# Data Structures and Algorithms

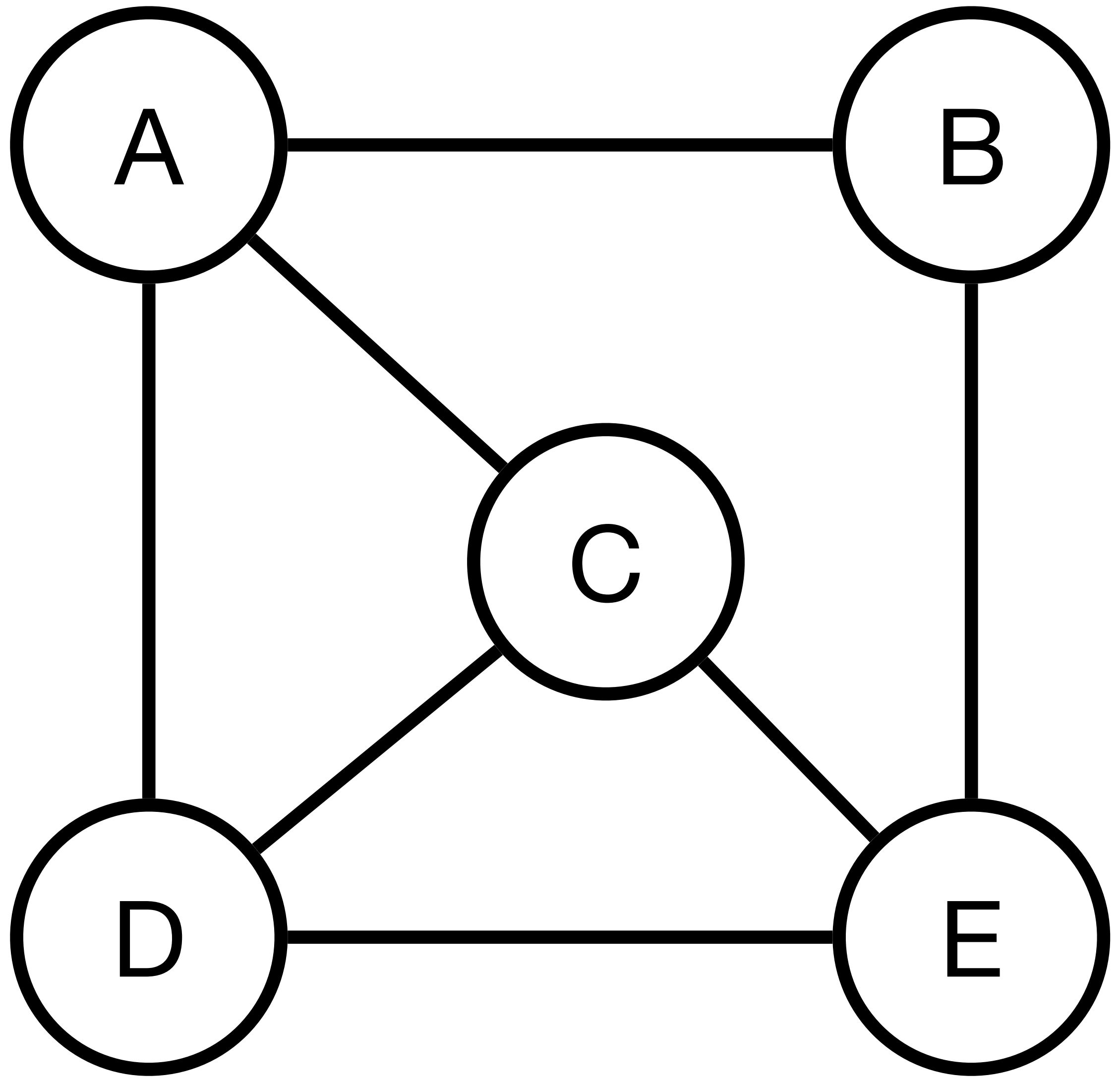
## Week 9 - Graphs, ADT

**Subodh Sharma and Rahul Garg**

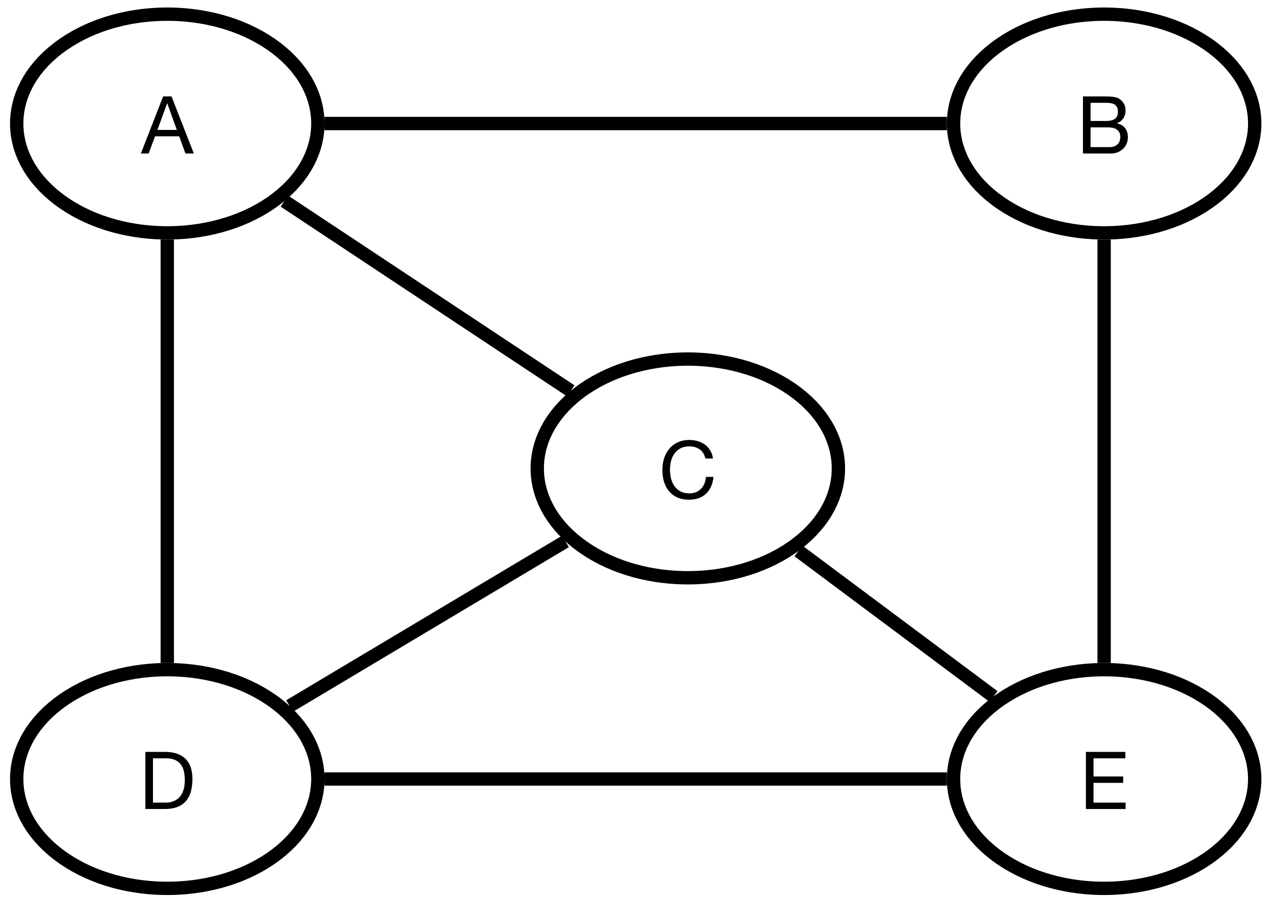
**{svs,rahulgarg}@iitd.ac.in.**

# Graphs — Definitions

- A graph $G = (V, E)$ consists of set of vertices and a set of edges
- Edges are defined by a pair of vertices, i.e., $(v, w) \in E$ where $v, w \in V$
  - If the pair of vertices is **ordered** then the graph is called **directed**
- Example: $V = \{A, B, C, D, E\}, E = \{(A, B), (B, E), (E, D), (A, D), (A, C), (C, E)\}$



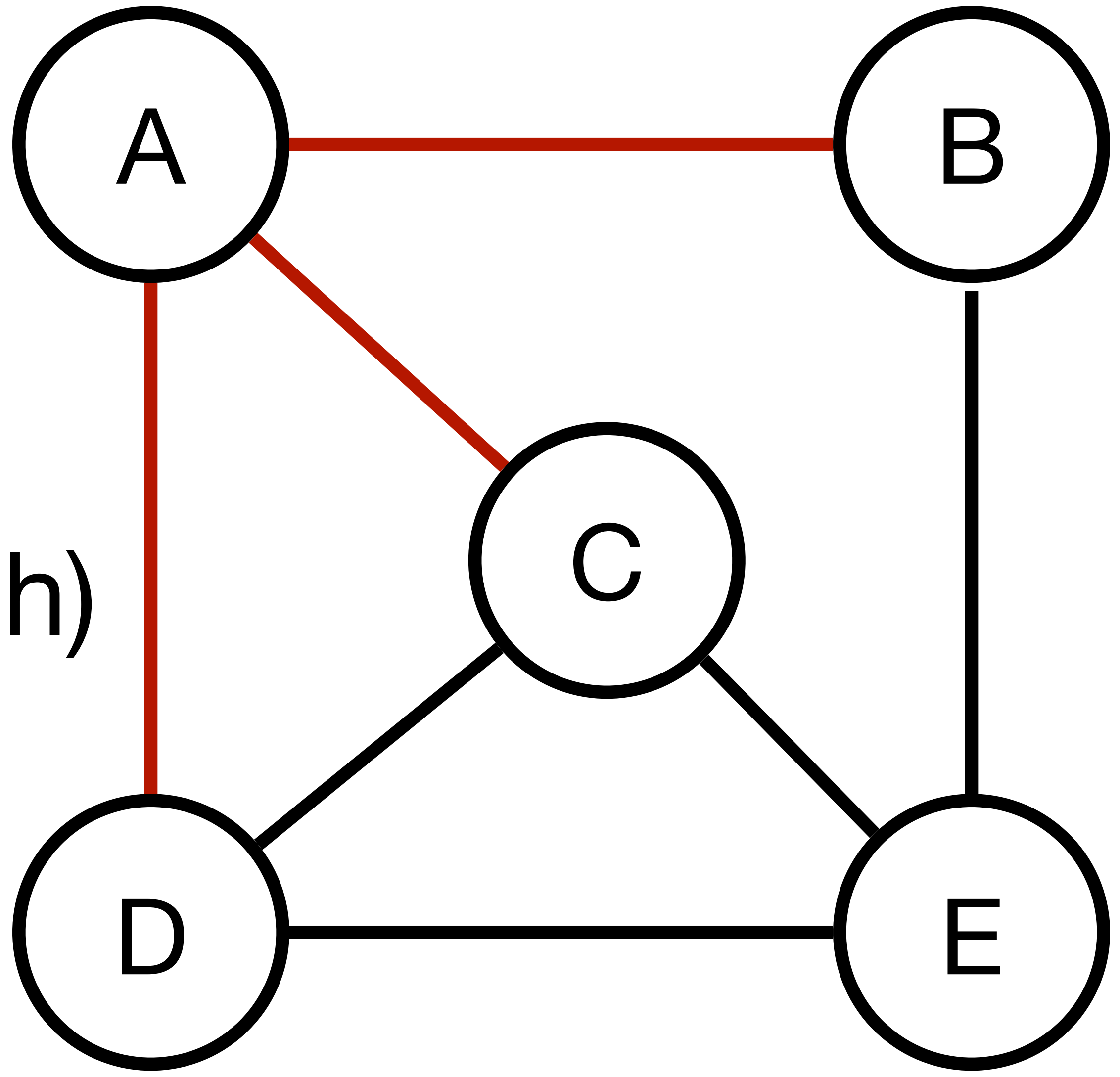
# Graphs — Definitions



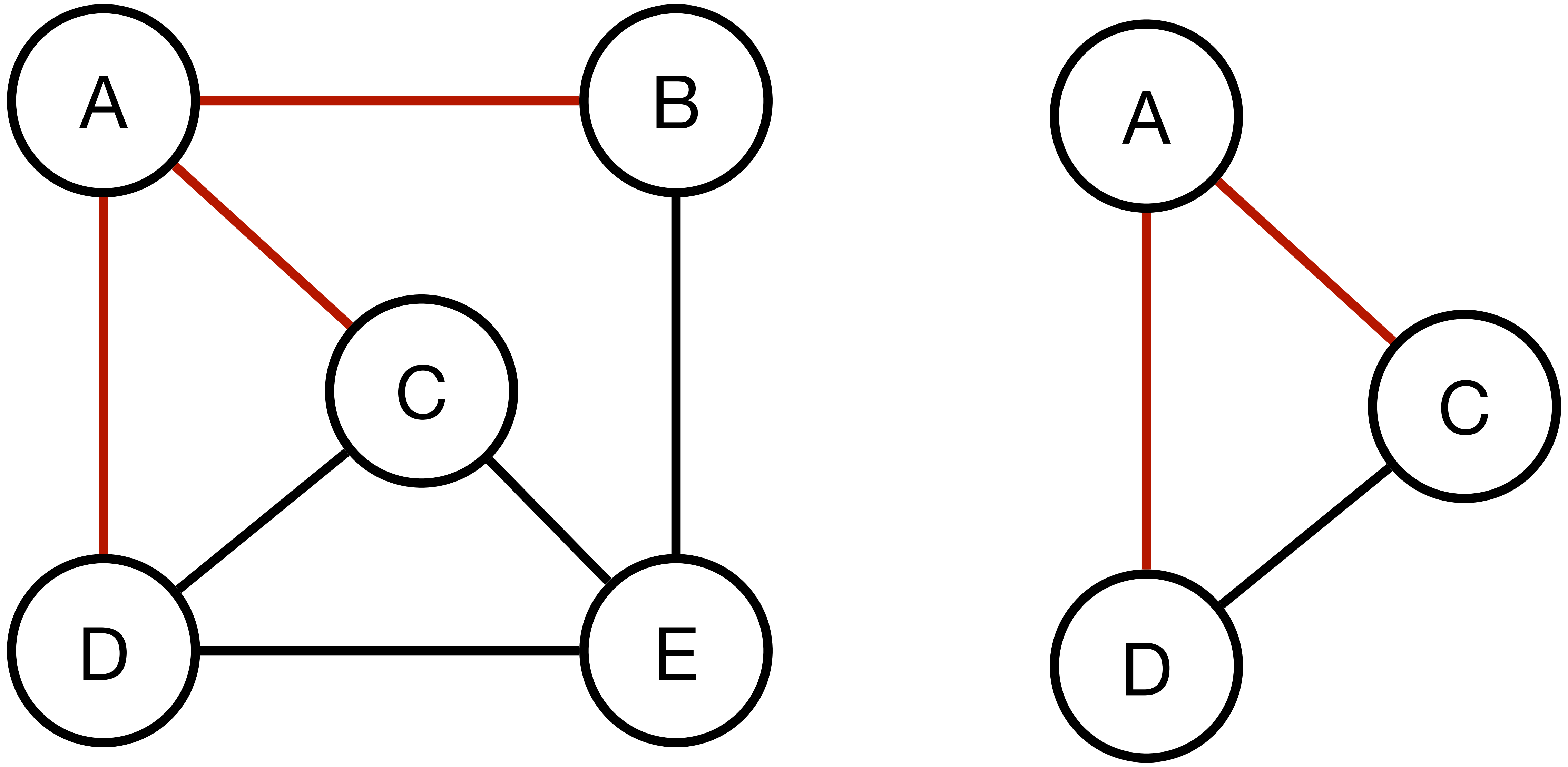

- **Adjacent vertices**: $v$ is adjacent to $w$ iff $(w, v) \in E$
  - In an undirected graph $v$ is adjacent to $w$ and vice-versa
- **Weighted edges**: Sometimes edges can have weights or costs
- **Degree** (of a vertex): Number of adjacent vertices
  - What is the sum of the degrees of all vertices in an undirected graph?
  - Ans: Twice the number of edges! Why?

# Graphs — Definitions

- **Path**: Sequence of vertices $w_1, w_2, \ldots, w_n$ s.t. $(w_i, w_{i+1}) \in E$
- **Simple path:** Path with no repeated vertices
  - Eg: ADAC? ADCE?
- **Cycle:** Simple path where first and last vertices are the same
  - Eg: ACDA
  - A directed graph w/o cycles is called **DAG** (directed acyclic graph)
- **Connected graph:** Any two vertices are connected by some path



# Graphs — Definitions



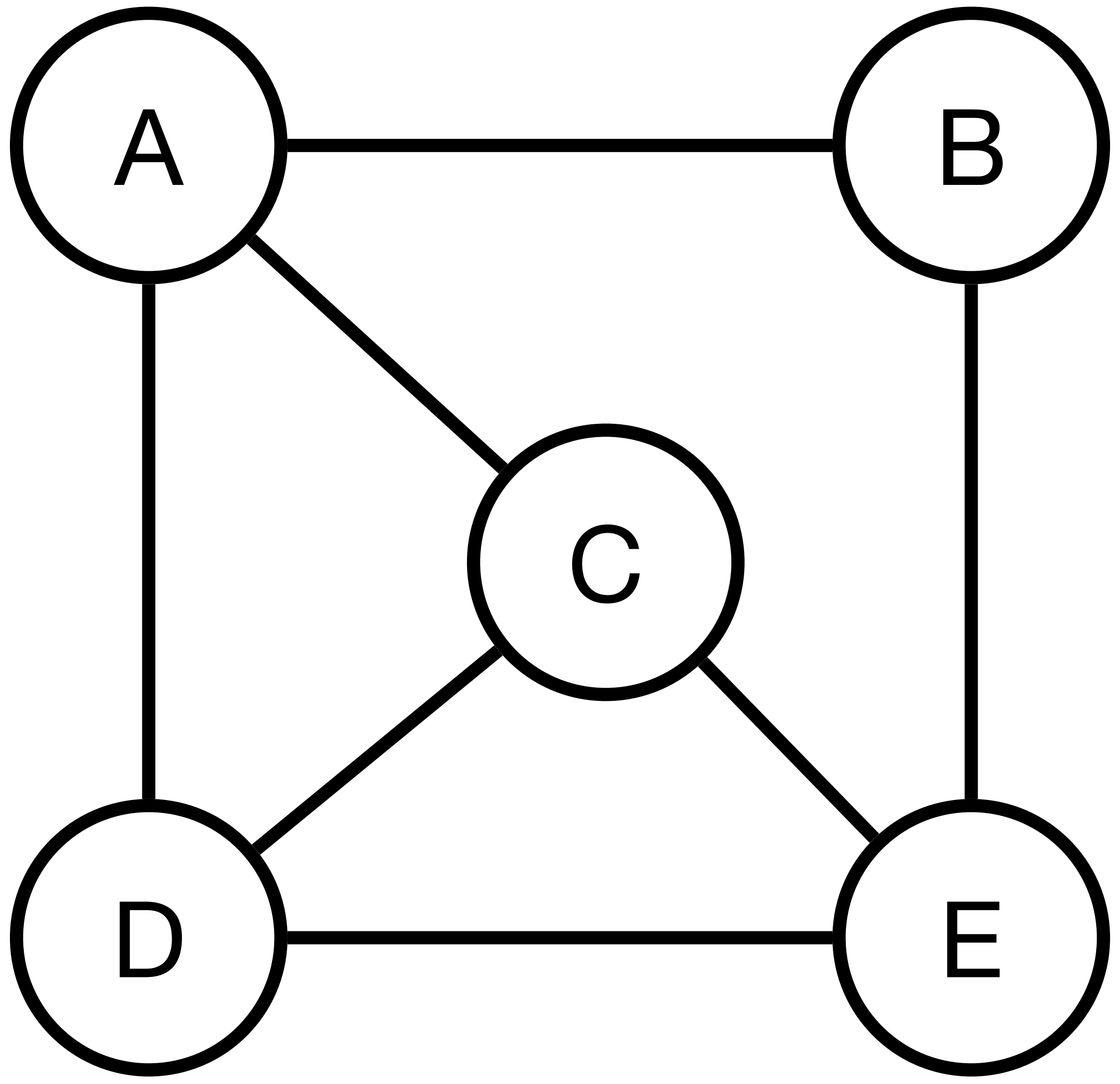

- **Subgraph** (of G)**:**
  - Subset of vertices and edges of G
- **Connected component:**
  - Every pair of vertices connected by a path
  - Component is a subgraph
  - Maximality: Cannot include any vertex adjacent to the vertices in the connected component and still have a connected graph.
  - Trees: Connected graphs w/o cycles.

# The Graph ADT

- Graph is a positional container which consists a (possibly mutable) set of vertices and edges
    - Edges are unordered pair of vertices for undirected graphs; ordered otherwise
    - Additionally, each edge may have a symbolic label or numeric value capturing cost, capacity, length or some other attribute
- Basic operations on Graph ADT:
    - size(), isEmpty(), numVertices(), numEdges(), vertices(), edges(), opposite(v,e), degree(v), inDegree(v), outDegree(v), adjacent(v), inAdjacent(v), outAdjacent(v), areAdjacent(v,w), endVertices(e), source(e), destination(e), isDirected(e)
    - Update operations: makeEdge(u,v), insertVertex(v), removeEdge(e), setDirection(e, v)

# Implementing Graph ADT: Data Structures

## Adjacency Matrix




| A | B | C | D | E |
|---|---|---|---|---|
| F | T | T | T | F |
| T | F | F | F | T |
| T | F | F | T | T |
| T | F | T | F | T |
| F | T | T | T | F |

- **Adjacency Matrix:** Matrix M with entries for each pair of vertices
- M[i][j] = T (edge between i and j)
- M[i][j] = F (**no** edge between i and j)
- Space: $\Theta(n^2)$

# Implementing Graph ADT: Data Structures

## Adjacency Matrix

- **Adjacency Matrix:**
  - If the graph is undirected then what we can say about the matrix?
    - Matrix will be **symmetric**
  - **Sparse Matrices:**
    - **Dictionary of Keys —** map (row-col) pairs to the non-zero value of $M_{i,j}$
  - **Sparse storage:**
    - $|V^2|/8$ bytes to store directed graphs?
    - For Undirected graphs?
      - Store lower triangle matrix in $|V^2|/16$ bytes

| A | B | C | D | E |
|---|---|---|---|---|
| F | T | T | T | F |
| T | F | F | F | T |
| T | F | F | T | T |
| T | F | T | F | T |
| F | T | T | T | F |

# Implementing Graph ADT: Data Structures

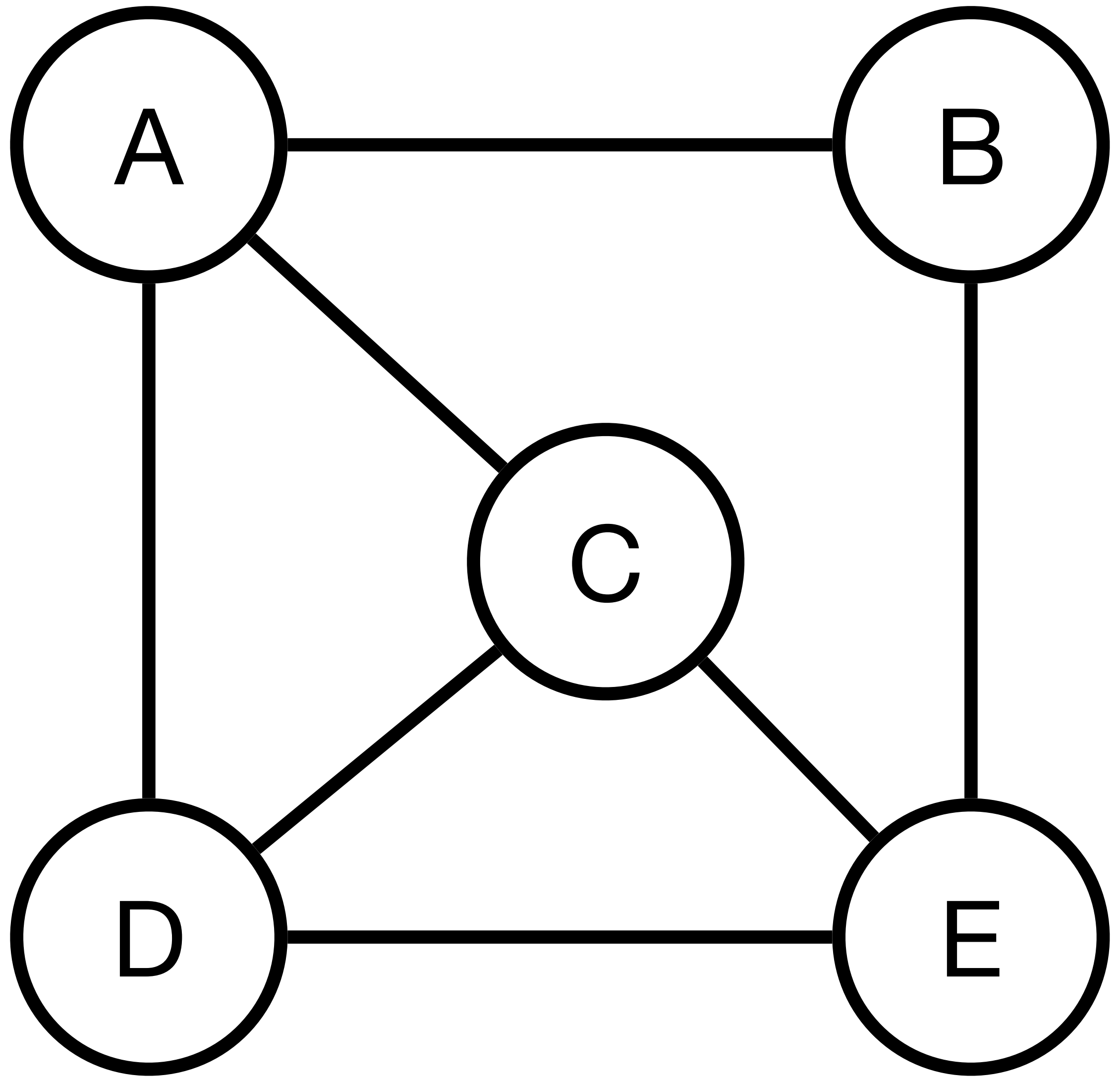
## Adjacency Matrix

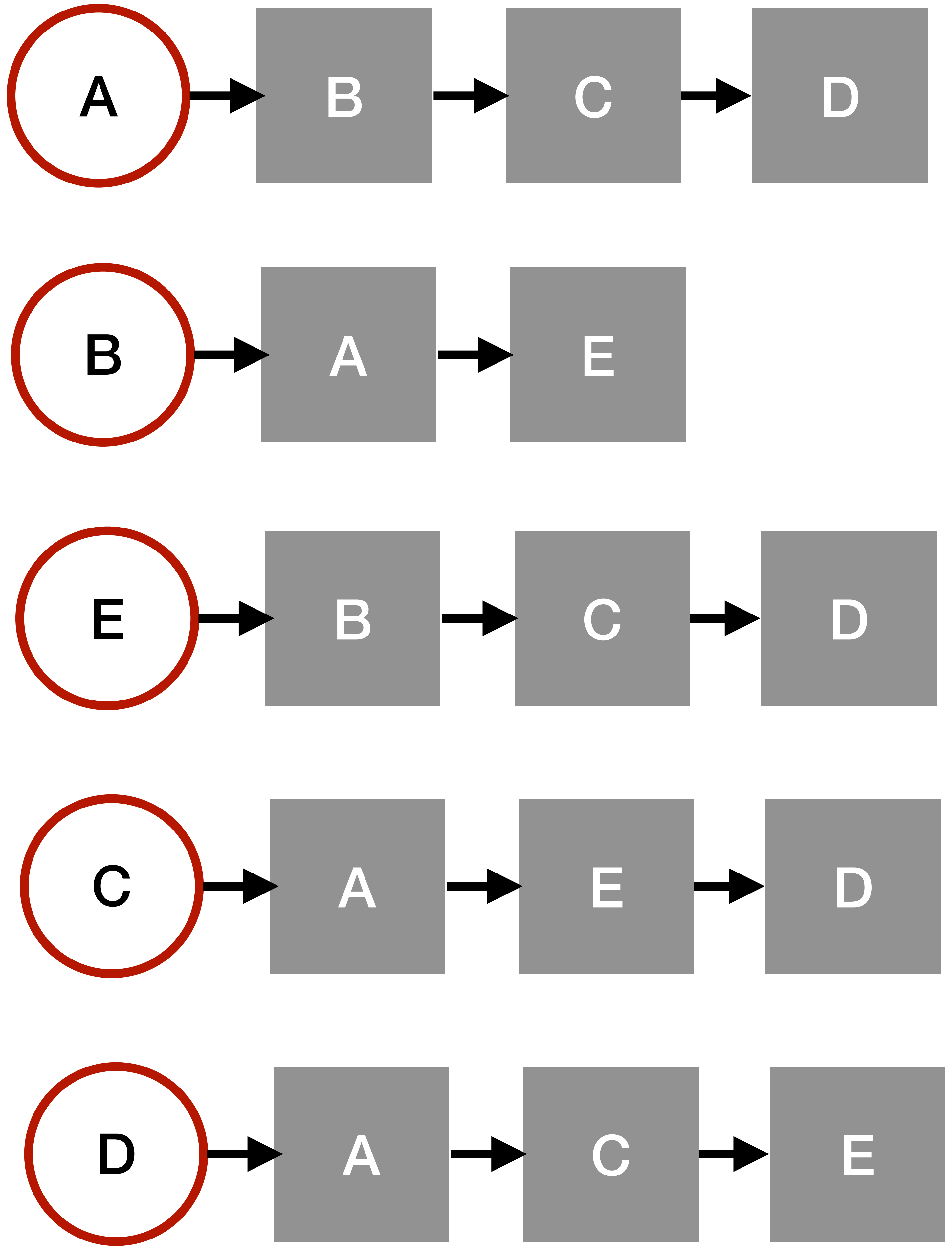
- **Complexity**
  - Size(), isEmpty, numVertices, numEdges, degree, inDegree, outDegree, source, destination, areAdjacent, insertions/removals of edges **— O(1)**
  - vertices — **O(n), edges — O(m)**
  - insertVertex, removeVertex — $O(n^2)$
  - incidentEdges, adjacentVertices — **O(n)**

# Implementing Graph ADT: Data Structures

## Adjacency List

- For each vertex **v** maintain a sequence of vertices **adjacent** to **v**
- Graph is now a collection of adjacency lists
- Worst-case space complexity: $\Theta(n + m)$

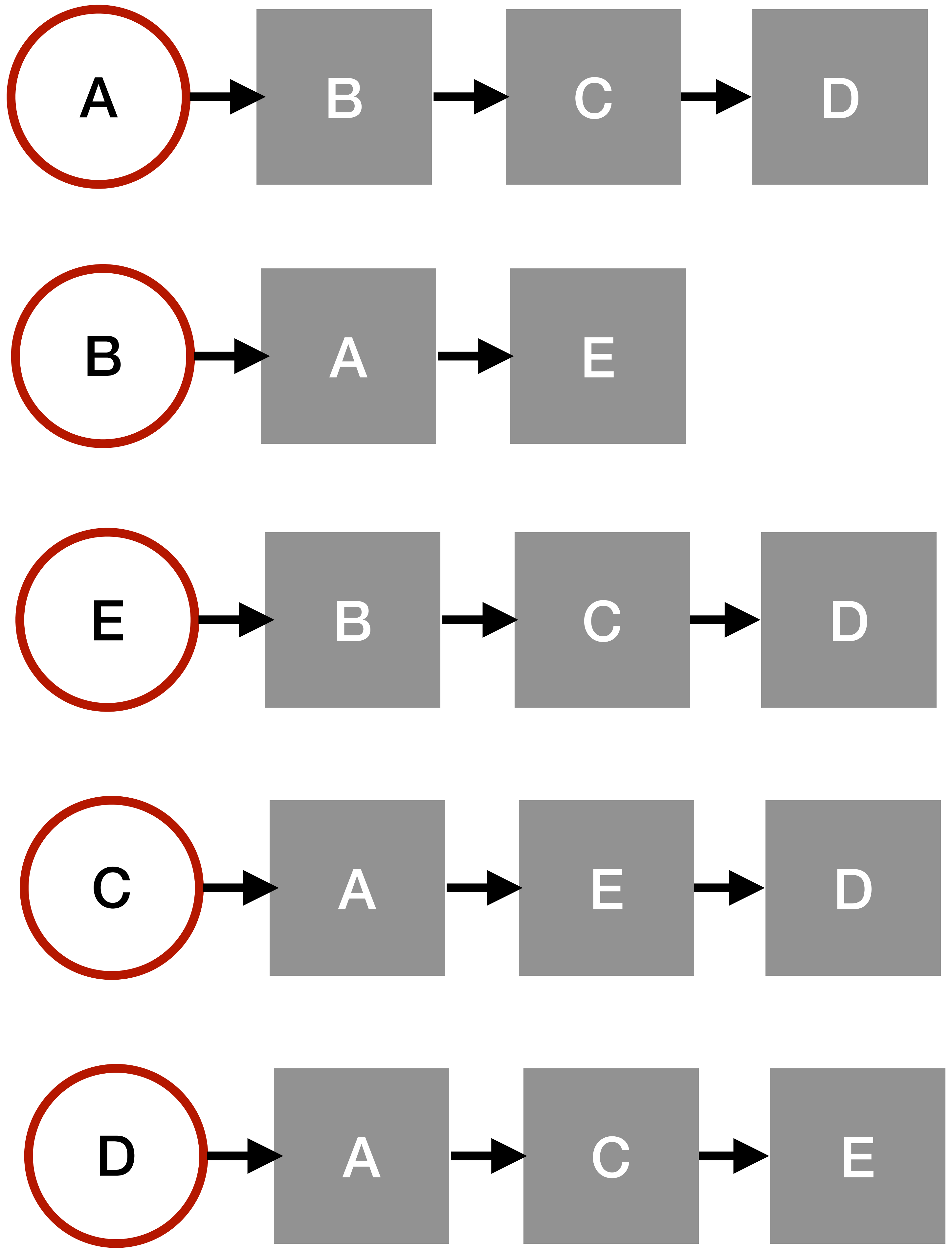


# Implementing Graph ADT: Data Structures

## Adjacency List

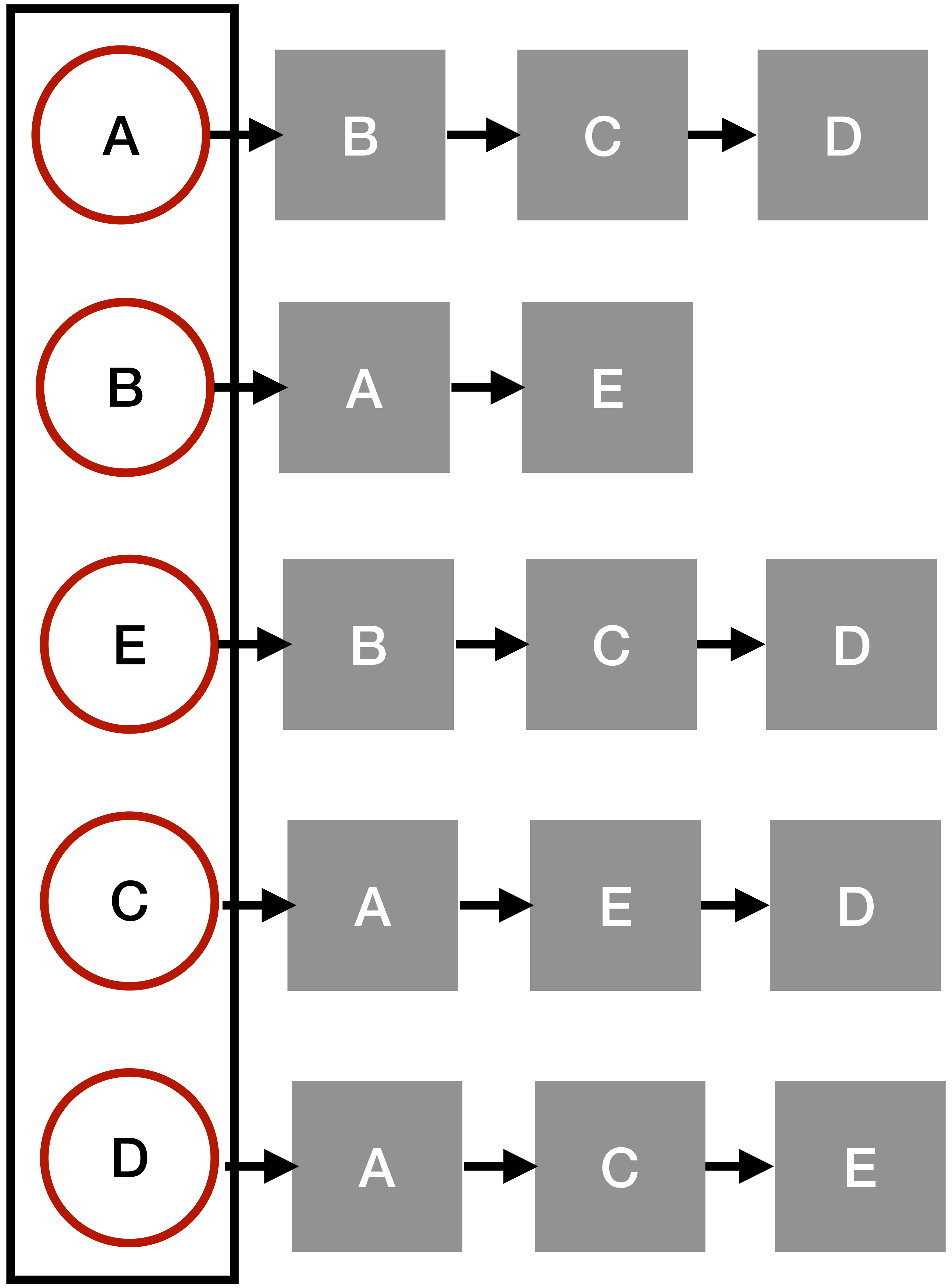
- For each vertex **v** maintain a sequence of vertices **adjacent** to **v**
- Graph is now a collection of adjacency lists
- Worst-case space complexity: $\Theta(n + m)$



# Implementing Graph ADT: Data Structures

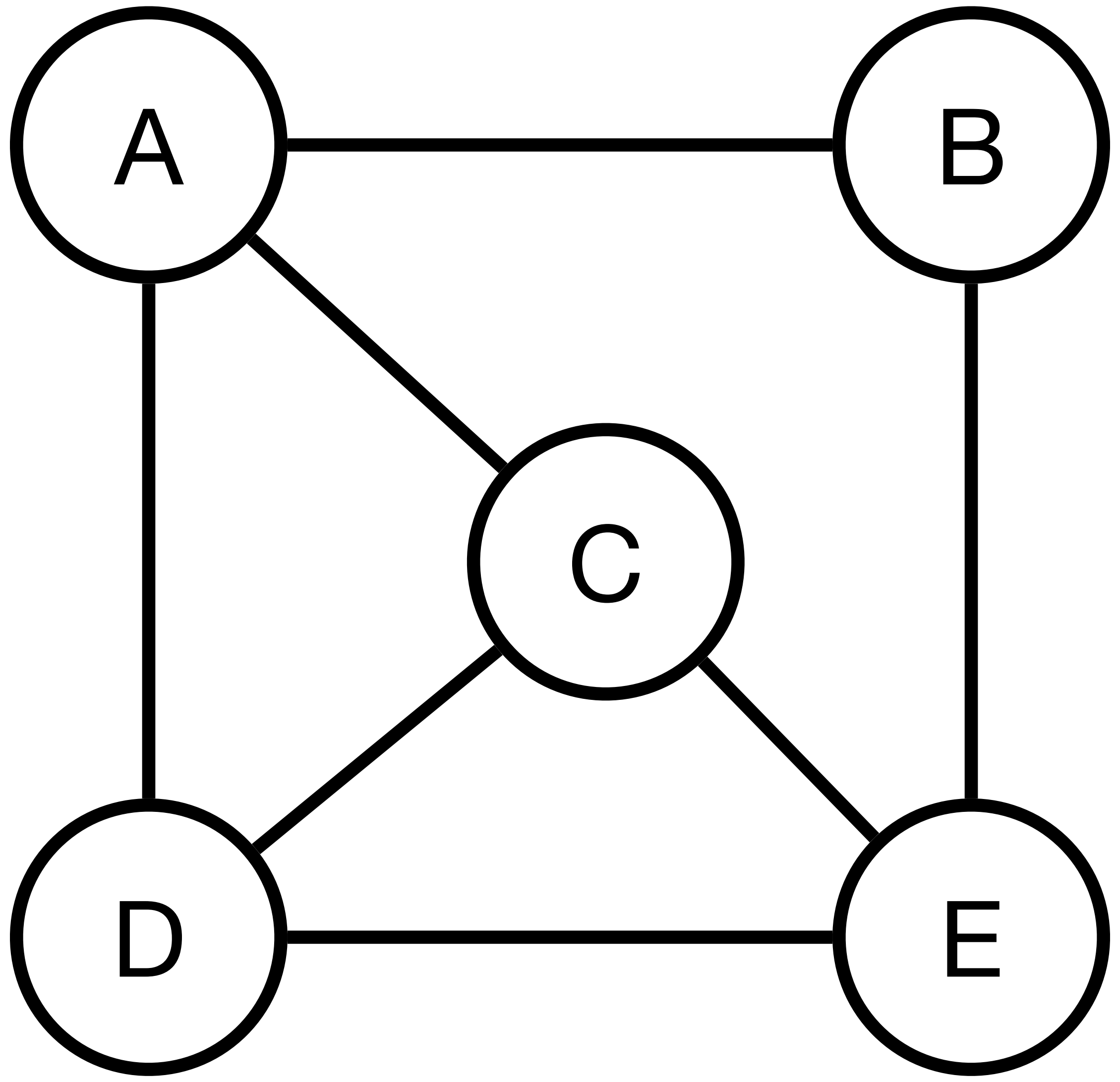
## Adjacency List: Variation of the basic idea

- Guido van Rossum (Python creator):
    - Use of **hashtable** to map each vertex to its adjacent vertices
    - No explicit representation of edges in the hash table
- Cormen et al.:
    - **Array of indices**, each vertex is mapped to an index of the array; each **array cell points to a singly linked list** of adjacent vertices

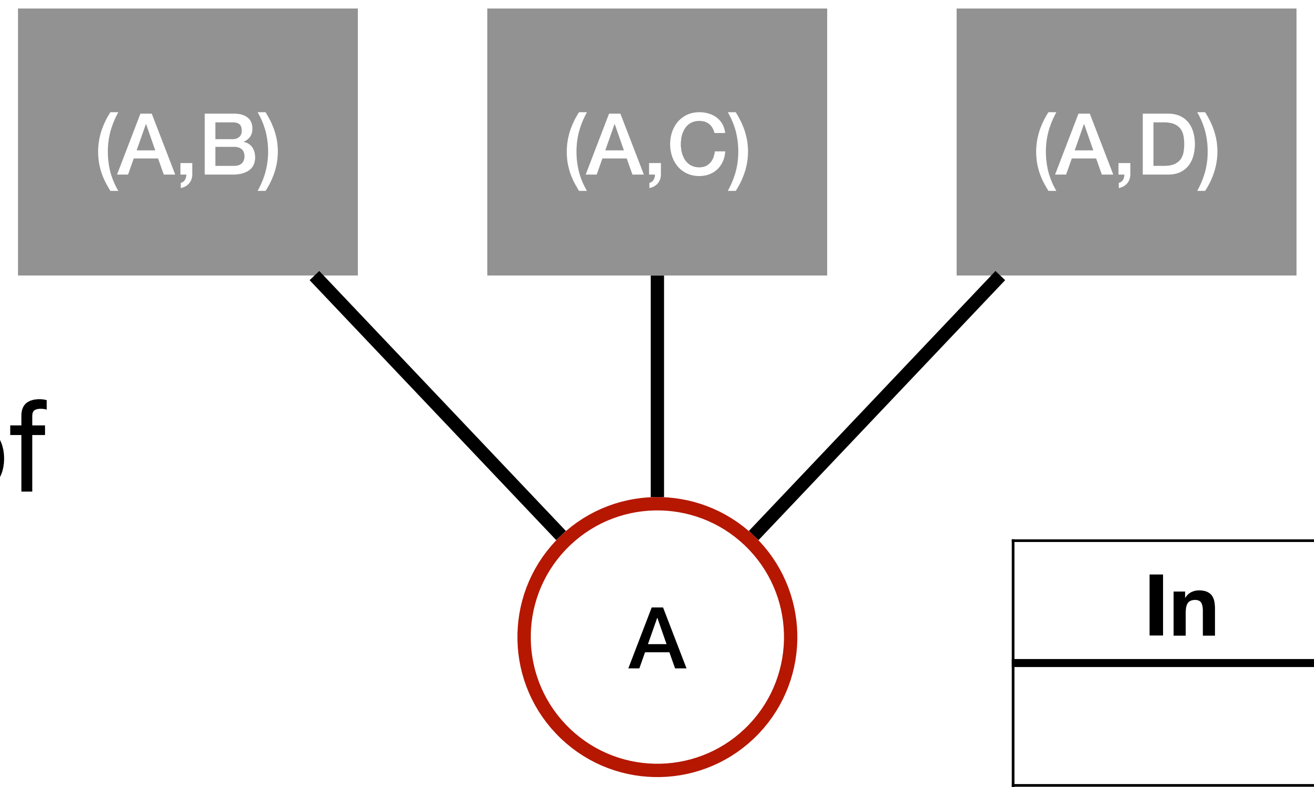


# Implementing Graph ADT: Data Structures

## Adjacency List: Variation of the basic idea




- Tamassia and Goodrich: maintains both vertex and edge objects.
  - Each vertex points a collection of edges to whom it connects
  - Each edge points to its two endpoints
  - More memory requirement: $O(n+m)$
    - Allows extra edge information to be stored
- **Ungraded assignment** - Time complexity analysis of all the methods we discussed before!




| In | Out |
|---|---|
| | |
| | |

# Graph Traversal
## DFS For Search

- **Depth-first Search:**
  - Idea is to visit child vertices before visiting sibling vertices of a vertex
  - Algorithm:
    - Start from chosen root vertex and iteratively visit the unvisited adjacent vertex until we cannot continue
    - Backtrack along previously visited vertices until unvisited vertices are found to be connected to them