**GRAPH TRAVERSAL**

Queues

* First In First Out (remember stacks = L I F O)

Characteristic Operations:

* **Enqueue(Queue q, Item it)** add item onto queue
* **Item it = dequeue(Queue q)** remove item from stack

Other possible operations:

* **isEmpty(Queue q)** queue contains no items
* **Length(Queue q)** how many items in queue
* **Show(Queue q)** display queue on stdout
* **Queue q = newQueue()** create new empty queue
* **dropQueue(Queue q)** release queue data

Queue implementations

* As a linked list w/ array data
  + Pointer to first element of the list
  + Pointer to last element of the list
  + Dequeue = take off FIRST element
  + Enqueue = add to LAST element

**PRIORITY QUEUES**

Instead of FIFO, it is “one with the highest priority goes out” regardless of what goes in first.

* Items are processed in order of “key” / importance

Priority Queues (PQs) provide this via:

* **void join(PQ, Item)**: insert item in PQ
* **Item leave(PQ)**: remove item with the highest priority
* **Item remove(PQ, Key)**: remove item with specified key (non-standard)

Plus generic ADT operations:

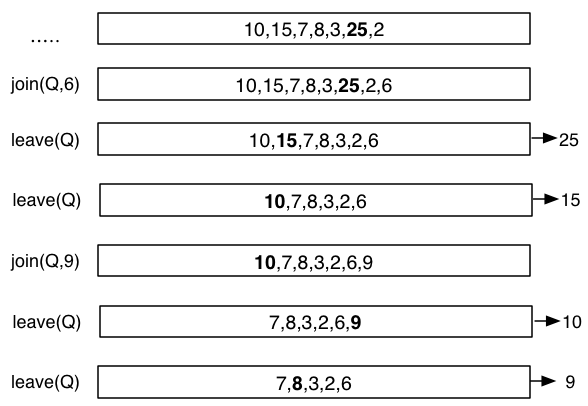
* **newPQ** / **dropPQ** / **isEmpty**…

Highest priority could be:

* largest key
* smallest key
* E.g. taking out item with the highest or lowest timestamp

Priority order may involve “weight” based on other factors than just a key

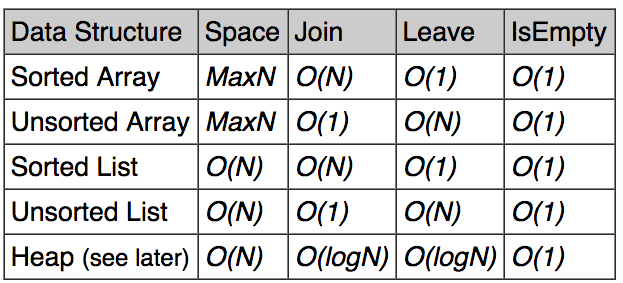
Behaviour of priority queue



E.g. Highest priority = largest no.

* 6 joins queue
* Highest (25) leaves queue
* Highest (15) leaves queue
* 9 joins queue
* Highest (10) leaves queue
* Highest (9) leaves queue

Priority Queue Cost Representations:



For a **PQueue** containing N items.

* Space = number of items in the data structure

HEAP is one of the better queue representations

UNSORTED is simple to join  
SORTED is simple to leave

**RECURSION**

**Add notes here**

**LISTS**

**Add notes here**

**GRAPH FUNDAMENTALS**

Many applications require:

* A collection of **items** (a set)
* **Relationships** / connections between the items

Examples:

* Maps: items are cities, connections are roads
* Web: items are pages, connections are hyperlinks
* Facebook: items are people, connections are “friends”

Collection types we’ve seen so far

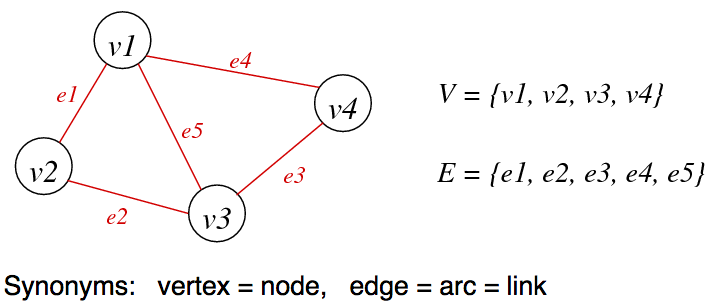
* Sets: unordered collection of items
* Lists: linear sequence of items (list, stack, queue)

**Graphs are more GENERAL, they show arbitrary connections (more random)**

A **graph G = (V, E)**

* V is set of vertices
* E is a set of edges (subset of VxV)

Example:



Edges can sometimes have info associated with them like:

* Weight / length / distance (maps) etc. / travel time
* A direction

Questions we might ask about a graph:

* Is there a way to go from A 🡪 B
  + E.g. Is it possible to drive from Sydney to Melbourne
* What is the best way to get from A 🡪 B
  + E.g. look at lowest weights / lengths / distances for edges in a map
* Which vertices are connected?
  + E.g. Is graph of roads from Sydney connected to graph of roads from New Zealand?

Graph algorithms are generally more complex than tree/list ones:

* No implicit order of items
  + There is no first vertex / item
* Graphs may contain cycles
  + For trees / lists you eventually reach the end
  + For graphs you can go around in a cycle
* Concrete representation is less obvious
  + Trees / list representations are more simple.
  + Graphs are harder to picture and can look like anything.
* Algorithm complexity depends on connection complexity
  + How connected vertices are between each other can increase complexity.

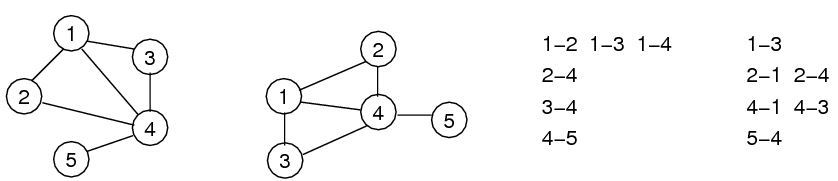
Properties of graphs:

* Terminology: **|V|** and **|E|** normally written as V and E
* A graph with V vertices has at most **V(V-1) / 2 edges**
* The ratio V:E can vary considerably (ratio of no. vertices to no. edges)
  + **E is closer to V2/2** = graph is DENSE
  + **E is closer to V** = graph is SPARSE
* Knowing whether a graph is sparse or dense is important
  + May affect choice of data structures to represent graph
  + May affect choice of algorithms to process graph

Describing graphs

* V needs to be identified (1 … V)
* E needs to be drawn or numbered

Four representations of the same graph:

x

**GRAPH TERMINOLOGY**

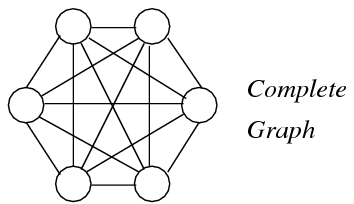
Graph Terminology

* For an **edge e**, that connects **vertices v and w**:
  + V and W are **adjacent**
  + E is **incident** on both V and W
* **Degree** of a vertex V = number of edges incident on V
* **Path** = sequence of vertices, each vertex has an edge to its predecessor
* **Cycle** = a type of PATH where last vertex in path is the same as first vertex in path

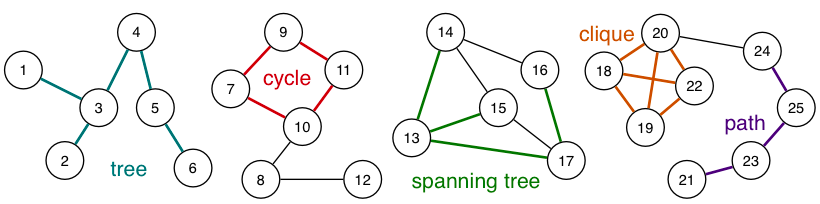


Path: 1-2 , 2-3 , 3-4 (connection stops at end)  
Cycle: 1-2 , 2-3 , 3-4 , 4-1 (connects back to the beginning)

Types of graphs

* **Connected Graph**
  + There is a PATH from each vertex to every other vertex
  + If a graph is not connected, it has connected components
  + E.g. Sydney and Melbourne are connected within, but not  
    connected to each other
* **Complete Graph**
  + There is an EDGE from each vertex to every other vertex
  + In a complete graph, **E = V(V-1) / 2**
* **Tree** = Connected subgraph with no cycles
* **Spanning Tree** = Tree containing all vertices
* **Clique** = Complete subgraph

Consider the following SINGLE GRAPH:



This graph has 25 vertices, 28 edges and 4 connected components

Other types of graphs:

* **Directed Graph (di-graph)** = Each edge has an associated direction (e.g. hyperlinks)
* **Weighted Graph** = Each edge has an associated value (weight) (e.g. Aus roads)
* **Multi-Graph** = Multiple edges between two vertices (e.g. function call graph 🡪 **f( ) calls g( ) in several places**)

**GRAPH ADT**

Data:

* Set of edges + Set of vertices

Operations

* Building: create graph, create edge, add edge
* Deleting: remove edge, drop whole graph
* Scanning: get edges, copy, show

Notes:

* Set of vertices is fixed when graph is initialised
* We treat vertices as **INTS** but they could be **Items**

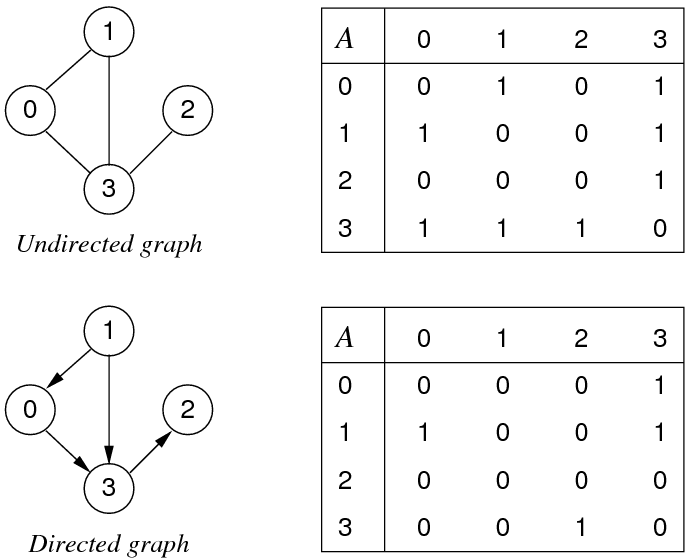
With edges, an edge from vertex 2 🡪 3 is the same as an edge from vertex 3 🡪 2.

We can fix this by using **Canonical Edge Representation**

* We build vertexes in a way such that FIRST VERTEX is always smaller than SECOND VERTEX
* This is the preferred way when building graph
* See code for more details

**ADJACENY MATRIX REPRESENTATION**

Edges represented by a VxV matrix



**Undirected Graph**

* Always going to have a symmetric matrix

**Directed Graph**

* Matrix won’t necessarily be symmetric

In C, a matrix is basically an **array of arrays**

Implementation of **GraphRep**

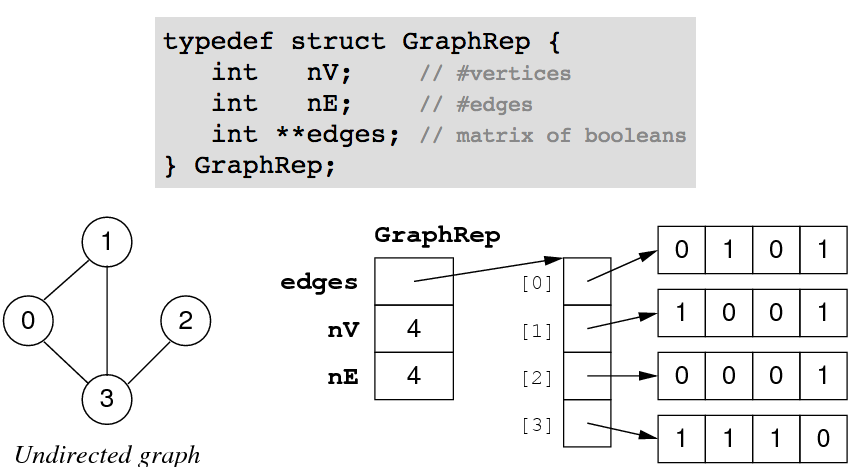
**int nV** = # vertices

**int nE** = # edges

**int \*\* edges** = ptr to an **int ptr**.

**int ptr** = ptr to the start of an array of integers

We can’t do int \*\*edges in the form of int edges[X][Y] unless we know for sure the number of vertices in the graph. Otherwise for unknown, we need pointers. (dynamic allocation with ptrs)



Storage cost: **V int ptrs + V2 ints**

Storage optimisation: store only top-right part of matrix

Cost of operations:

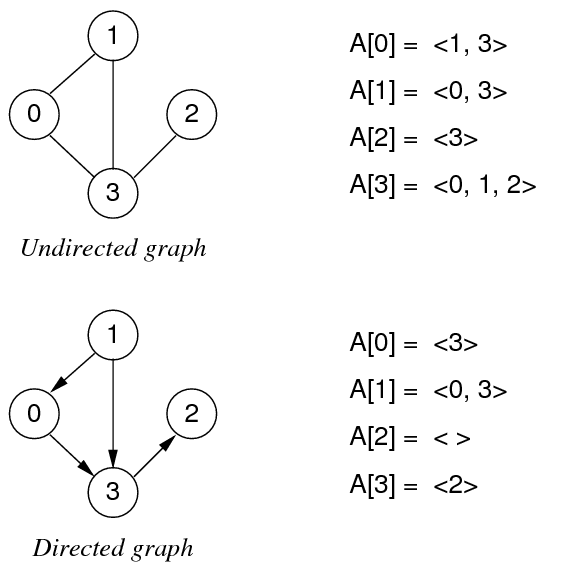
* **Initialisation: O(V2)** (initialise VxV matrix)
* **Insert edge: O(1)** (set two cells in matrix)
* **Delete edge: O(1)** (unset two cells in matrix)

**ADJACENCY LIST REPRESENTATION**

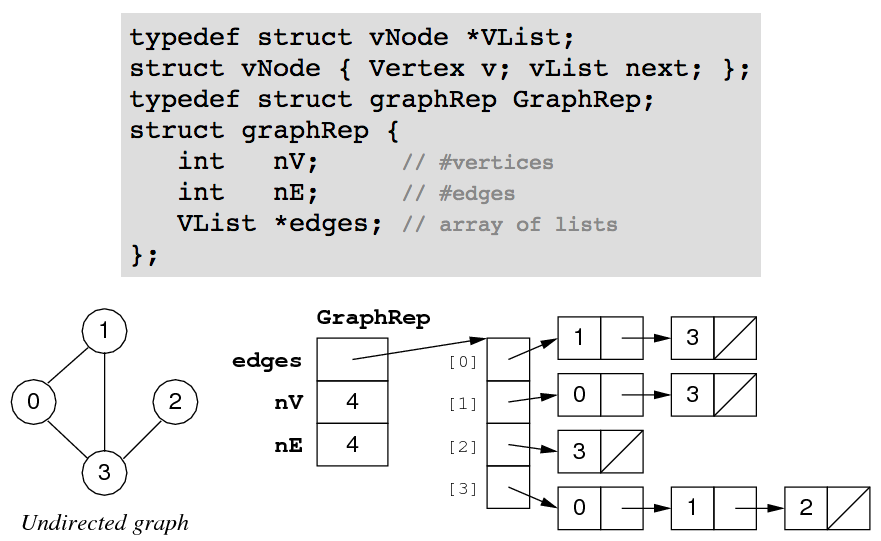
For each vertex, store linked list of adjacent vertices:

An array of elements:

* Each index represents a vertex  
  a[0] = vertex 0 a[1] = vertex 1 etc.
* Each index contains a list of vertices that are connected (linked lists)



Implementation of **GraphRep**



An array of elements:

* Each index represents a vertex  
  a[0] = vertex 0 a[1] = vertex 1 etc.
* Each index contains a list of vertices that are connected (linked lists)

**COMPARISON OF GRAPH REPRESENTATIONS**

