CS 234: Data Types and Structures

Naomi Nishimura

Module 6

Date of this version: October 24, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Case study

Problems:

- Given flight information for all the airlines you trust, determine whether it is possible to fly from point A to point B.
- If it is possible, determine the smallest number of flights needed to get from point A to point B.
- Or, determine the cheapest way of getting from point A to point B.

Recipe for user/plan

1. Determine types of data and operations.
2. For each type, choose/modify/create an ADT.
3. Develop pseudocode algorithm using ADT operations.
4. Calculate cost of algorithm with respect to costs of operations.
5. Using information from provider, choose best option.

Comparing graphs and trees

Similarities:

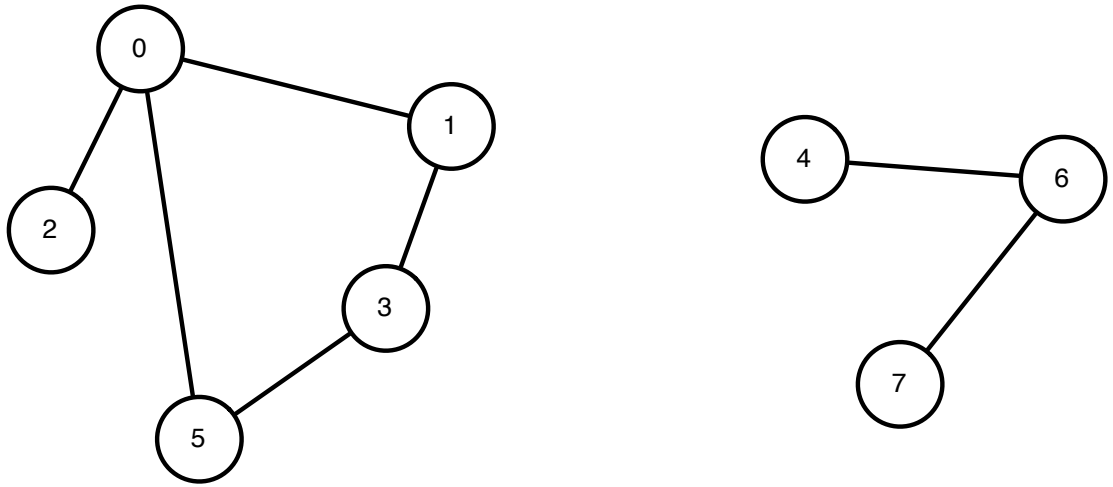
- Both store data items in positions with respect to other data items.
- Both relate two data items by an **edge** connecting them.

Differences:

- A rooted tree has a specified root from which all nodes can be reached.
- In a rooted tree, each edge connects a parent and a child.
- Any two nodes are connected by exactly one simple path.

Graph terminology

A **simple undirected graph** G is a set $V(G)$ of **vertices** and a set $E(G)$ of **edges**, that is, unordered pairs of vertices $\{u, v\}$ such that $u \neq v$.

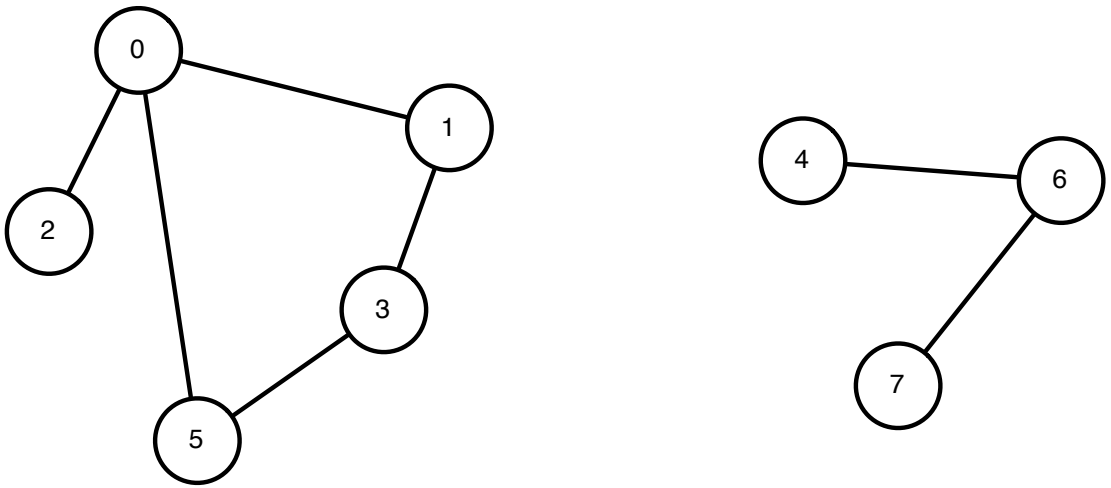


Useful terms:

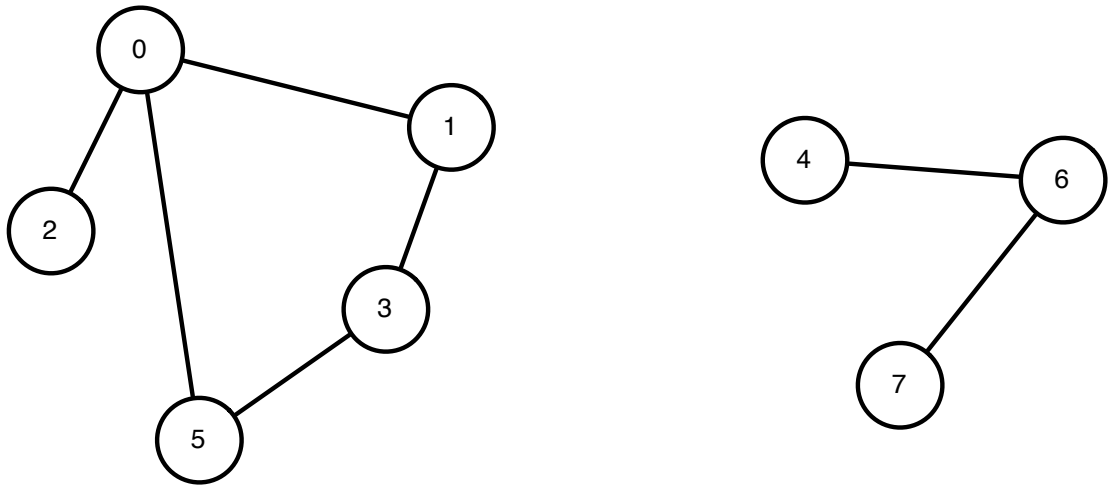
- Two vertices u and v are **adjacent** if $\{u, v\} \in E(G)$.
- The vertices u and v are the **endpoints** of the edge $\{u, v\}$.
- The edge $\{u, v\}$ is **incident on** the vertices u and v (and vice versa).
- Two edges are **incident** if they share an endpoint.

Degree and neighbours

- The set of all vertices adjacent to u is the set of **neighbours** of u, forming the **neighbourhood** of u.
- The **degree** of a vertex is the number of incident edges.
- A vertex of degree zero is an **isolated vertex**.



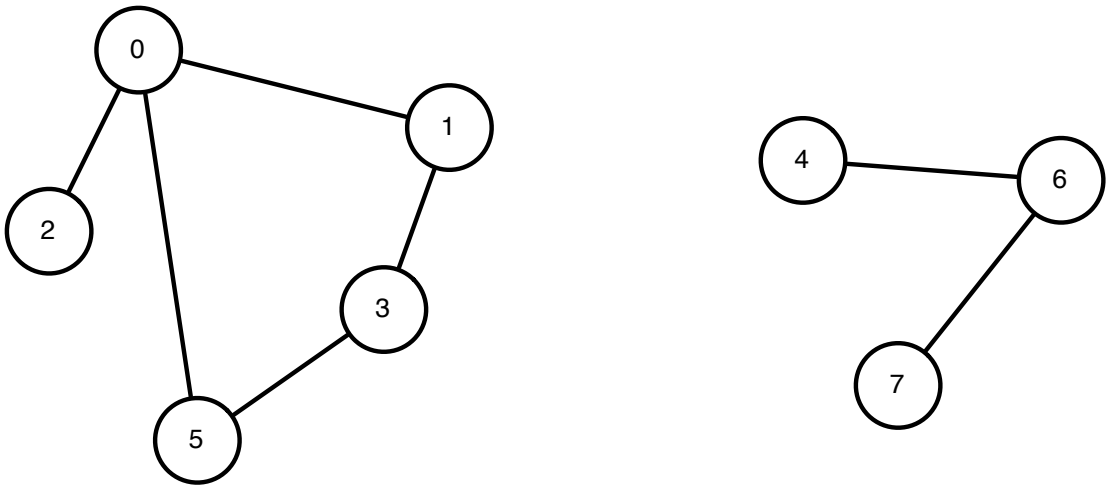
Paths and cycles



- A **path** between vertices v_0 and v_k is a sequence of vertices $\{v_0, \ldots, v_k\}$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $0 \leq i < k$. (In a **simple path**, no vertex is repeated.)
- A **cycle** is a sequence of vertices $\{v_0, \ldots, v_k\}$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $0 \leq i < k$ and $\{v_k, v_0\} \in E(G)$.
- A graph without a cycle is **acyclic**.

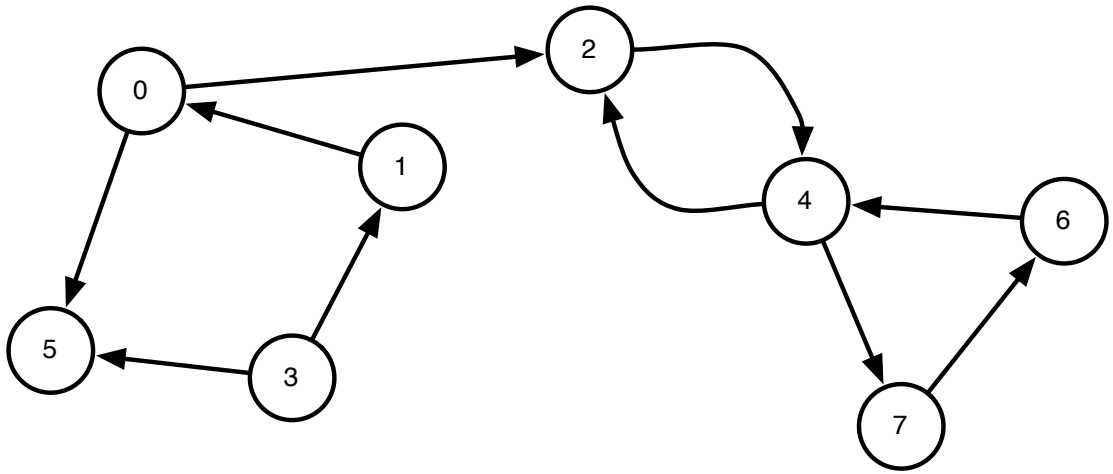
Connected graphs

A graph is **connected** if there is path between any pair of vertices in $V(G)$.



Directed graphs

In a **directed graph**, each edge (or **arc**) has a direction.

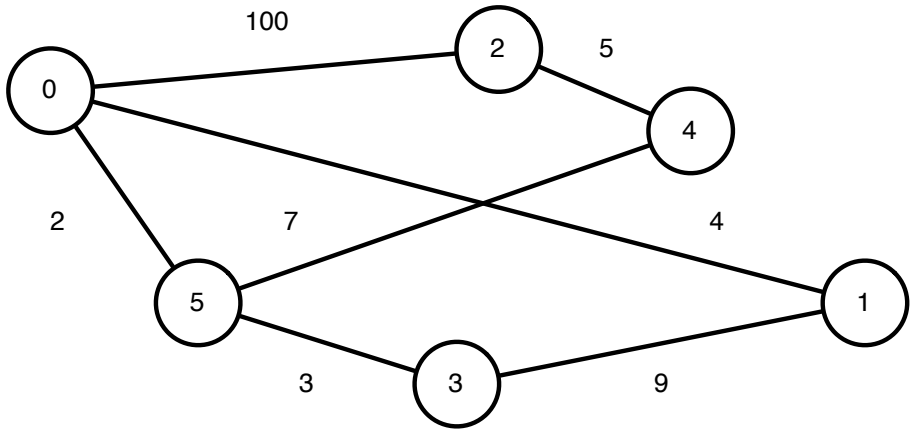


Observations:

- An arc is written as an ordered pair of endpoints (u, v), where u is the **tail** and v is the **head**; the direction is from u to v (as in $(3, 1)$).
- The arcs $(2, 4)$ and $(4, 2)$ are distinct.
- The vertex 0 has one **in-neighbour** (1) and two **out-neighbours** (2 and 5). It has **indegree** one and **outdegree** two.
- There is a **directed path** from vertex 3 to vertex 7.

Weights

There can be weights assigned to edges and/or vertices.



Observations:

- This graph has weights on edges.
- The weight of the path from 0 to 2 through 5 and 4 has a total weight of $2 + 7 + 5 = 14$, which is less than the weight of the edge from 0 to 2.

It is also possible for a directed graph to be weighted.

Choosing data for an ADT

Depending on the application, a vertex or an edge can store various types of data, such as:

- a value
- a weight
- a colour

As in trees, we can use a unique ID for each vertex.

Various options include:

- Use IDs for edges.
- Identify an edge with the IDs of its endpoints.
- Use integers in the range from 0 through $n - 1$ for n vertices.

We will typically use the second or third option.

Compound data

We can view each vertex or edge as a single data item, such as by defining an ADT Vertex and an ADT Edge.

Operations might include:

- Return the value.
- Return the weight.
- Return the colour.
- Set the value.
- Set the weight.
- Set the colour.

Alternatively, we can allow ADT Graph operations to access or set just one part (e.g. value, weight, or colour) of a data item.

Search operations for graphs

In an undirected graph:

- Find all vertices in the graph
- Find all edges in the graph
- Find all the neighbours of a vertex
- Determine if there is an edge between two vertices
- Find the value, weight, or colour of a vertex
- Find the value, weight, or colour of an edge

In a directed graph:

- Operations similar to those above (arcs, not edges)
- Find all the in-neighbours of a vertex
- Find all the out-neighbours of a vertex

Modification operations for a graph

In an undirected (directed) graph:

- Add a vertex
- Add an edge (arc)
- Delete a vertex
- Delete an edge (arc)
- Set the value, weight, or colour of a vertex
- Set the value, weight, or colour of an edge (arc)

ADT Undirected Graph, without modifications

Preconditions: For all *G* is a graph, *One* and *Two* are IDs of vertices in *G*; for *EDGE_VALUE* there exists an edge between vertices with IDs *One* and *Two*.

Name	Returns
CREATE()	a new empty graph
VERTICES(G)	IDs of all vertices (Group B ADT)
EDGES(G)	pairs of IDs for endpoints of edges (Group B ADT)
VERTEX_VALUE(G, One)	value of vertex with ID *One*
EDGE_VALUE(G, One, Two)	value of edge connecting vertices with IDs *One* and *Two*
NEIGHBOURS(G, One)	IDs of all neighbours of vertex with ID *One* (Group B ADT)
ARE_ADJACENT(G, One, Two)	*True* if there exists an edge connecting vertices with IDs *One* and *Two*, else *False*

ADT Undirected Graph, modifications

Preconditions: For all *G* is a graph, *One* and *Two* are IDs of vertices in *G* for all except *ADD_VERTEX*, and *Data* is a data item; the edge between vertices with IDs *One* and *Two* exists (resp., does not exist) for *SET_EDGE_VALUE* and *DELETE_EDGE* (resp., *ADD_EDGE*).

Postconditions: Mutation by *SET_VERTEX_VALUE* (changes vertex value), *SET_EDGE_VALUE* (changes edge value), *ADD_VERTEX* (adds vertex), *ADD_EDGE* (adds edge), *DELETE_VERTEX* (deletes vertex), *DELETE_EDGE* (deletes edge).

Name	Returns
SET_VERTEX_VALUE(G, One, Data)	
SET_EDGE_VALUE(G, One, Two, Data)	
ADD_VERTEX(G, One)	
ADD_EDGE(G, One, Two)	
DELETE_VERTEX(G, One)	
DELETE_EDGE(G, One, Two)	

Calculating running times for graph operations

Conventions used:

- Unless other information is known, running times are expressed as functions of n (the number of vertices) and m (the number of edges).
- If extra information is known, the function can be simplified.

For any graph, $m \in O(n^2)$.
For a connected graph $m \in \Omega(n)$.

Examples:

Function	General	Connected
$n + m^2$	no simpler	$\Theta(m^2)$
$n^2 + m$	$\Theta(n^2)$	$\Theta(n^2)$
$n \log n + m$	no simpler	no simpler

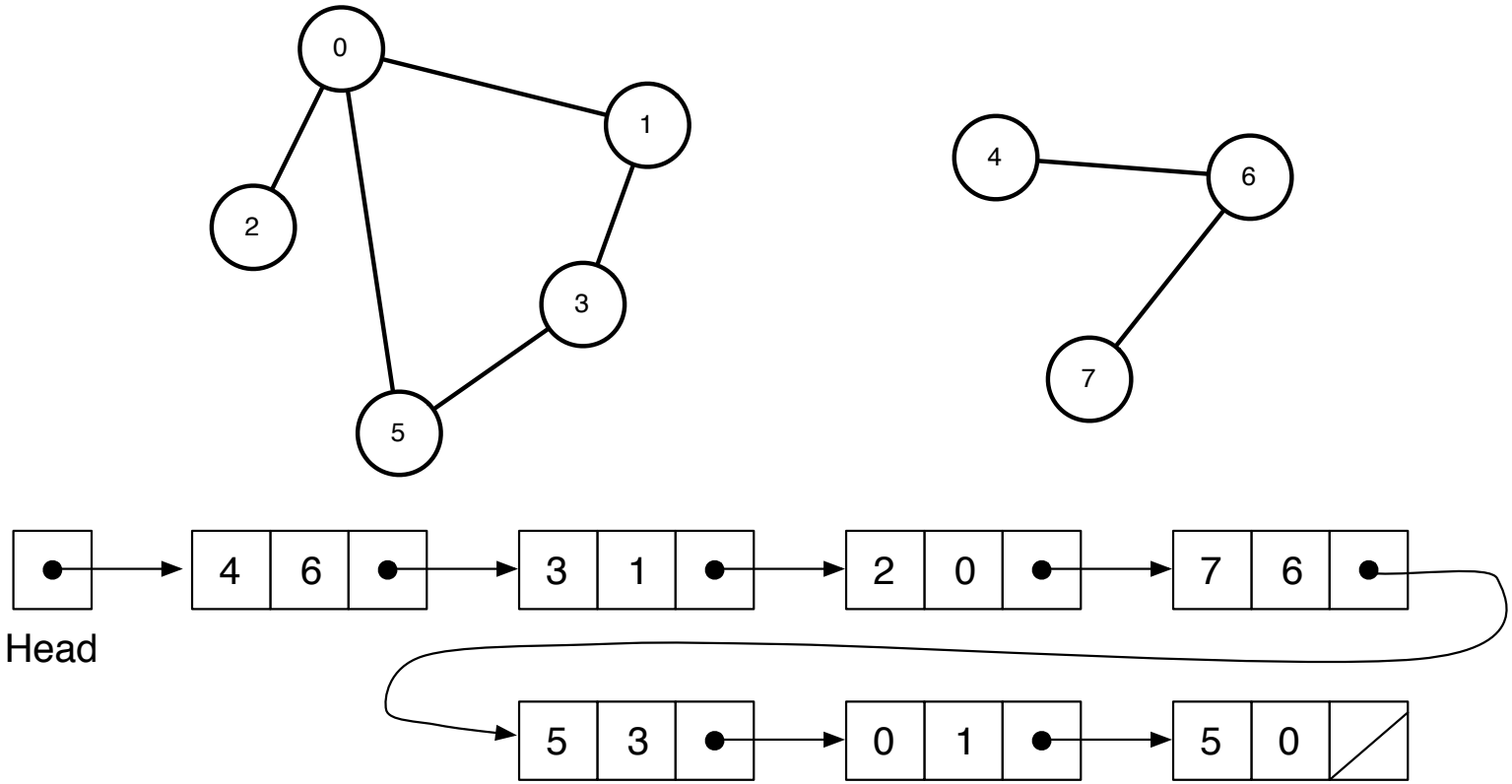
Special case: No isolated vertices or extra data

An isolated vertex has degree zero (and hence no incident edges and no neighbours).

Idea: If there are no isolated vertices, we can describe the graph by listing all the edges.

Since there is no extra information, each edge can be represented as the IDs of its endpoints.

Data structure: **Edge list**



Data structure: **Edge list**

Data structure:

- Variable *Head* storing a pointer to a linked list of nodes, where each node stores the endpoints of an edge and a pointer to the next node in the list

Algorithms for selected operations:

- *ADD_EDGE(G, One, Two)*: Add the new edge at the front of the list.
- *ARE_ADJACENT(G, One, Two)*: Search through the list in order to find whether the edge is present.
- *NEIGHBOURS(G, One)*: Search through the list in order to find all edges in which *One* is an endpoint and return all the other endpoints found.
- *ADD_VERTEX(G, One)*: This operation, followed by one or more *ADD_EDGE* operations using *One*, can be executed by adding new edges at the front of the list.

Analyzing and extending edge list

Worst-case running time of operations in terms of n and m:

- *ADD_EDGE(G, One, Two)*: $\Theta(1)$
- *ARE_ADJACENT(G, One, Two)*: $\Theta(m)$
- *NEIGHBOURS(G, One)*: $\Theta(m)$
- *ADD_VERTEX(G, One)*: $\Theta(1)$

Extensions:

- Arcs can be handled by distinguishing between the head and the tail.
- Edge weights can be handled by using a third value in each linked node.

Special case: Integer IDs

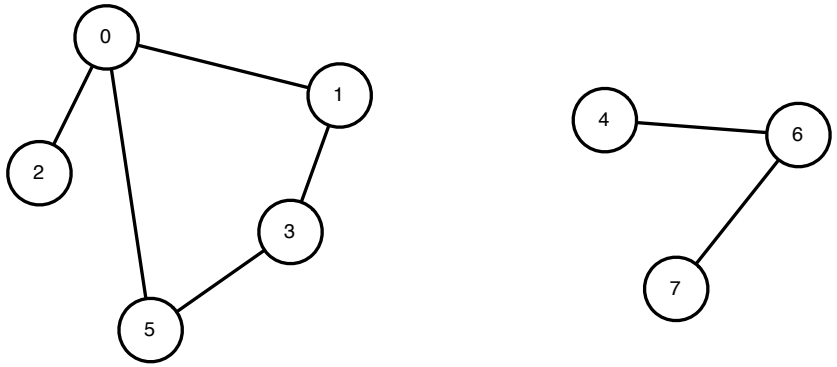
Idea: Each integer ID can be used as a position in contiguous memory.

Each edge has two endpoints, and hence two indices; information for each edge can be stored in an ADT Grid.

Notes:

- In the illustration, the only information is whether or not an edge exists. Each entry is a 0 (no edge) or a 1 (edge).
- Integer IDs can be used to store extra vertex information in an array indexed by vertex ID.

Data structure: **Adjacency matrix**



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	1	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	1	0	0
4	0	0	0	0	0	0	1	0
5	1	0	0	1	0	0	0	0
6	0	0	0	0	1	0	0	1
7	0	0	0	0	0	0	1	0

Data structure: **Adjacency matrix**

Data structure:

- ADT Grid with information about the edge with endpoints *One* and *Two* at positions (*One*, *Two*) and (*Two*, *One*)

Algorithms for selected operations:

- *ADD_EDGE(G, One, Two)*: Update the entries at positions (*One*, *Two*) and (*Two*, *One*).
- *ARE_ADJACENT(G, One, Two)*: Check the entry at position (*One*, *Two*).
- *NEIGHBOURS(G, One)*: Check the entries at positions (*One*, *i*) for all values of *i*.
- *ADD_VERTEX(G, One)*: Create a new, larger ADT to accommodate the new vertex.

Analyzing and extending adjacency matrix

Assumption: ADT Grid implemented using contiguous implementation

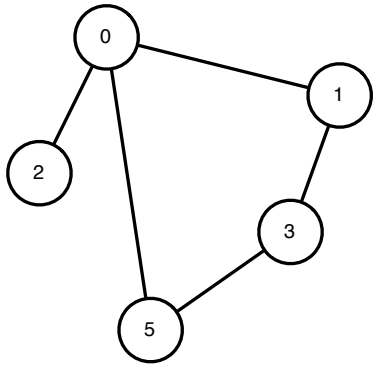
Worst-case running time of operations in terms of n and m:

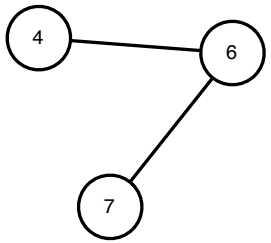
- *ADD_EDGE(G, One, Two)*: $\Theta(1)$
- *ARE_ADJACENT(G, One, Two)*: $\Theta(1)$
- *NEIGHBOURS(G, One)*: $\Theta(n)$
- *ADD_VERTEX(G, One)*: $\Theta(n^2)$

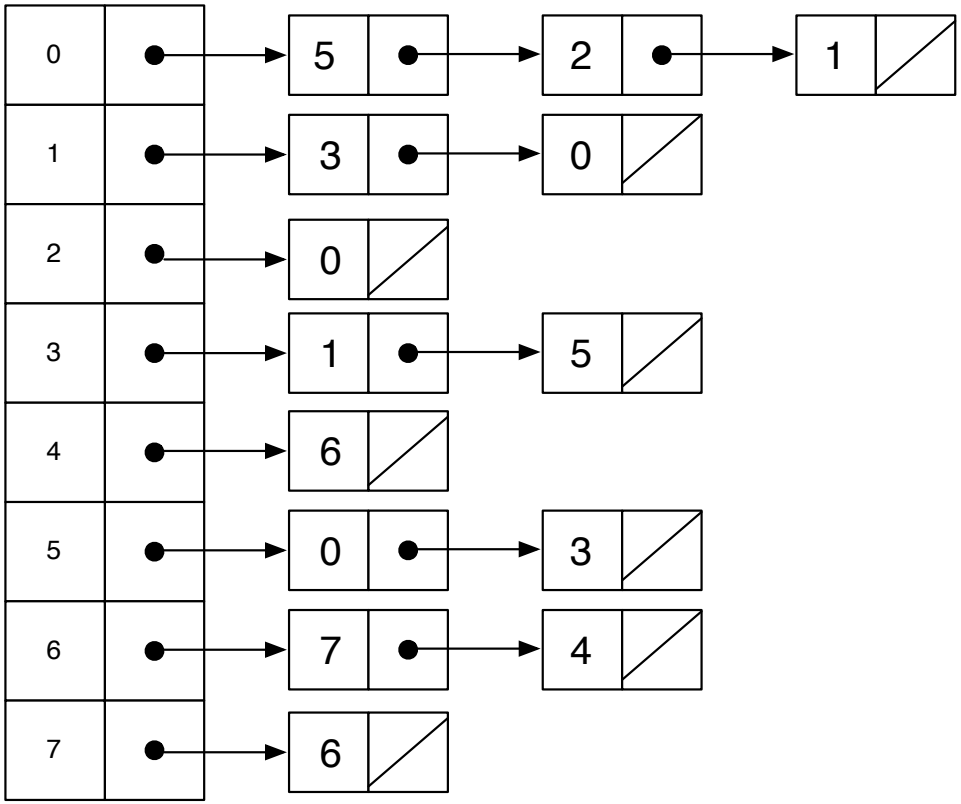
Extensions:

- Arcs can be handled by having the first position be the head and the second be the tail.
- Matrices (with 0 and 1 values) can be multiplied repeatedly to determine existence of paths between pairs of vertices.

Data structure: **Adjacency list**







Data structure: **Adjacency list**

Data structure:

- Array of size *n* with pointers to linked lists, where the linked list for vertex *One* has a node for each neighbour of *One*

Algorithms for selected operations:

- *ADD_EDGE(G, One, Two)*: Add the edge at the fronts of the linked lists for *One* and *Two*.
- *ARE_ADJACENT(G, One, Two)*: Search through the list in order to find whether the edge is present in the linked list for either *One* or *Two*.
- *NEIGHBOURS(G, One)*: Search through the list in order to find neighbours in the linked list for *One*.
- *ADD_VERTEX(G, One)*: Create a new, larger array to accommodate the new vertex and copy over all the pointers.

Analyzing and extending adjacency list

Worst-case running time of operations in terms of n, m, d_o (the degree of *One*) and d_t (the degree of *Two*):

- *ADD_EDGE(G, One, Two)*: $\Theta(1)$
- *ARE_ADJACENT(G, One, Two)*: Depending on the algorithm, either $\Theta(d_o)$ or $\Theta(d_t)$
- *NEIGHBOURS(G, One)*: $\Theta(d_o)$
- *ADD_VERTEX(G, One)*: $\Theta(n)$

Note: Each degree is in $O(\min\{m, n\})$.

Extensions:

- Arcs can be handled by storing separate lists of in-neighbours and out-neighbours.
- Edge weights can be handled by using a third value in each linked node.

Comparison of data structures for graphs

All of the data structures can be adapted for use in general graphs.

Each has its own strengths:

- Edge list: Fastest *ADD_VERTEX*
- Adjacency matrix: Fastest *ARE_ADJACENT*
- Adjacency list: Fastest *NEIGHBOURS*

Which one to select depends on how frequently each of the operations will be used.

Further extensions

Possible additional requirements:

- Vertices with arbitrary IDs
- Vertices storing extra information
- Edges storing extra information
- Multiple edges between a pair of vertices

Possible approaches:

- Use an ADT Dictionary (Module 7) to map ID names to 0 through $n-1$
- Use an array where slot i contains ADT Vertex for vertex i (also can be used for isolated vertices in an edge list)
- Use an ADT Edge for an edge (e.g. stored in a slot in an adjacency matrix)

Case study, revisited

Given flight information for all the airlines you trust, determine whether it is possible to fly from point A to point B, **possibly stopping along the way at intermediate airports**.

Special cases:

- Use *ARE_ADJACENT* if no stops are required.
- Multiply adjacency matrices repeatedly and check for 1's if the data structure can be accessed directedly.

If we find neighbours of neighbours repeatedly, we will eventually find all possible destinations *B* that can be reached by *A*.

New problem: Given *A*, what is the set of vertices connected by a path to *A*?

Colouring vertices

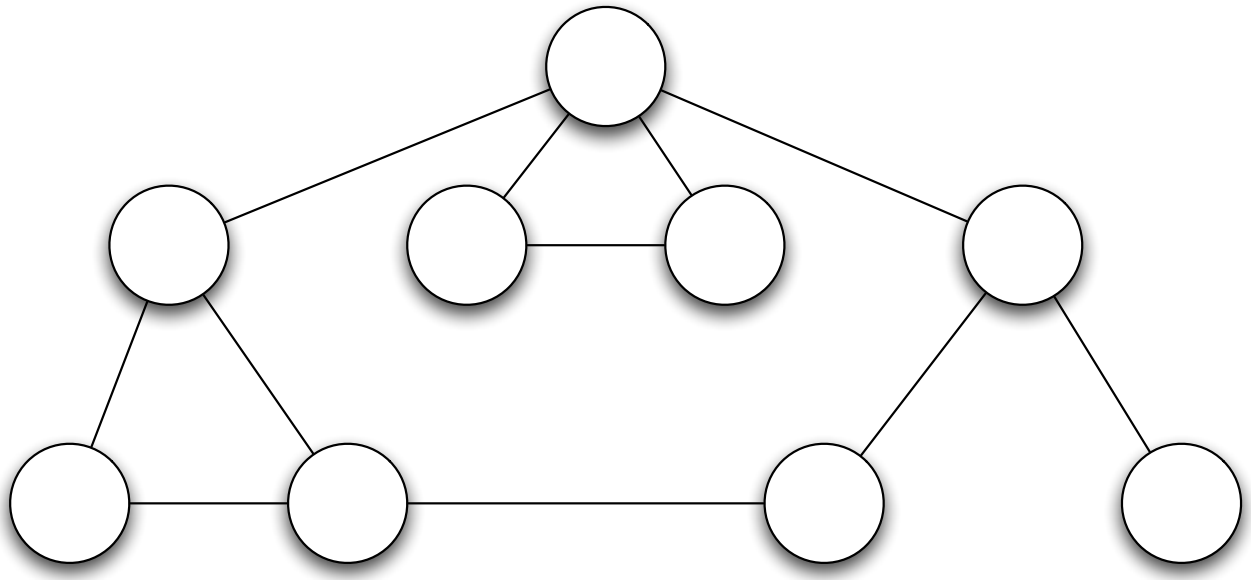
Colours:

- unvisited - white
- visited but not yet completely processed - gray
- completely processed - black, where completely processed includes all neighbours having been checked

Augmenting the ADT:

- *VERTEX_COLOUR(G, One)*
- *SET_VERTEX_COLOUR(G, One, New)*

Breadth-first search illustrated



Breadth-first search pseudocode template

```
BFS(G, Start)
INPUT:      A graph G and a vertex Start in G; all vertices are white
EFFECTS:    Visits all vertices reachable from Start; all such vertices are black
1   Q ← CREATE_QUEUE()
2   ENQUEUE(Q, Start)
3   SET_VERTEX_COLOUR(Q, Start, gray)
4   Optional steps placed here
5   while not IS_EMPTY_QUEUE(Q)
6       Current ← DEQUEUE(Q)
7       for Nbr in NEIGHBOURS(G, Current)
8           if VERTEX_COLOUR(G, Nbr) == white
9               ENQUEUE(Q, Nbr)
10              SET_VERTEX_COLOUR(G, Nbr, gray)
11      SET_VERTEX_COLOUR(G, Current, black)
12      Optional steps placed here
```

Complexity analysis of BFS

Each reachable vertex is enqueued and dequeued, total $O(n)$.

When processed:

- All neighbours are checked.
- All other operations are constant time.

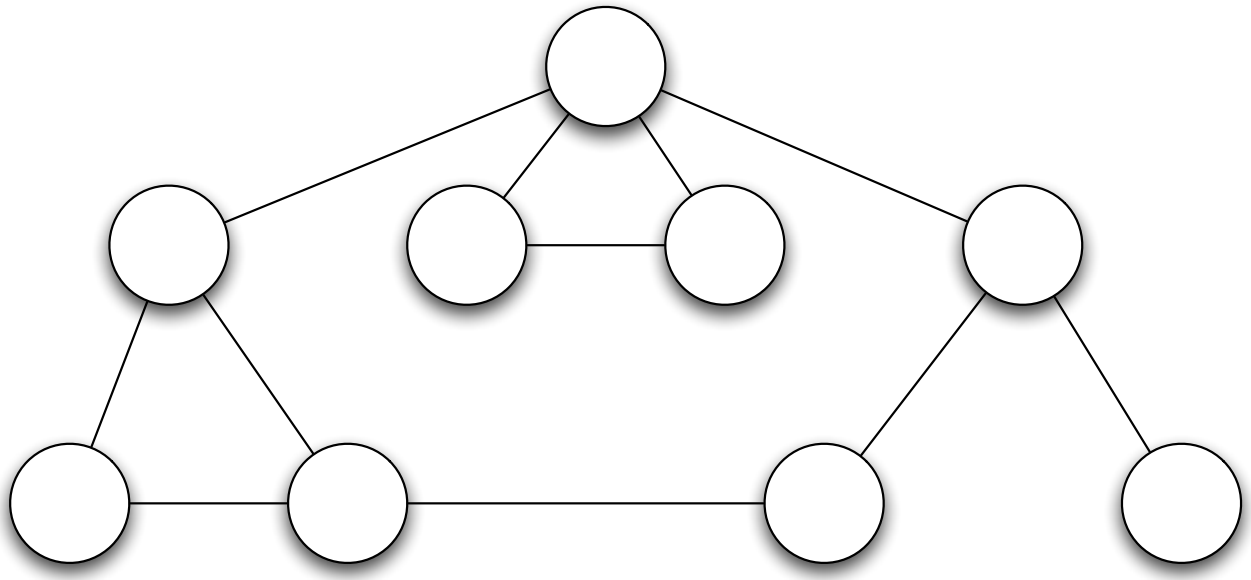
Total cost: sum over all vertices of cost of *NEIGHBOURS*

Edge list: $O(nm)$

Adjacency matrix: $O(n^2)$ (for each of n vertices, $O(n)$ to check all neighbours)

Adjacency list: $O(n+m)$ (cost of accessing array and each linked node; sum of degrees is twice number of edges)

Depth-first search illustrated



Depth-first search pseudocode template

```
DFS(G, Start)
INPUT:      A graph G and a vertex Start in G; all vertices are white
EFFECTS:    Visits all vertices reachable from Start; all such vertices are black
1   SET_VERTEX_COLOUR(G, Start, gray)
2   Optional steps placed here
3   for each Nbr in NEIGHBOURS(G, Start)
4       if VERTEX_COLOUR(G, Nbr) == white
5           DFS(G, Nbr)
6   SET_VERTEX_COLOUR(G, Start, black)
7   Optional steps placed here
```

Case study, not completed

Determine the smallest number of flights needed to get from point A to point B, or determine the cheapest way of getting from point A to point B.

This problem is studied in CS 231, using techniques outside the scope of this course.

Module summary

Topics covered:

- Case study: Flights
- Graphs
- Directed graphs
- Compound data
- ADT Undirected Graph
- Special case: No isolated vertices or extra data
- Data structure: Edge list
- Special case: Integer IDs
- Data structure: Adjacency matrix
- Data structure: Adjacency list
- Breadth-first search
- Depth-first search