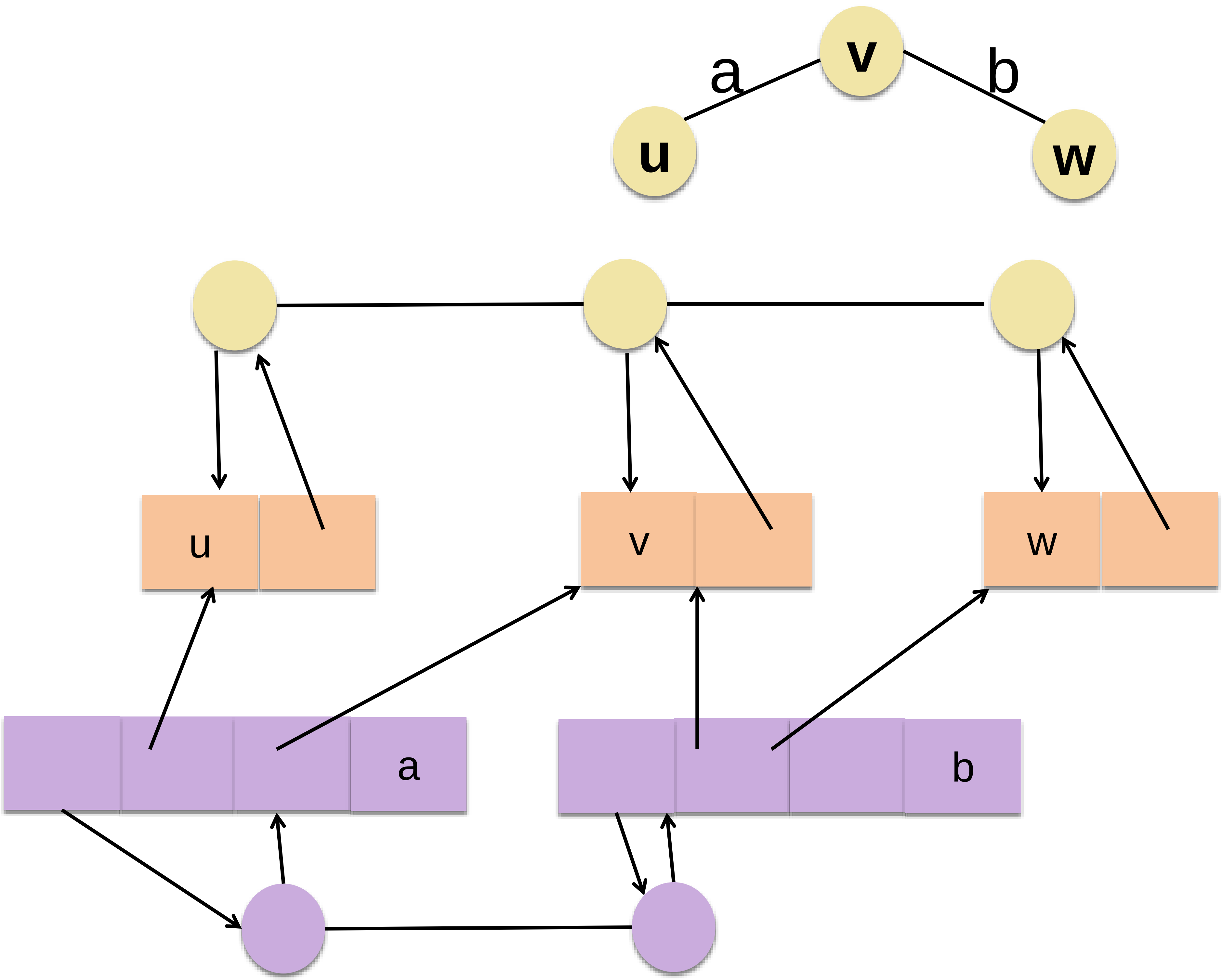

removeEdge(e)
O(1)

# Using Linked List (22)



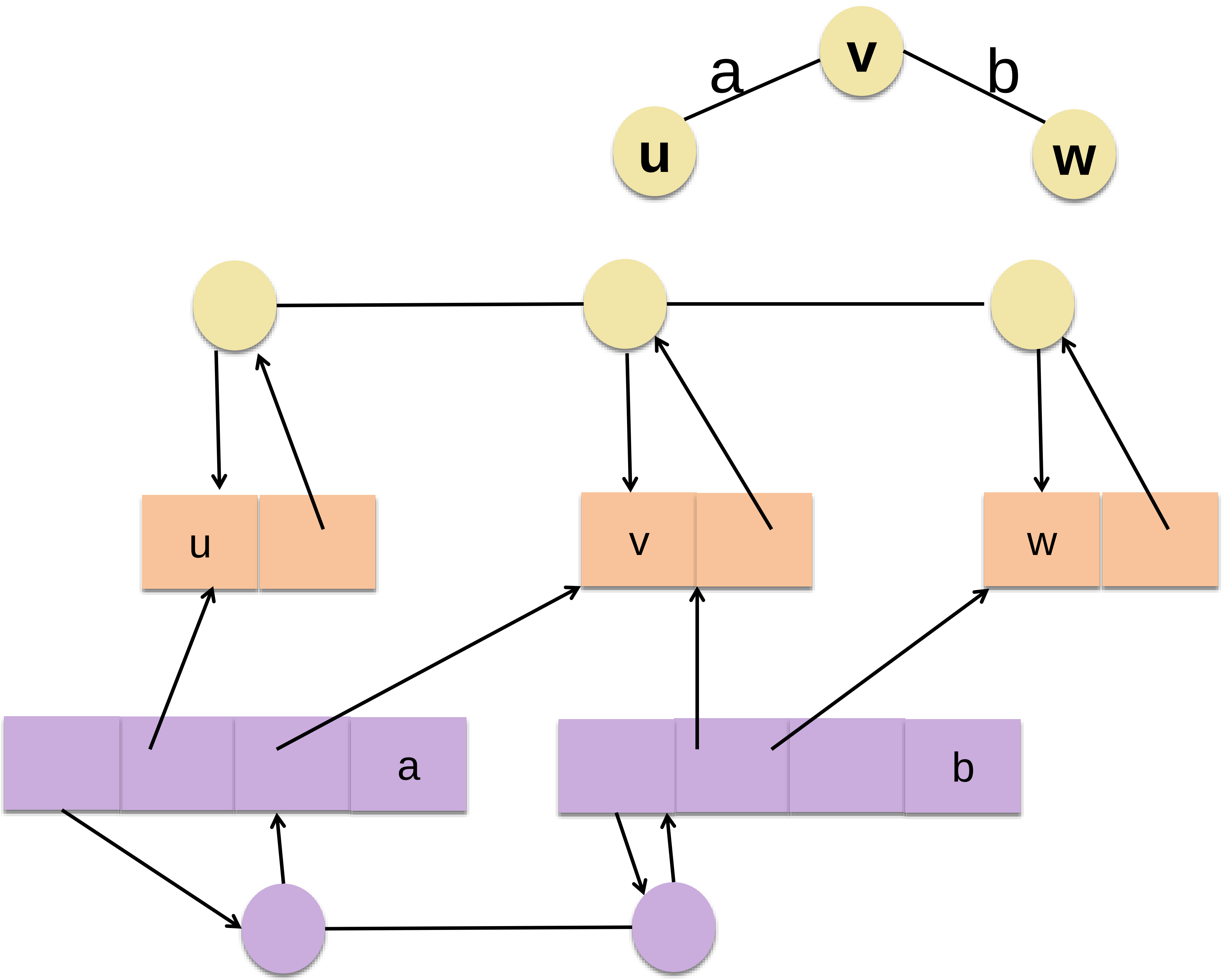

What about removeVertex(v)?

# Using Linked List (23)



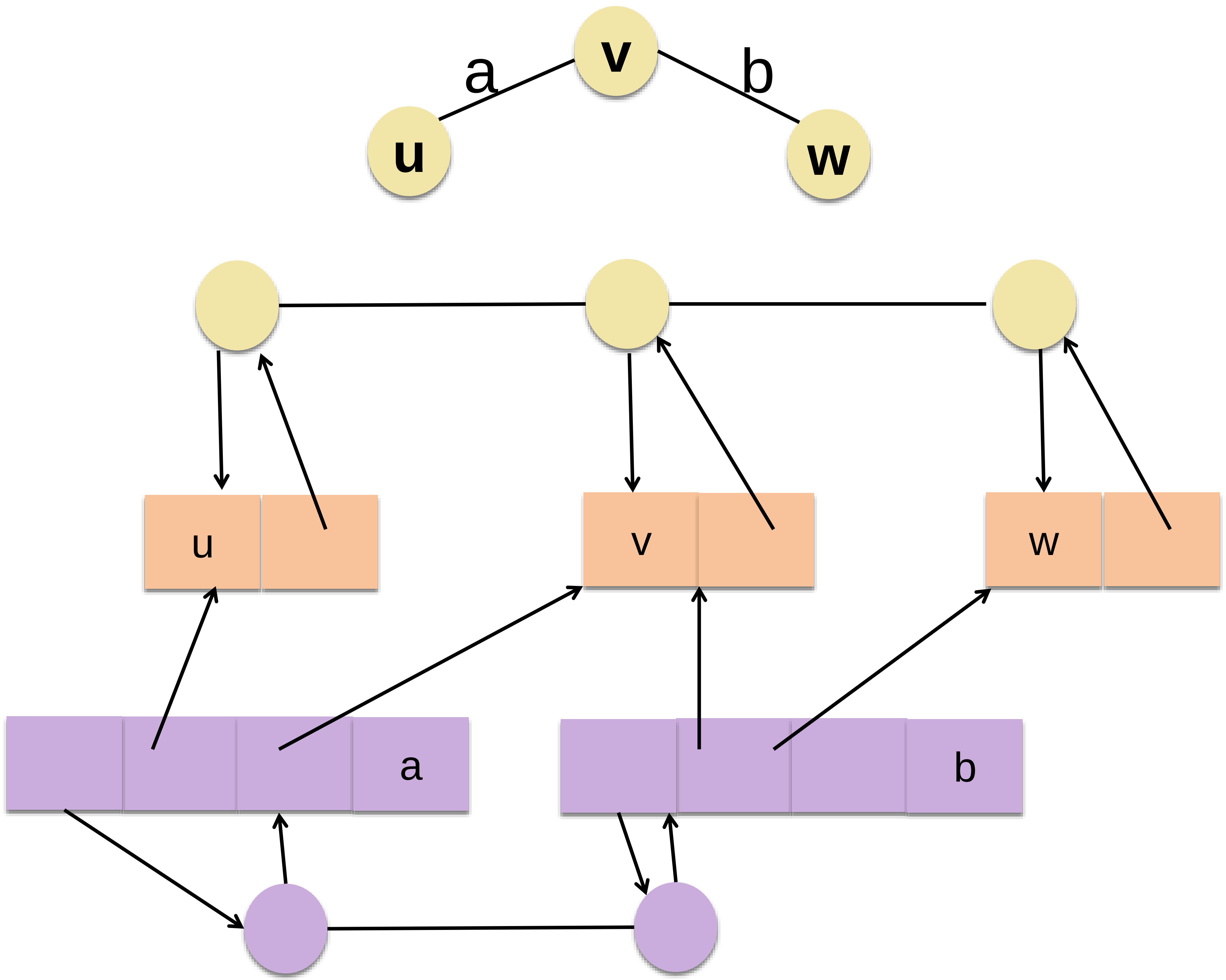

What about degree(v)?

# Using Linked List (24)



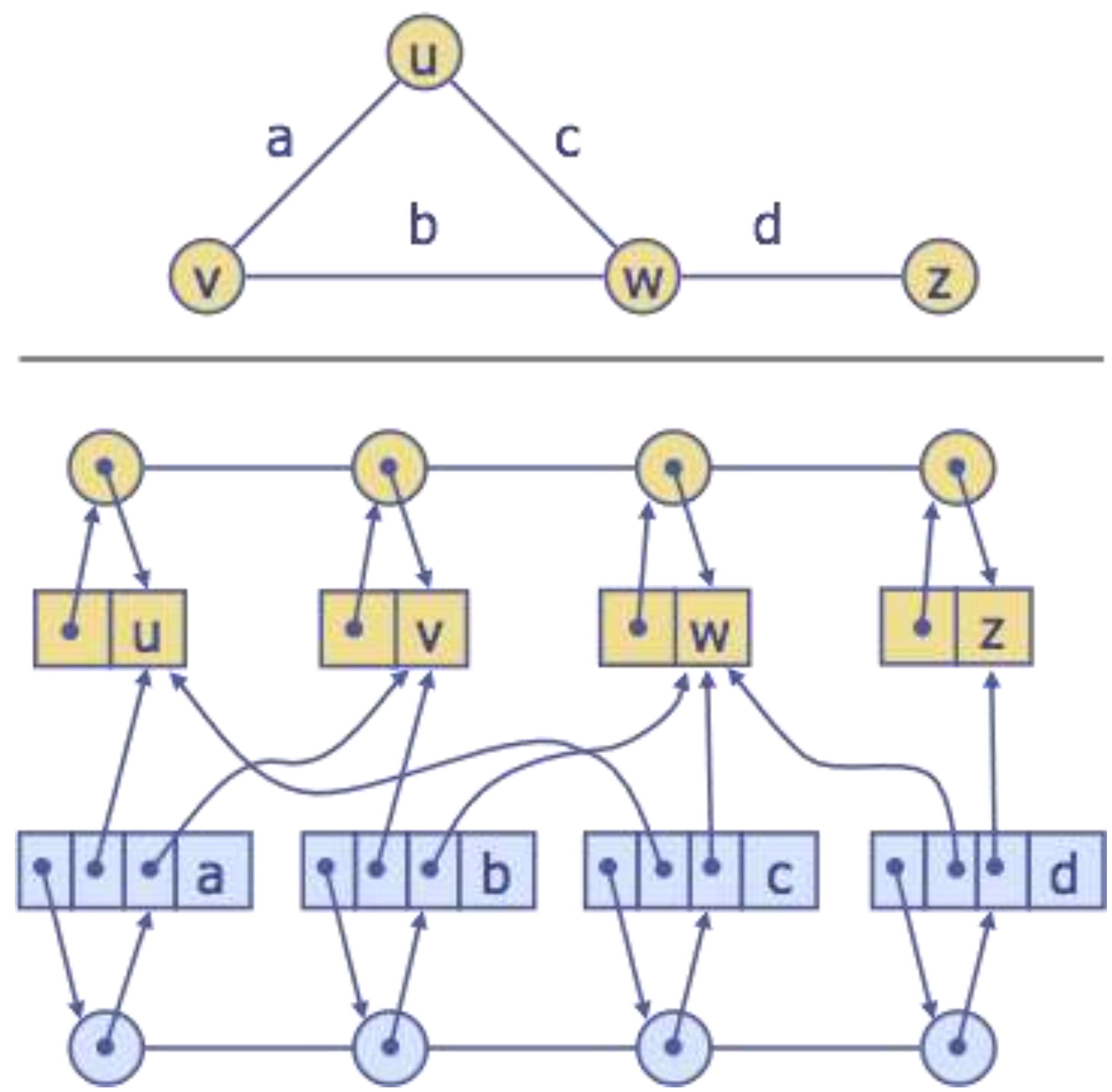

What about areAdjacent(v1, v2)?

# Edge List Structure

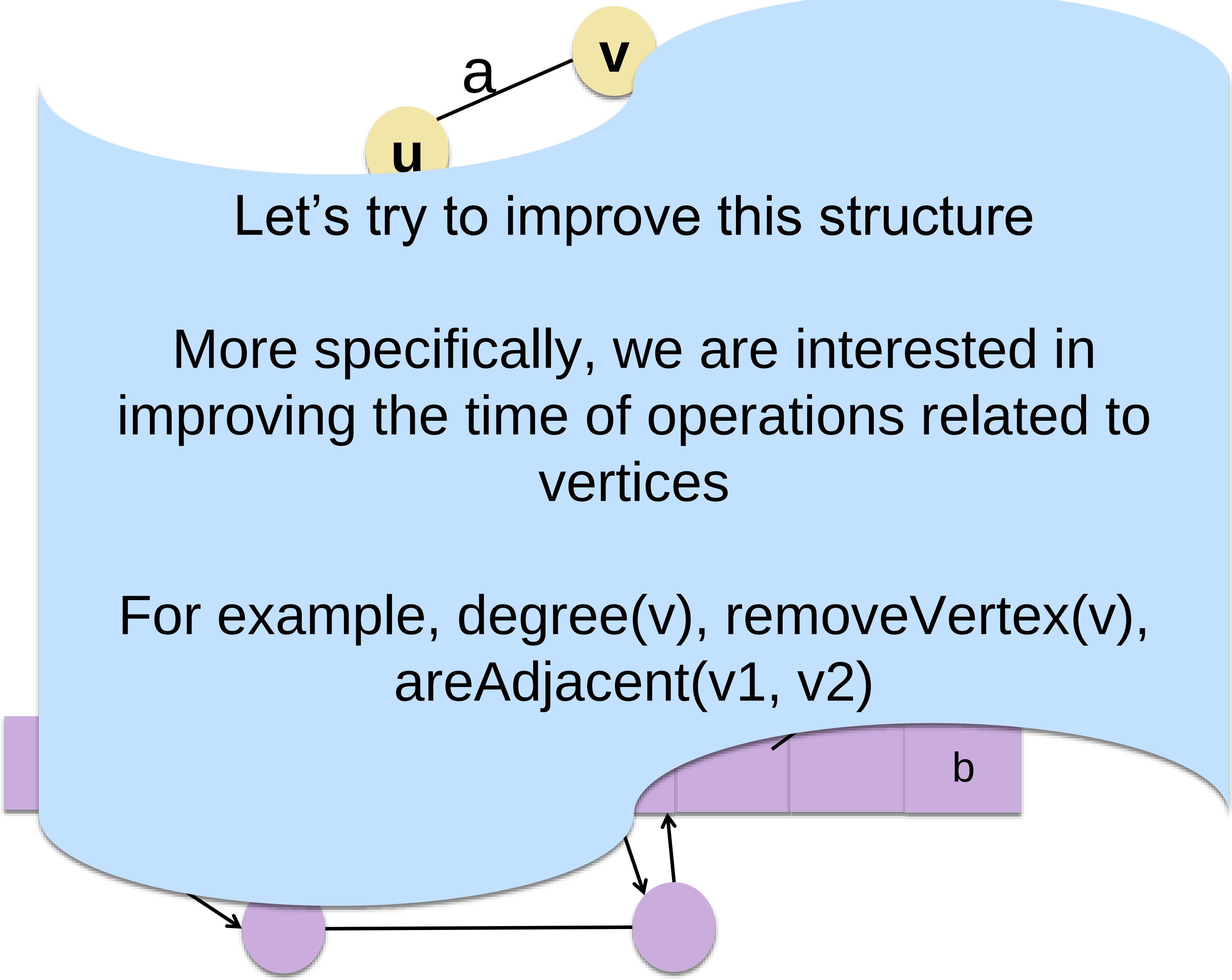


# Edge List Structure (2)

- **Another example for practice**



# Edge List Structure (3)

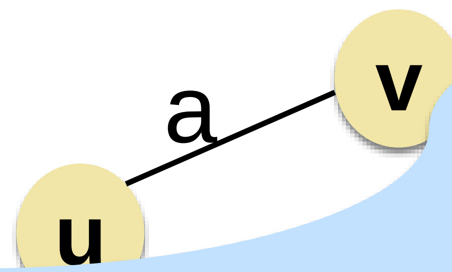



Let's try to improve this structure

More specifically, we are interested in improving the time of operations related to vertices

For example, degree(v), removeVertex(v), areAdjacent(v1, v2)

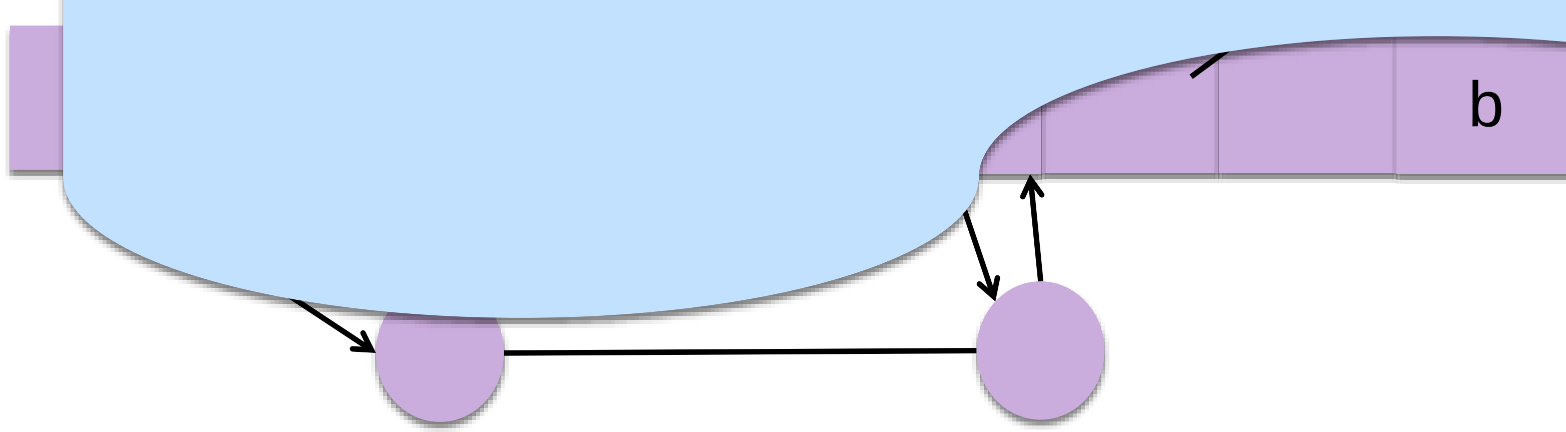
# Improvement



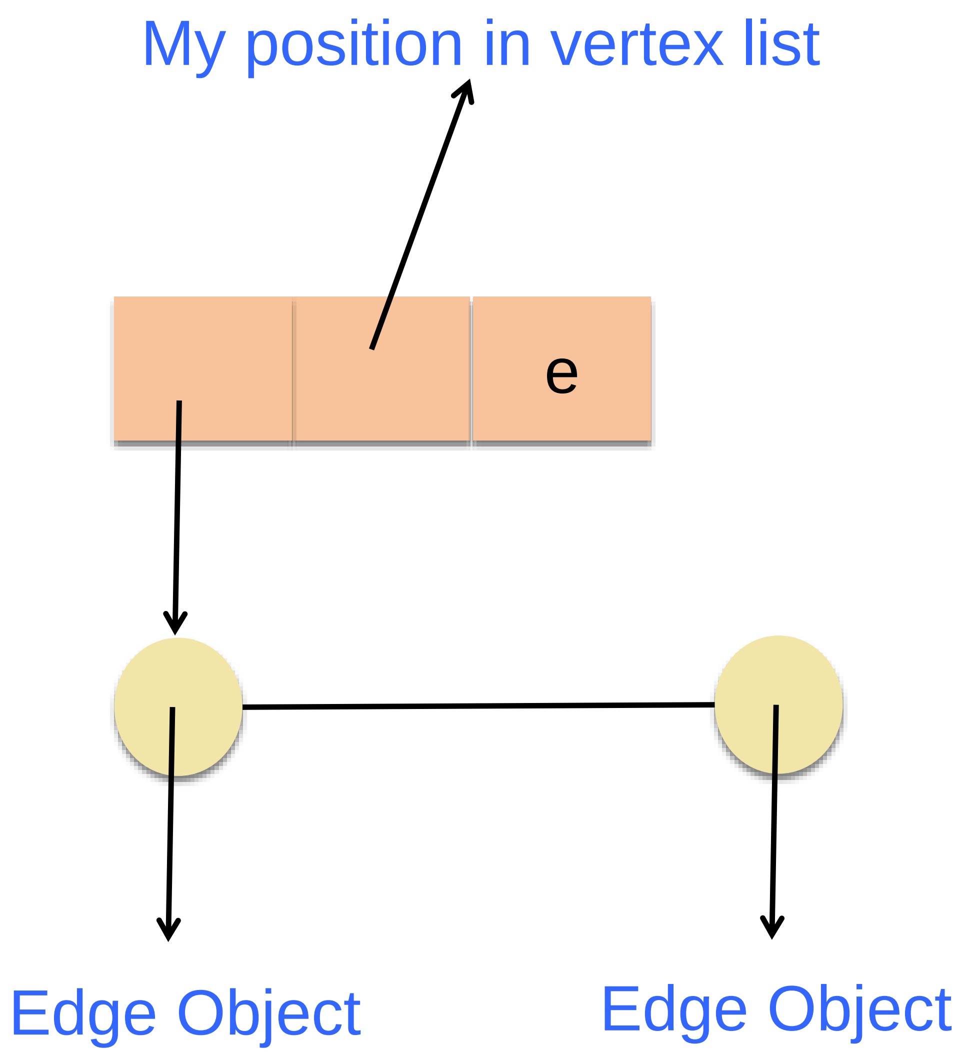

Vertex object is given more information

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

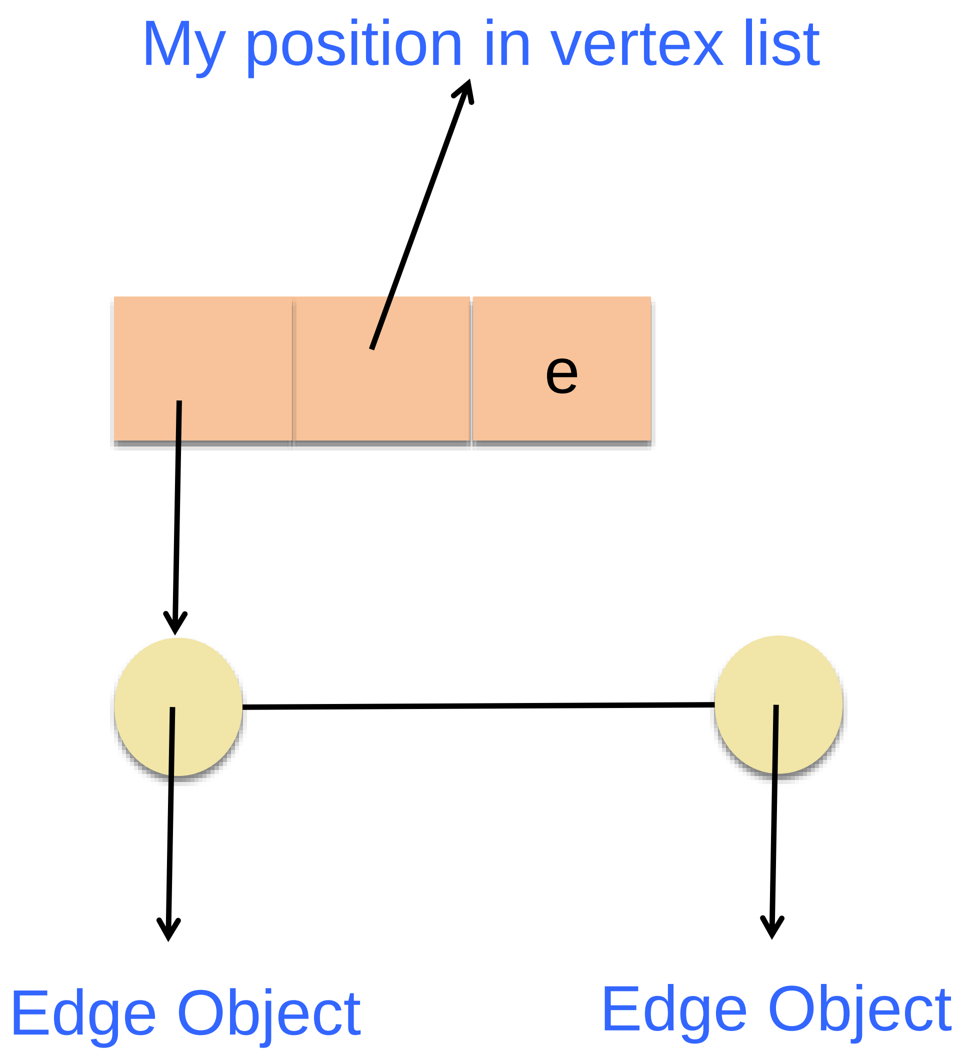



Vertex object is given more information

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

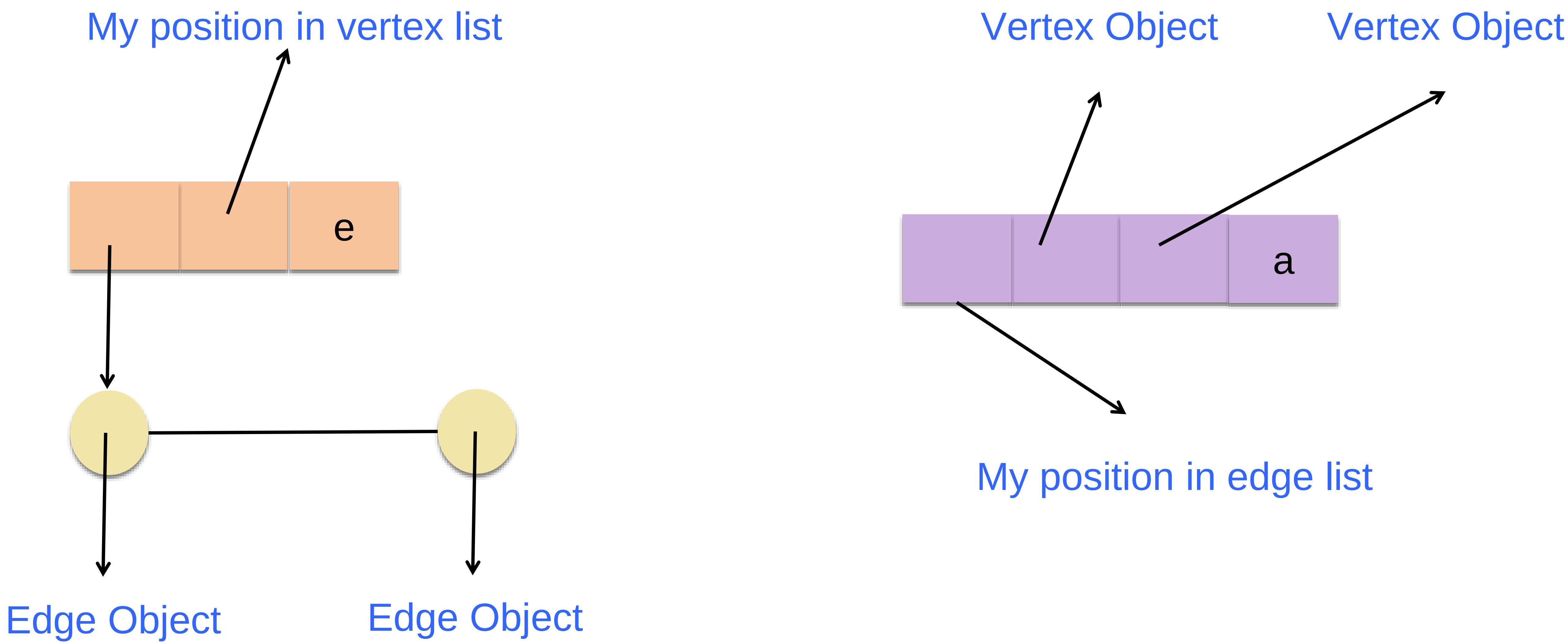



degree(v) ?

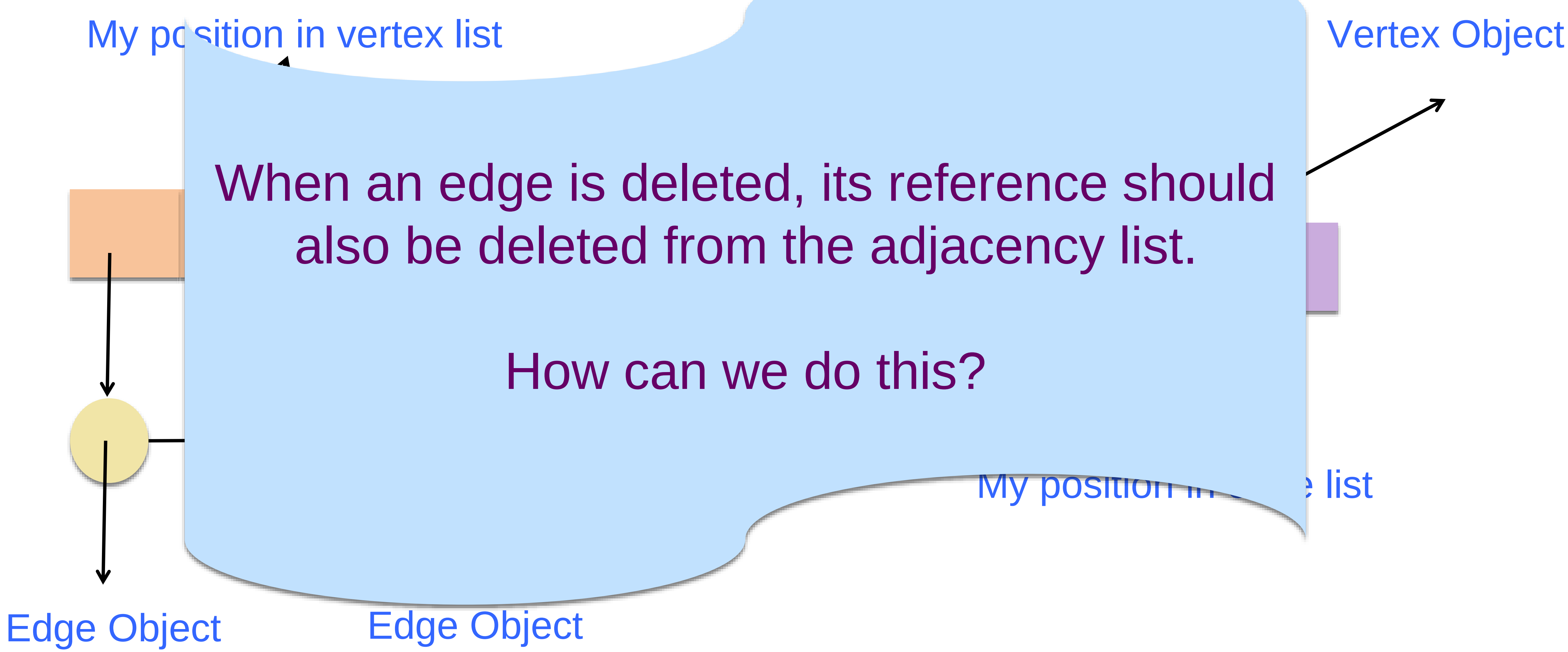
areAdjacent(v1, v2) ?

removeVertex(v) ?

# New Vertex with Previous Edge Object



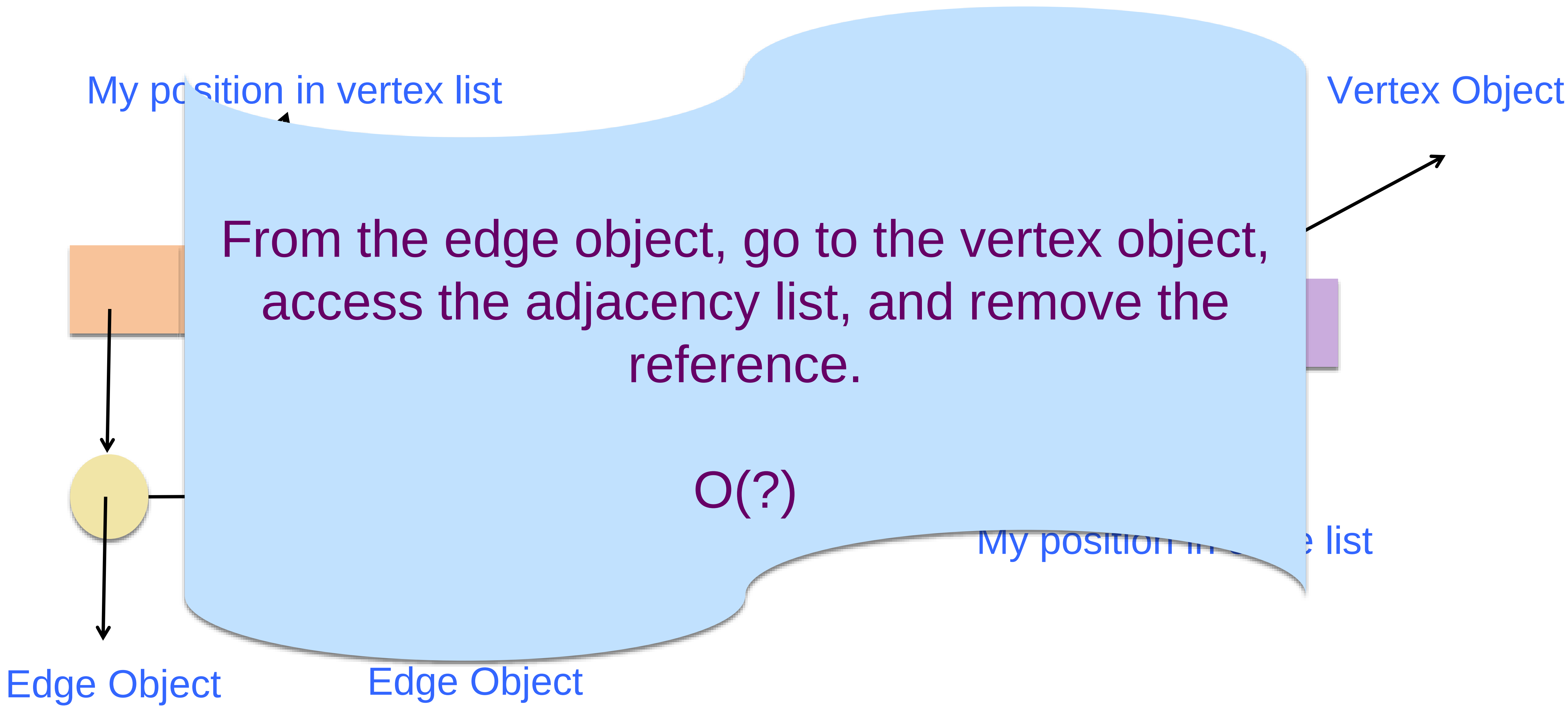
# Edge Removal




When an edge is deleted, its reference should also be deleted from the adjacency list.

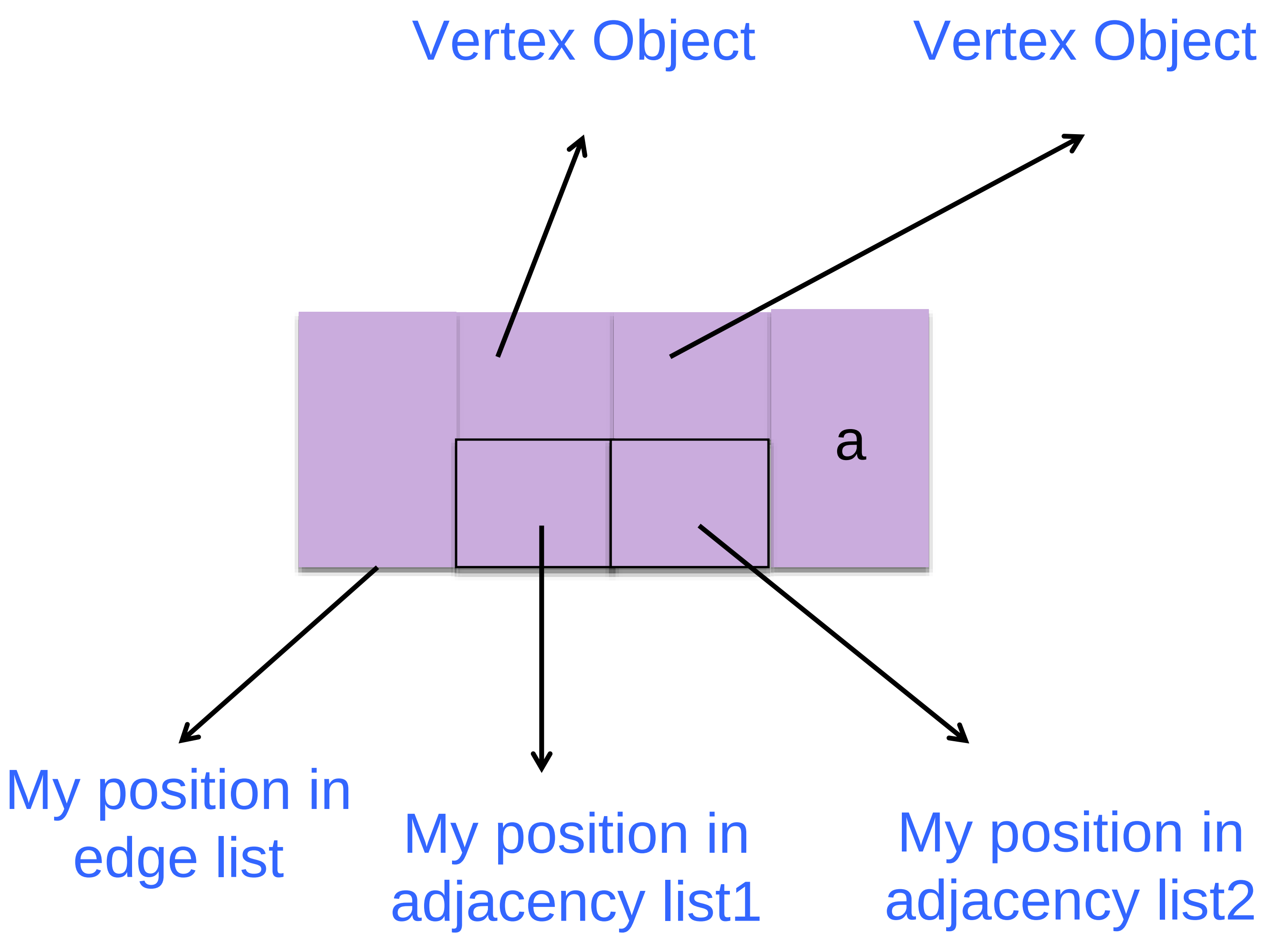
How can we do this?

# Removing Edge Reference from Adjacency List



# Removing Edge Reference from Adjacency List (2)

# Updated Edge Object



# Updated Edge Object (2)

Vertex Object

Vertex Object

Removing an edge – O(1)

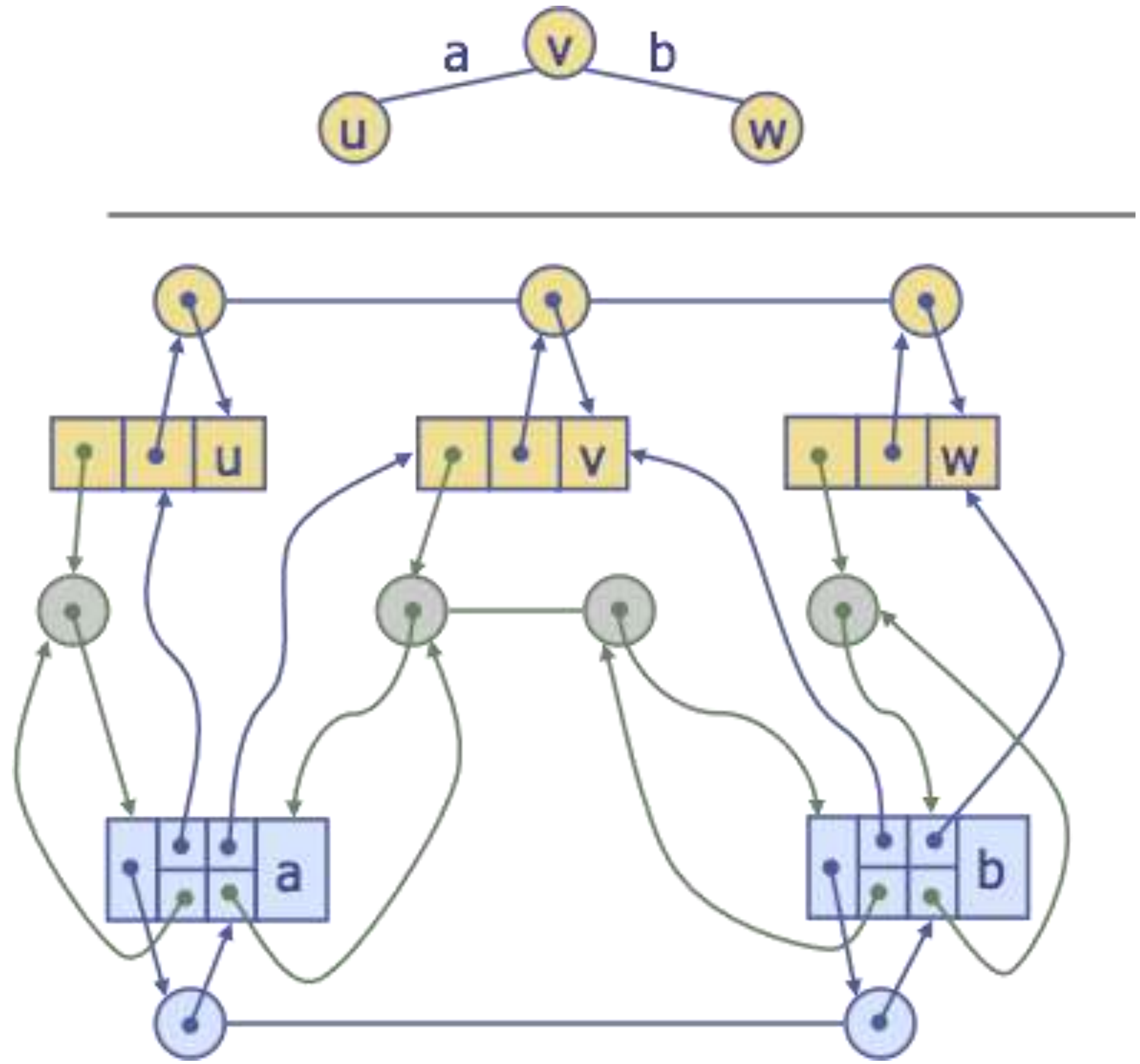
edge list

position in
adjacency list1

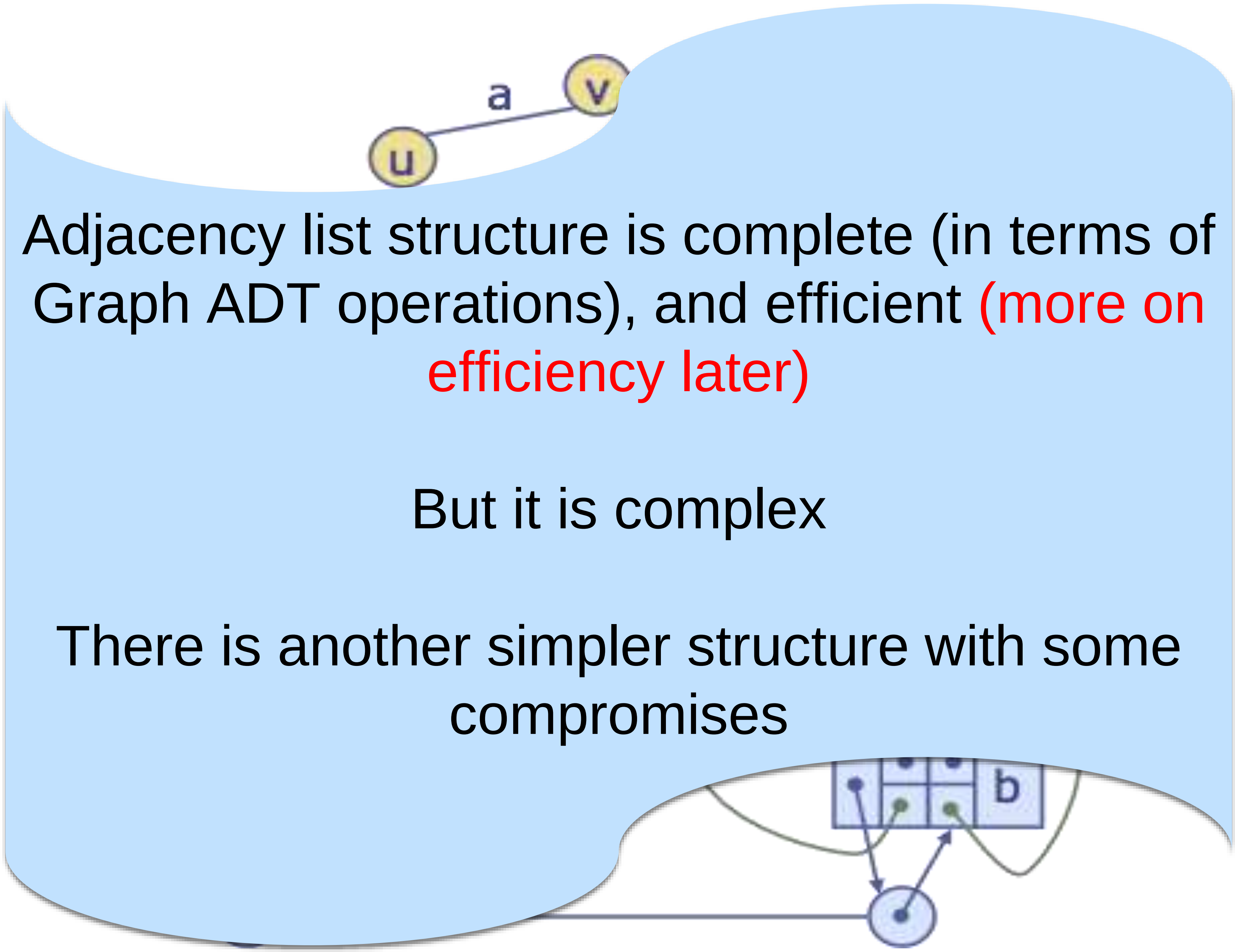
My position in
adjacency list2

# Adjacency List Structure

- **The entire structure!**



# Adjacency List Structure (2)

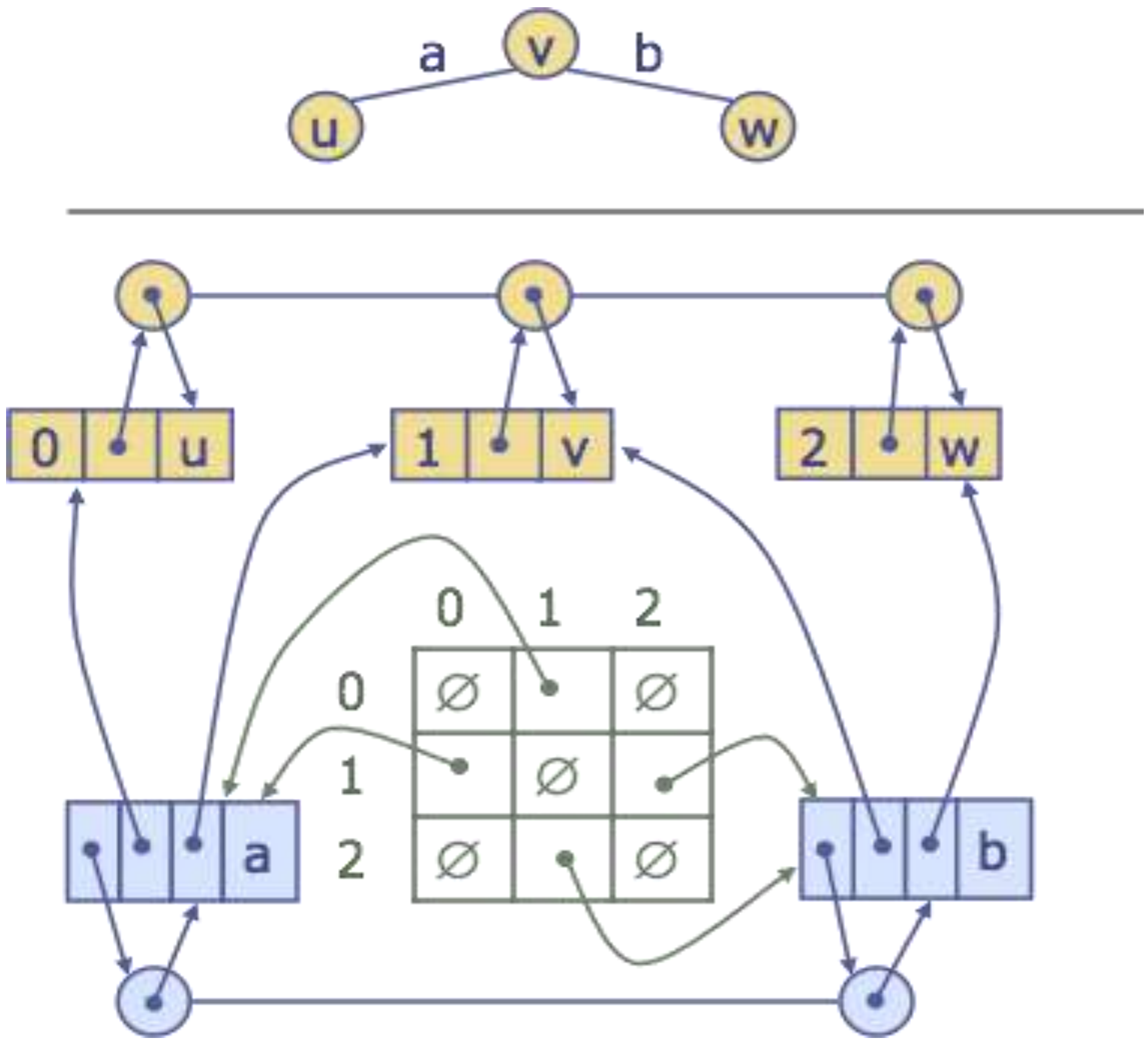



Adjacency list structure is complete (in terms of Graph ADT operations), and efficient (more on efficiency later)

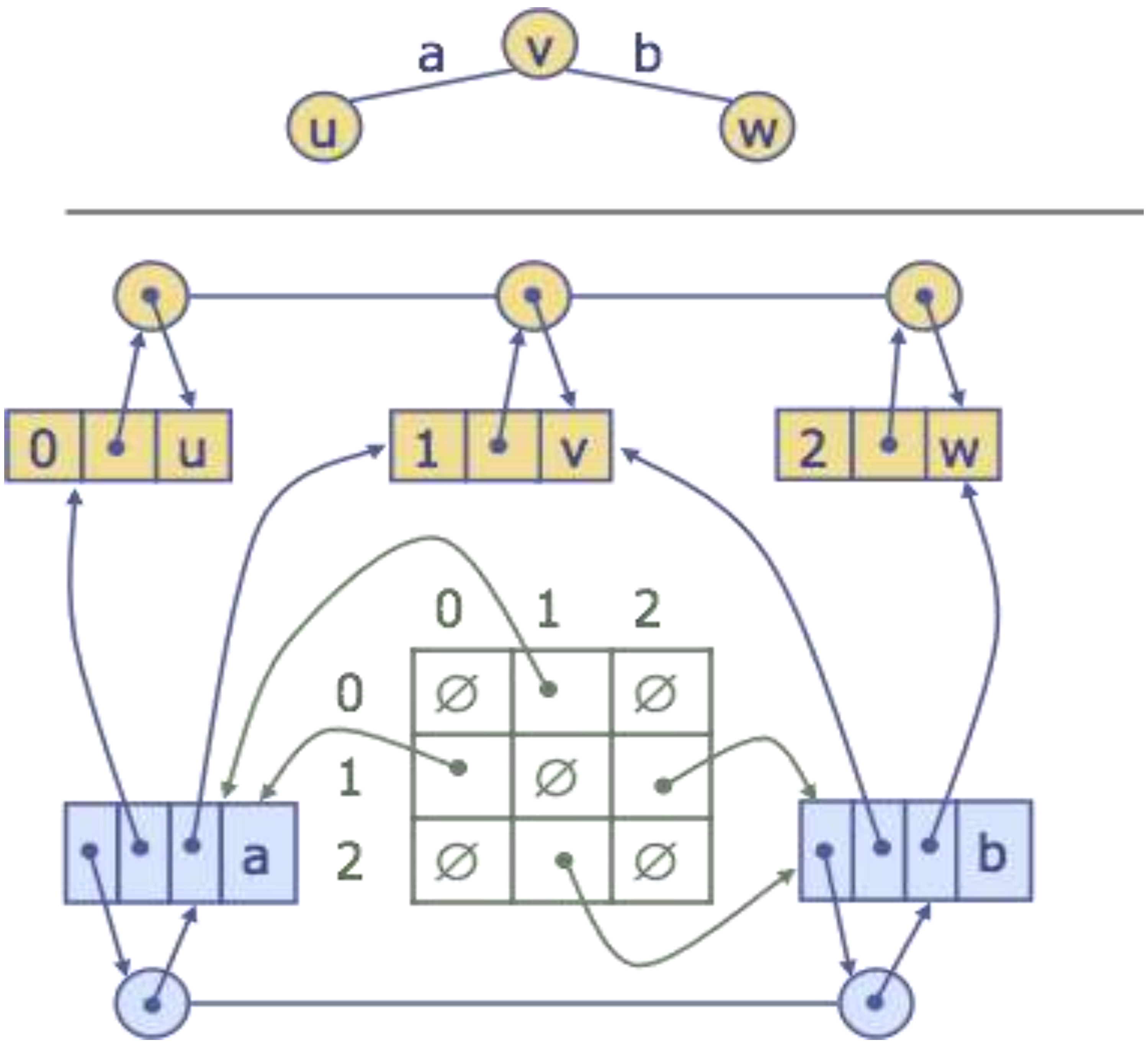
But it is complex

There is another simpler structure with some compromises

# Adjacency Matrix Structure



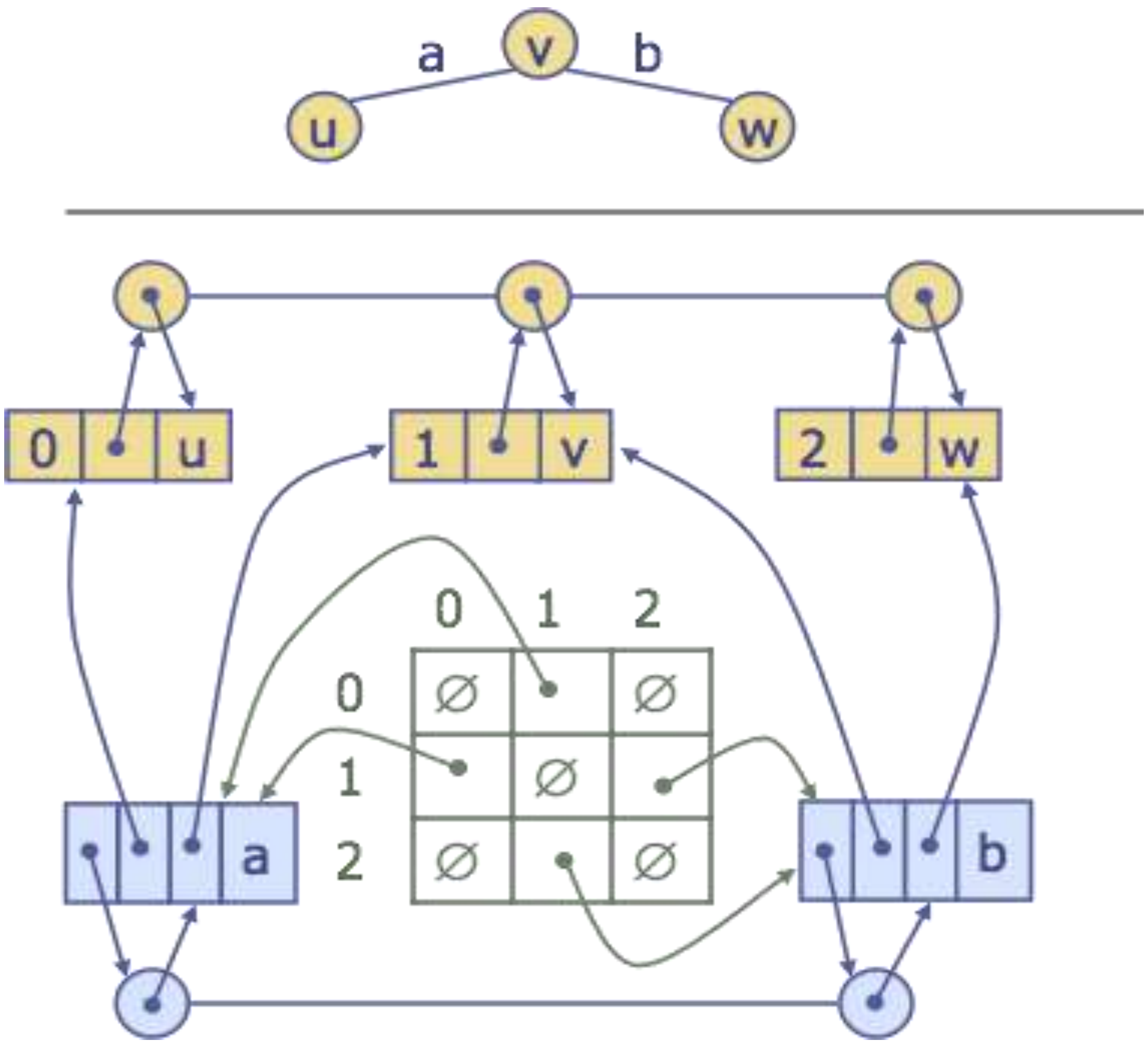
# Adjacency Matrix Structure (2)




Edge object goes back to "Edge List" stage!

Only knows four things!

# Adjacency Matrix Structure (3)

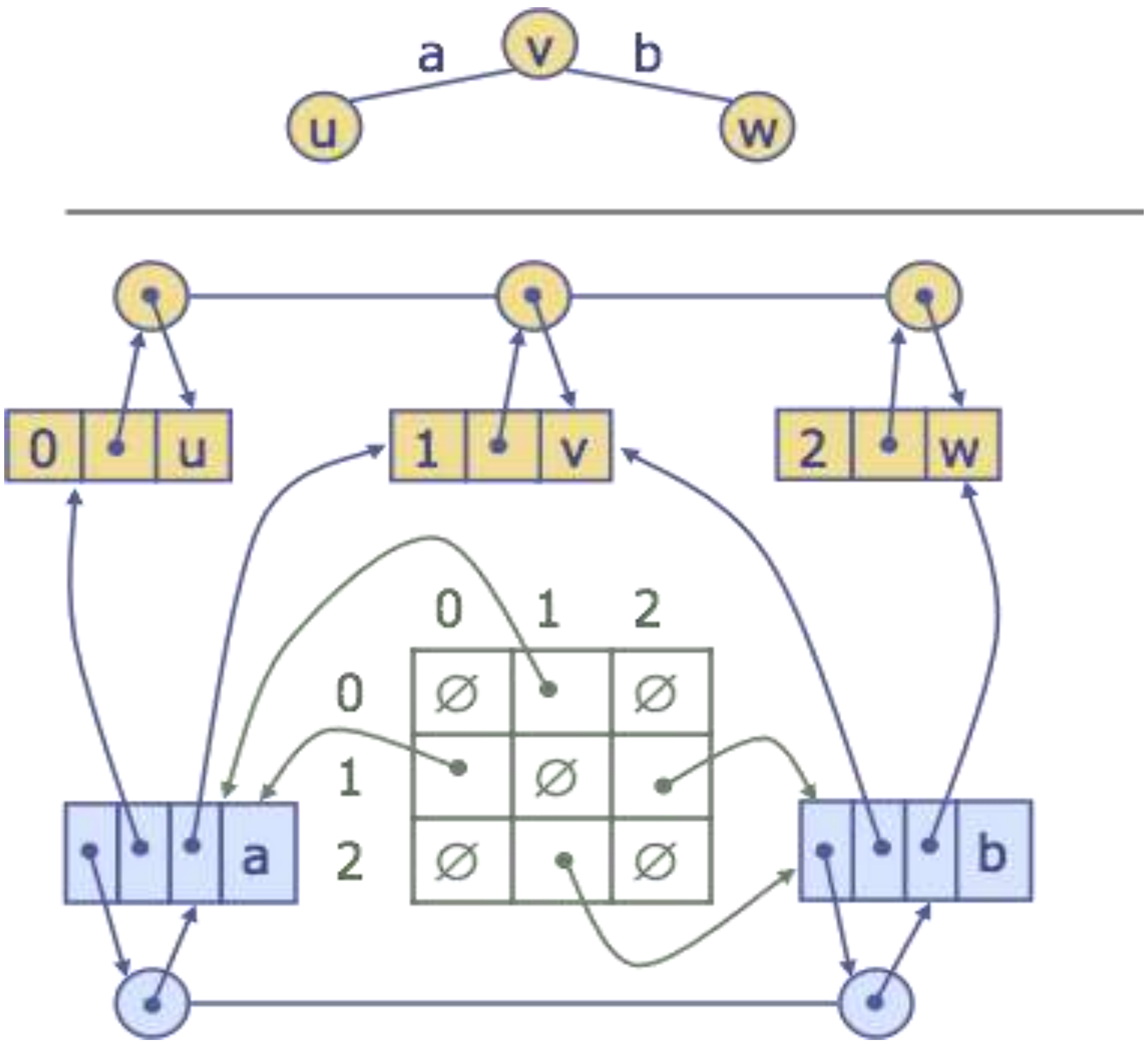



A nxn matrix is introduced – Adjacency Matrix

Each cell stores a reference to an edge object

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

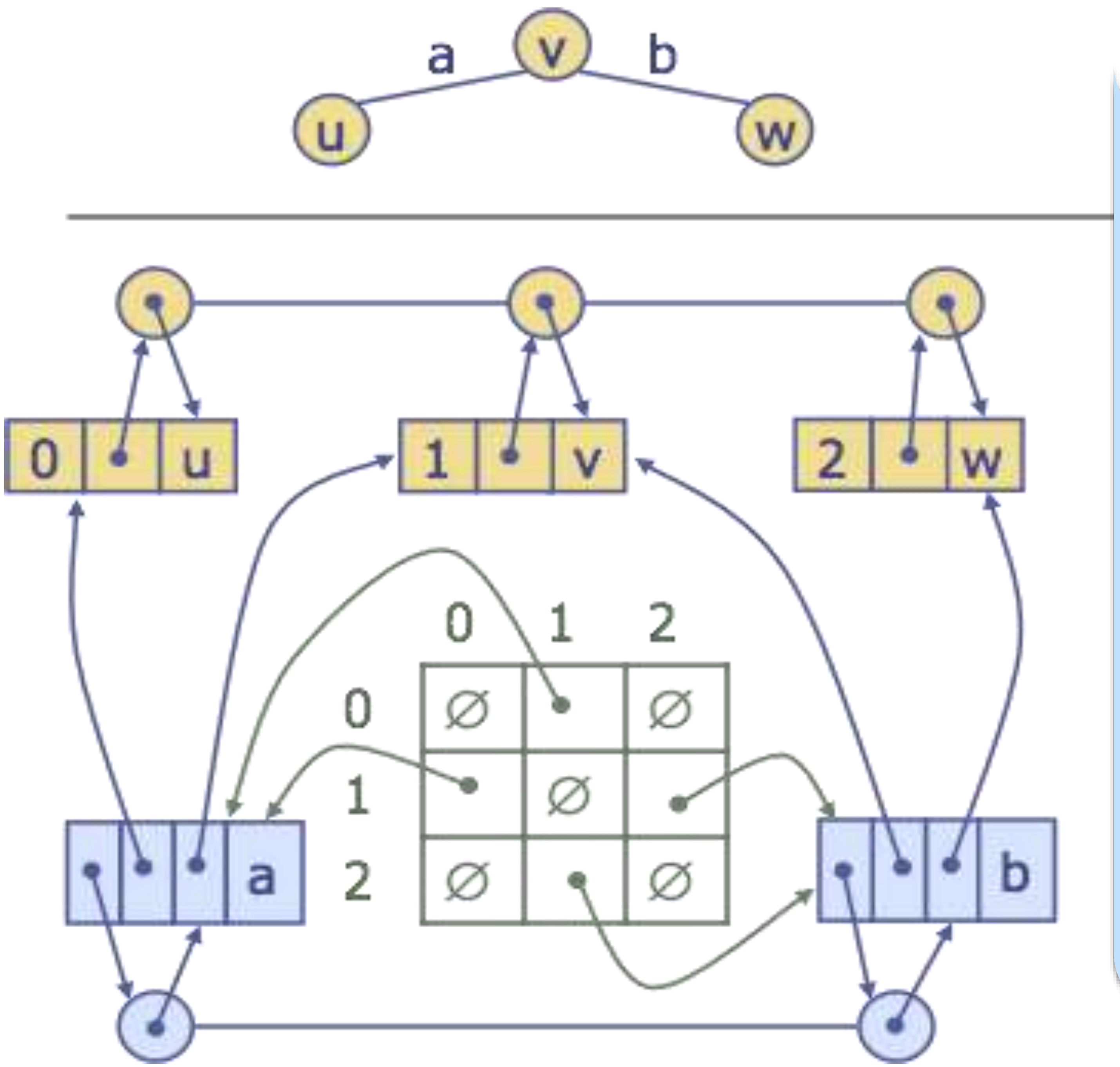
# Adjacency Matrix Structure (4)




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 (5)




incidentEdges(v) – O(?)

areAdjacent(v0, v1) – O(?)

insertVertex(v) – O(?)

insertEdge(e, o, d) – O(?)

removeVertex(v) – O(?)

removeEdge(e) – O(?)

# Graph Representations

| ▪ $n$ vertices, $m$ edges<br>▪ no parallel edges<br>▪ 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