Algorithms and their Applications CS2004 (2020-2021)

Dr Mahir Arzoky

5.1 Data Structures and their Applications

NOTICES

Class Test CR I
will be released today!

Previously On CS2004…

- So far we have looked at:
  - Concepts of Computation and Algorithms
  - Comparing algorithms
  - Some mathematical foundation
  - The Big-Oh notation
  - Computational Complexity

Data Structures and their Applications

- Within this lecture we will discuss:
    - Lists
    - Stacks
    - Queues
    - Hash Tables
    - Graphs
    - Trees
    - And their applications.....
    - Some of the material should be familiar....

Why Study Data Structures?

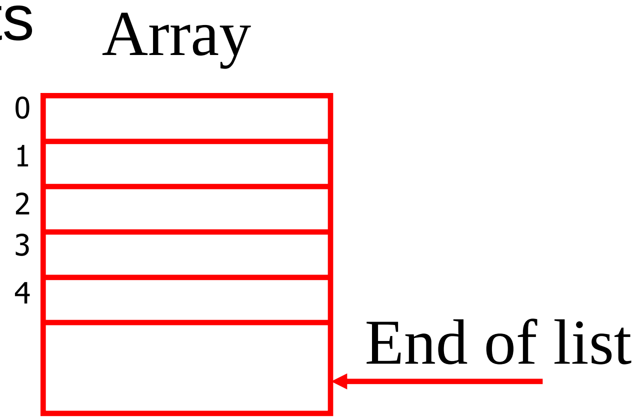
- Data structures are the "foundation stone" of all algorithms
  - Do not build on bad foundations…
- Representing the problem you are solving correctly will vastly help in designing a solution
- The use of the correct data structure will speed up an algorithm
- E.g. Sorting a list of 1,000 names
  - You would use a String array rather than 1,000 String variables!

Lists

- A **list** is a sequence of zero or more data items
- The total number of items is said to be the **length** of the list
- The length of a given list can grow and shrink on demand
- Items can be accessed, inserted, or deleted at any position in a list
- Let’s look at two ways to implement lists: Array and LinkedList

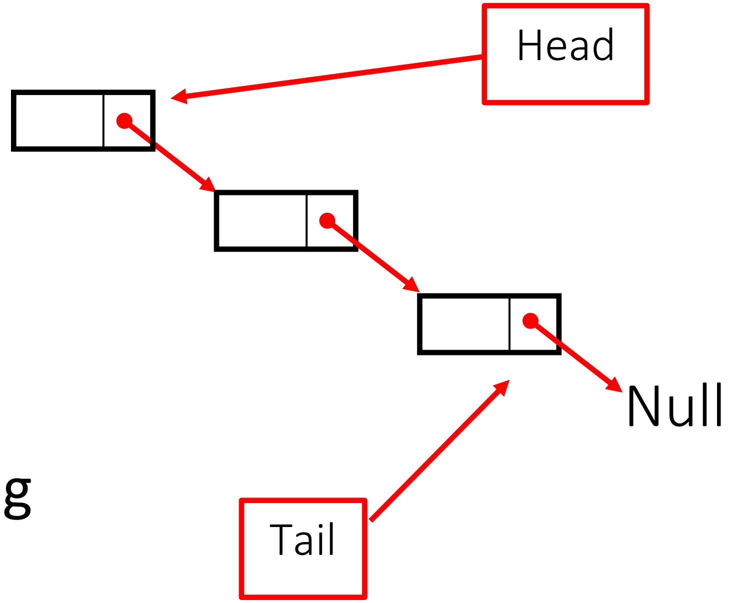
Array Implementation

- Arrays are the simplest and most widely used data structures
- Maintain insertion order of elements
- Elements are indexed…
  - Big(O) for searching for an index value: O(1)
- You are already familiar with `Arrays` and `ArrayLists` in Java
- An `Array` is a structure of fixed-size that can hold a collection of similar items
- `ArrayList` is a variable length Collection class. They can increase or decrease the size dynamically



LinkedList Implementation

- Implementation of the List interface
- Maintain the insertion order of elements
- Elements are not indexed – when searching, start with the head and work your way through…
- What is the complexity of finding a value?
  - O(n)
- Insertion / deletion can be done in O(1) (best) or O(n) (worst)



Stacks

- A stack is a special kind of list in which all insertions and deletions take place at one end
  - This is called the top
  - It has another name: “pushdown list‘”
- Its items are added and deleted on a last-in-first-out (**LIFO**) basis

Using Stacks

- To add an item to a stack, you **push** an item onto the top
- To remove an item from the top you **pop** it
- In the example below the top of the stack is on the right hand side

Empty	**Start**
20	Push 20
20 32	Push 32
20	Pop (returns 32)
20 21	Push 21
20	Pop (returns 21)

Queues

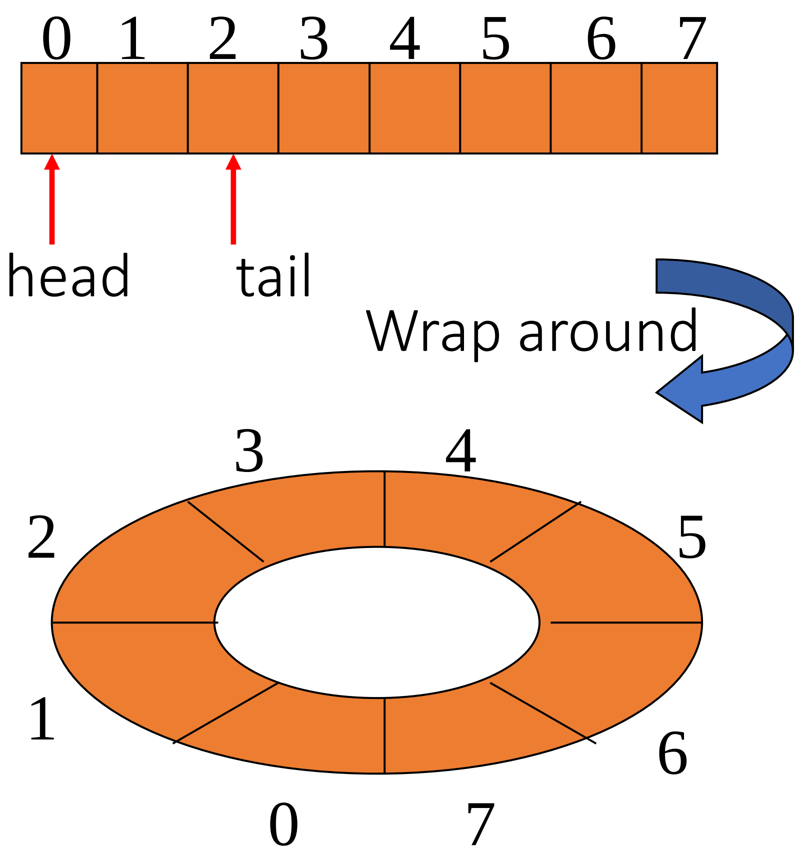
- A queue is another special kind of list
  - Items inserted at one end (the rear)
  - Items deleted at the other end (the front)
- A queue is a **FIFO** type data structure
  - The items are deleted in the same order as they were added
  - On a first-in-first-out basis
- For a queue structure, we have two special names for insertion and deletion:
  - ENQUEUE (insertion)
  - DEQUEUE (deletion)

Using Queues

Empty	Start
A	ENQUEUE A
A B	ENQUEUE B
A B C	ENQUEUE C
B C	DEQUEUE (A)
C	DEQUEUE (B)
C D	ENQUEUE D
D	DEQUEUE (C)

Return Value

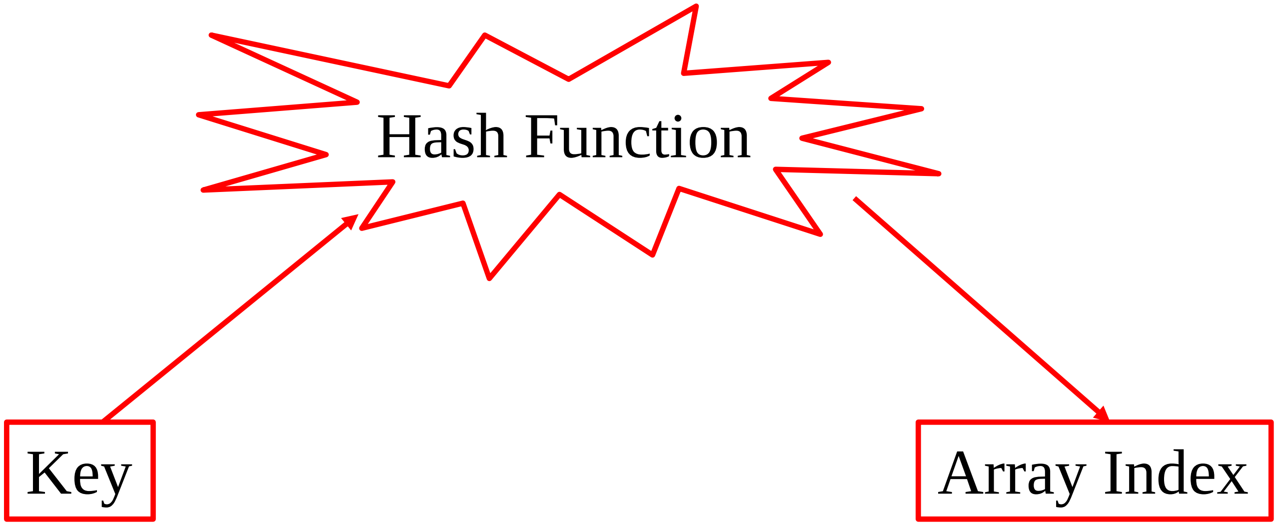
How to Implement Queues?



- We need to keep track of two indices, front and rear
- Enqueue (item) at rear and dequeue (item) at front
- Can not simply increment front/rear indices – as front may reach end of array!
- Solution: increase front/rear in circular manner

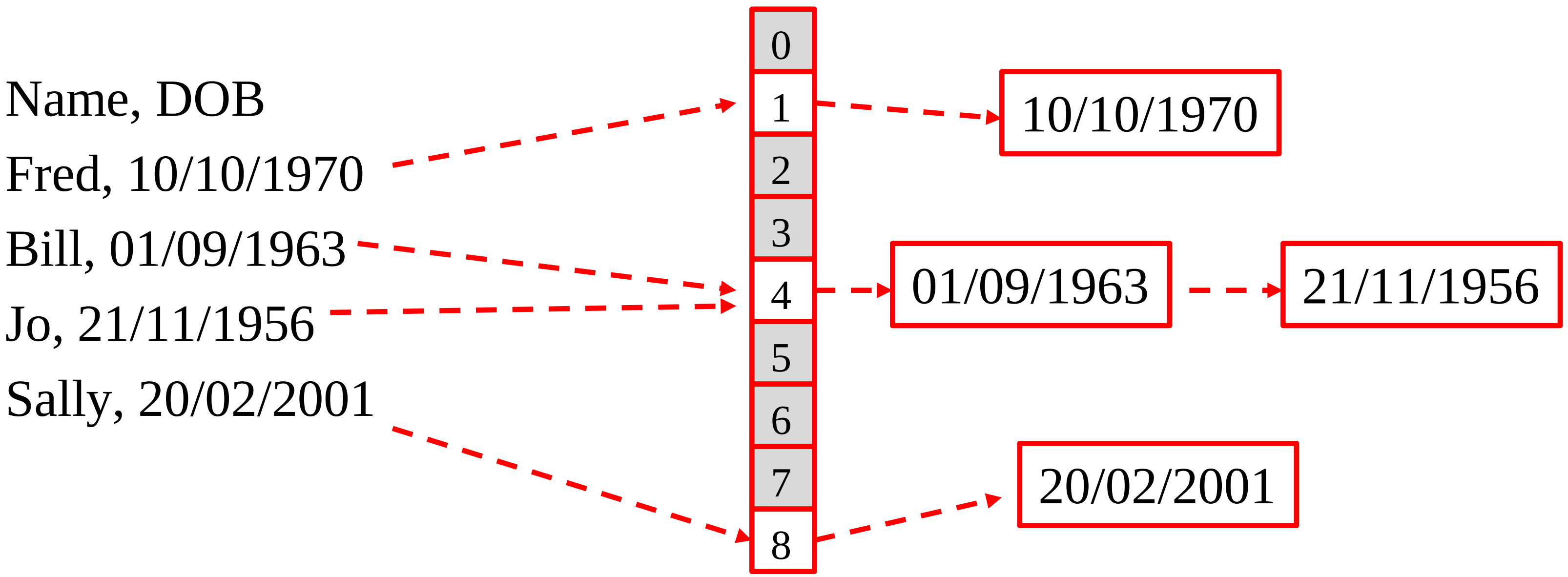
Hash Tables

- A Hash table or Hash Map is a data structure that maps a **Key** to a **Value**
- A special function called a **Hash Function** performs this mapping
- This function usually maps the key to an index in an array
- Key and value could be any type of data structure!



How To Implement Hash Tables

- An initial set of space is allocated in the array
- The hash function maps the key to an array index
- The function should map within the array bounds
- **Collision**: Two things with different hash codes could be mapped to the same index



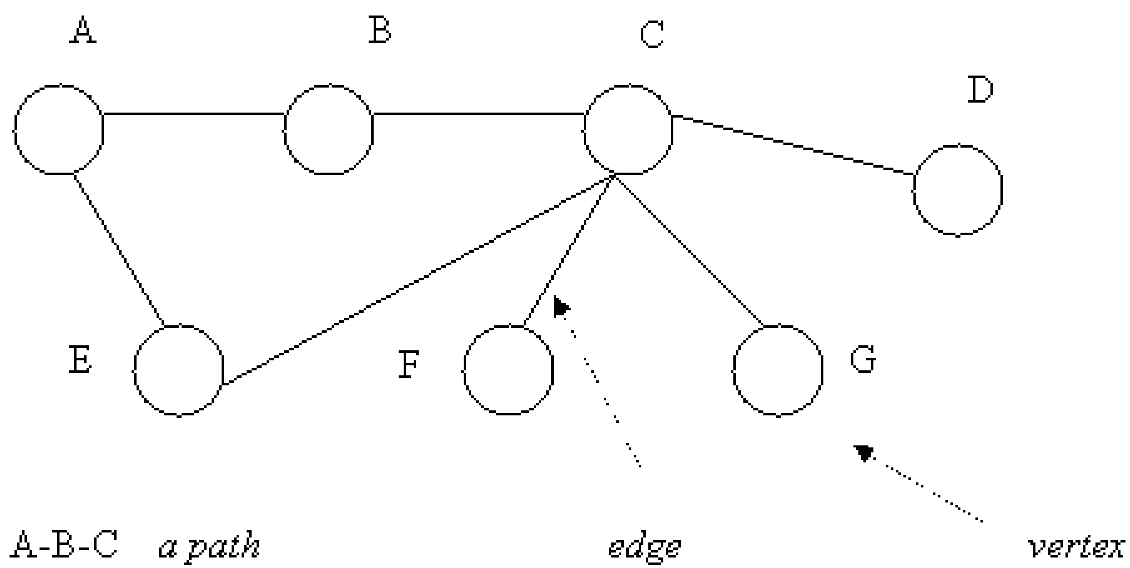
Applications

- Lists
  - Many places in real life applications…
  - Often used to implement other data structures e.g. queues and stacks
  - Used for mathematical vectors and matrices
- Stacks
  - Reverse a string
  - Undo mechanism in text editors
  - Web pages navigation in a web browser
  - Compilers – syntax evaluation
- Queues
  - Applications with a single resource that you are trying to share
  - Printer queues
  - Email message queues
  - Processor queues
- Hash Tables
  - Address books
  - Linking File name and path (local system)
  - Password verification

Graphs

- We are going to look in depth at a very useful data structure
- This is the **Graph**
- Example of uses include:
  - Road maps
  - Project networks
  - Electrical circuits
  - Molecules
    - Relationships between genes, proteins, pathways, etc…
  - Relationships
    - Family tree
    - Students on courses

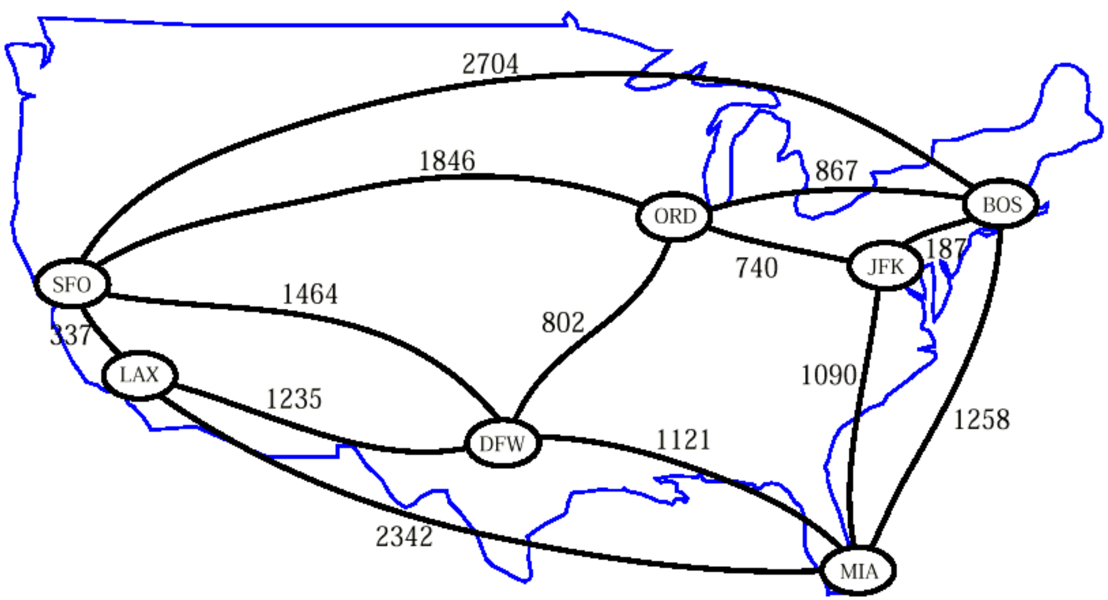
An Abstract View of Graphs



A graph is a collection of nodes (**vertices**) which maybe connected in pairs by line segments called **edges**

Why Study Graphs? – Part 1

- Example
  - Airline route map - an undirected graph
  - Cities – points (nodes, vertex)
  - Non-stop flight - lines connecting two cities (edges, arcs)

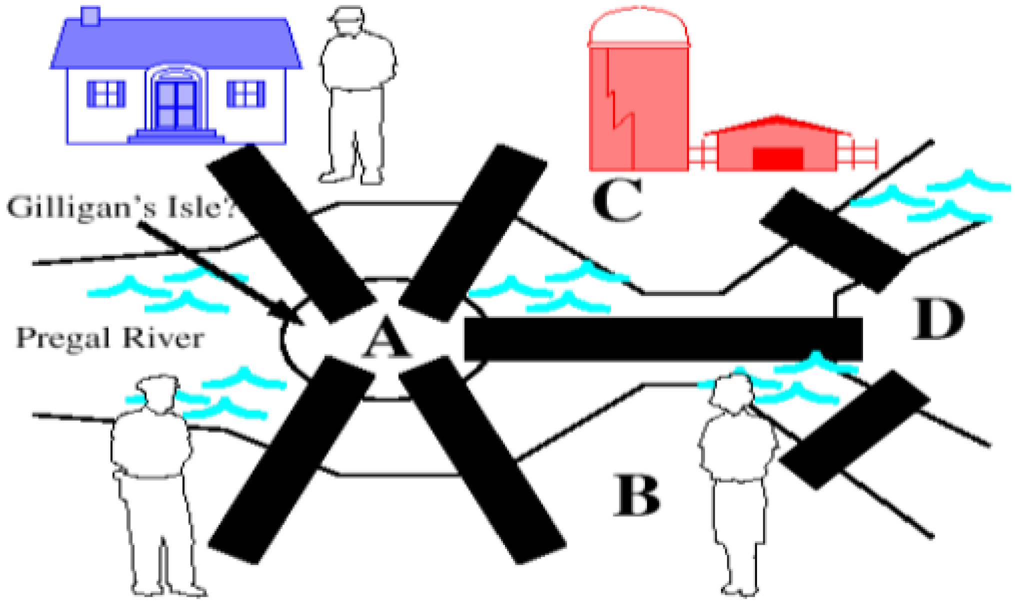


- What can we do with this graph?
  - Shortest path
  - Quickest flight
  - Cheapest way

Why Study Graphs? – Part 2

- Computer Networks
  - Computers – points (nodes, vertex)
  - Cables - lines connecting two computers (edges, arcs)
- What are your possible tasks?
  - Shortest cables
  - Lowest cost
  - Quickest delivery
  - Reliable
  - Fault-tolerant
  - ……
- Many, many more applications (tube, sat. nav.)

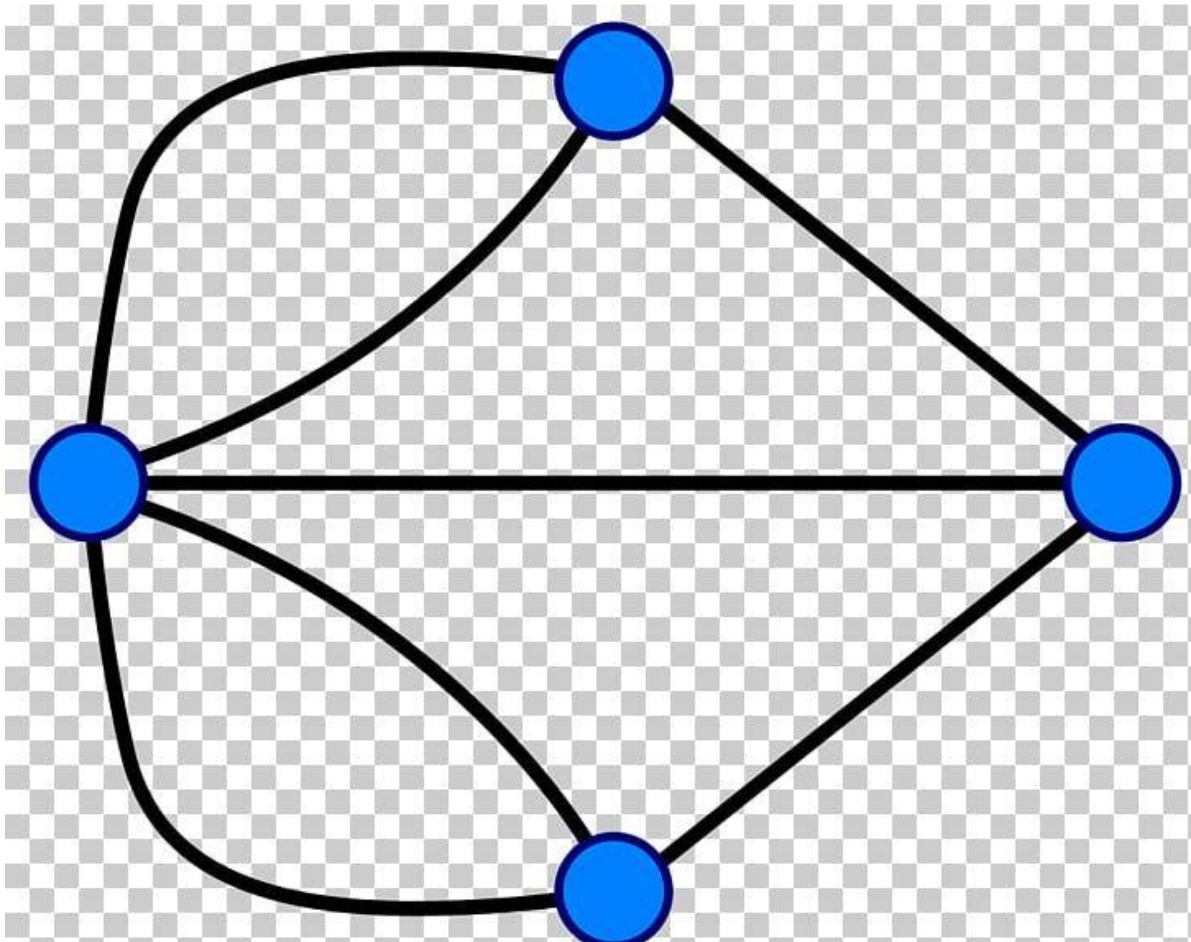
The Bridges of Konigsberg



Consider if you were a UPS driver and you didn't want to retrace your steps!

- Which route allows someone to cross all bridges exactly once?
- In 1736, Euler proved that this is not possible!
- Led to the field of Graph Theory

The Bridges of Konigsberg

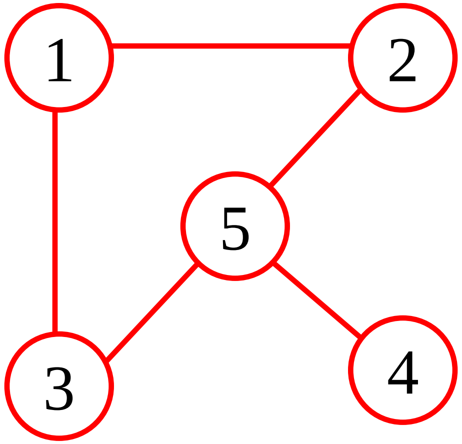


How Can a Graph Help?

- Questions
  - What is the cheapest way to fly from London to Rome?
  - Which route has the least flying time?
  - If Heathrow is closed by bad weather, can you still fly between every other pair of cities, such as Edinburgh-Rome, Manchester-Rome?
  - If one computer in a network goes down, can email be sent between every other pairs of computers in the network?
- Graph algorithms can solve the above problems!

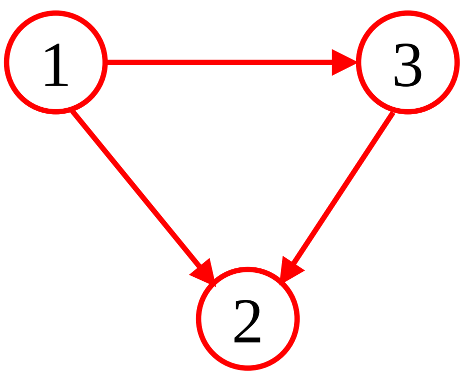
How to Define a Graph?

- What are the basic components of a graph?
- A graph $G = (V,E)$ is composed of:
  - V: a set of vertices or nodes
  - E: a set of edges or lines connecting the vertices in V
  - An edge $e=(u,v)$ is a connection between the vertices u and v
- In the example below:
  - $V= \{1,2,3,4,5\}$
  - $E= \{(1,2),(1,3),(3,5),(2,5),(5,4)\}$



Graph Definitions – Part 1

- Directed Graph
  - A directed graph is a pair $G=(V, E)$
  - Where V is the set of vertices, and E is a set of ordered pairs of elements of V
  - For directed edges (v, w) is in E, v is tail, w is head
- It can be represented as $v \rightarrow w$ or vw
- What is V and E for example below?



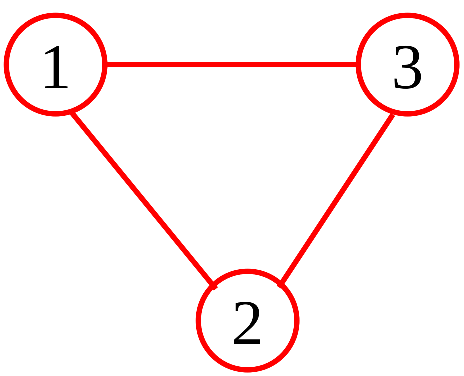
G=(V, E)

V={1, 2, 3}

E={(1,2), (1,3), (3,2)}

Graph Definitions – Part 2

- Undirected Graph
  - An undirected graph is a pair $G=(V, E)$, where E is a set of **unordered** pairs of distinct elements of V
  - Edges have no orientation
  - For undirected graphs, $vw = wv$
- What is V and E for example below?



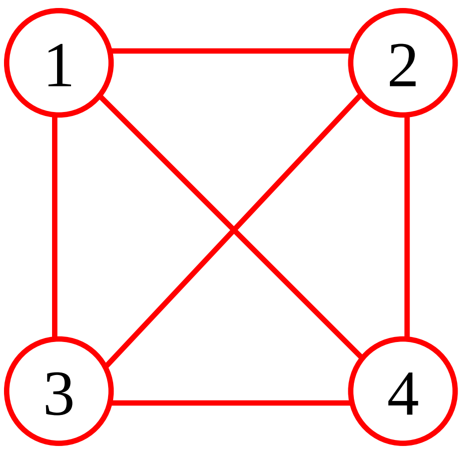
G=(V, E)

V={1, 2, 3}

E={(1,2), (1,3), (2,3)}

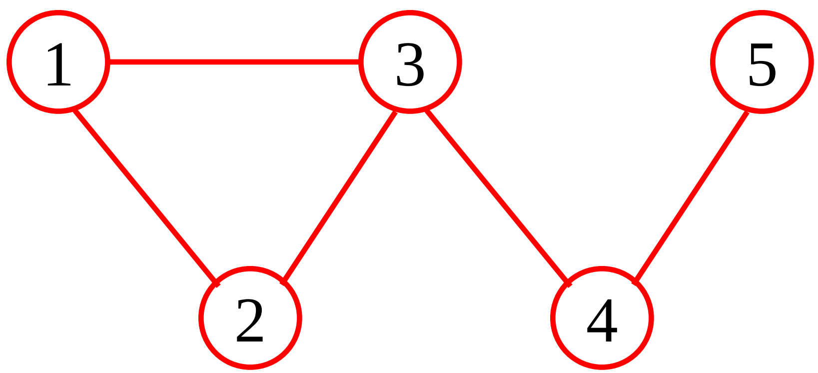
Graph Definitions – Part 3

- Complete Graph
  - A complete graph is normally an undirected graph with an edge between **each** pair of vertices
  - $G=(V, E)$
  - $\forall e \in V \times V \Rightarrow e \in E$



Graph Definitions – Paths

- A sequence of k vertices, $[v_1, v_2, \ldots, v_k]$, such that any pair of consecutive vertices, v_i, v_{i+1} are adjacent (connected by an edge) is called a **path**

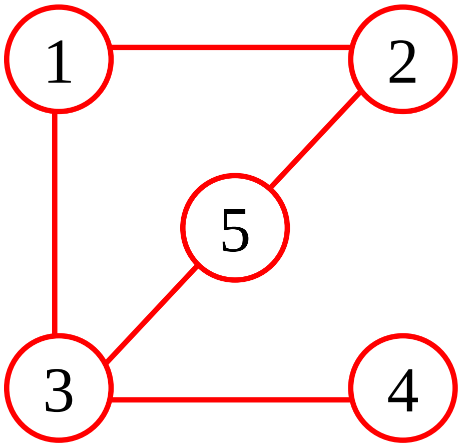


- Which of the followings are paths?

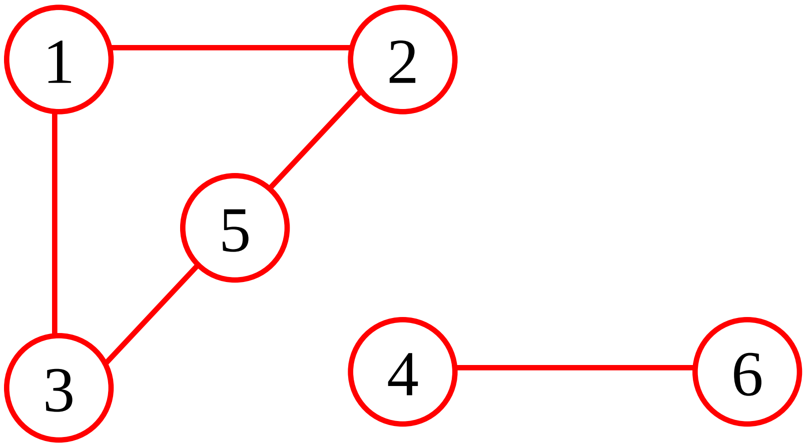
- [1,2,3,4,5] is a path
- [1,5,2,4] is not a path
- [1,2,3,1] is a path that contains a **cycle**

Graph Definitions – Connectivity

- Connectivity
  - An undirected graph is connected if and only if for each pair of vertices v and w, there is a path from v to w
  - A directed graph is strongly connected if and only if for each pair of vertices v and w, there is a path from v to w



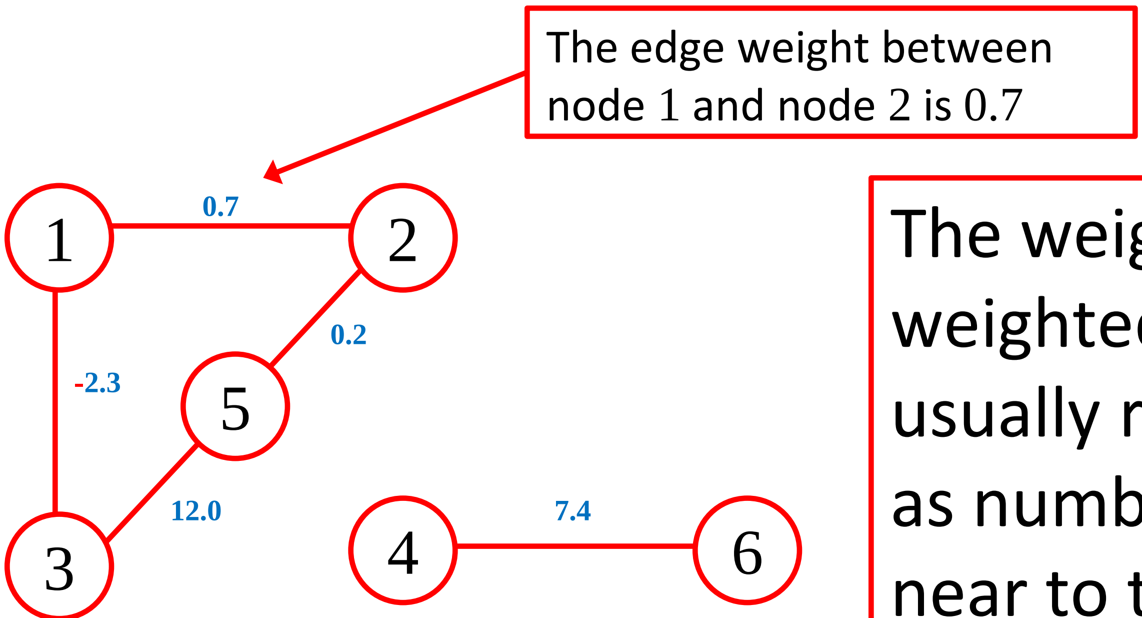
Connected



Not Connected

Graph Definitions – Weighted Graph

- A weighted graph is a triple $G=(V, E, W)$
- Where $W: E \rightarrow \boldsymbol{R}$
- $W(e)$ is called the weight of edge e



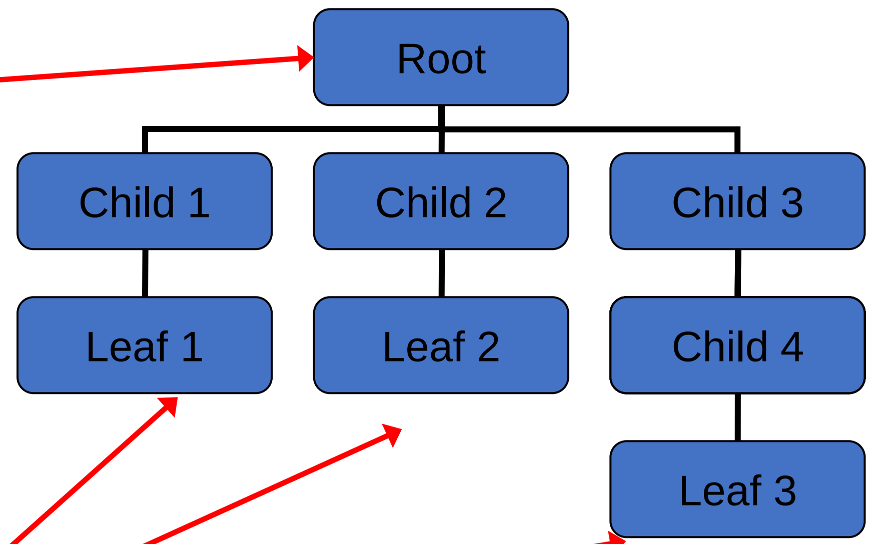
The weights in a weighted graph are usually represented as numbers centred near to the edges they apply to

How do we represent a graph when it comes to implementation?

- We often represent a graph as a **matrix (2D array)**, although other data structures can be used depending on the application
- If we have N nodes to represent
  - For an N **by** N **matrix** G, **a non-zero value of** g_{ij} **(**i**th row,** j**th column of** G**) means there is an edge between node** i **and** j
- **Undirected**
  - We assume that g_{ij} is the same as g_{ji}
- **Directed**
  - g_{ij} is not always the same as g_{ji}
- **Non-weighted**
  - g_{ij} is either one for an edge or zero for no edge
- **Weighted**
  - g_{ij} is the edge weight or zero for no edge
- **Complete**
  - g_{ij} is never zero

Trees – Part 1

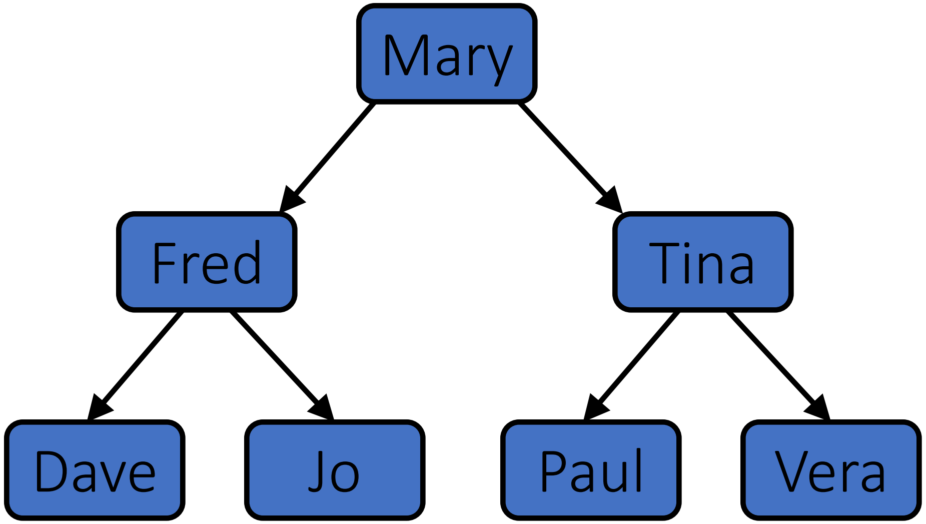
- These are special graphs that are without cycles
- Hierarchical graph
  - No cycles
- Root
  - The only node at the topmost part of the tree
  - Child nodes have parents
- All of the rest of the nodes must be linked to a parent node, and may have zero or more child nodes
- Leaf
  - A Node without children



Trees – Part 2

- A **binary** tree is a special type of tree where the maximum number of children is two
- Trees are usually ordered from the top to the bottom and left to right
- The height of a tree is the number of levels
- Trees are very fast to search
  - E.g. Binary search
- There are a very large number of applications of trees
  - Spell checkers
  - Parse trees in compilers
  - Computer file systems
  - Organisational structures and hierarchies
  - Gene ontology data (functional relationships)
  - Etc...

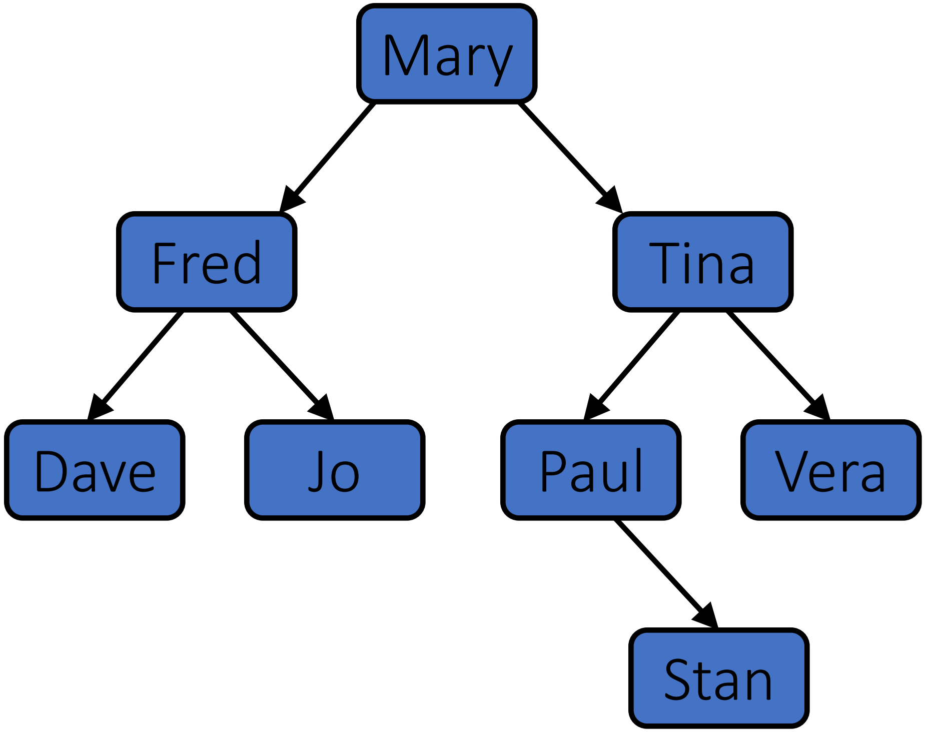
Searching for an Item in a Tree



- The following **binary tree** stores names
- Left side children appear before the parent in the alphabet
- Right side children appear after the parent in the alphabet

- Searching for Jo
  - Start at the root node Mary
  - Less than Mary, move to Fred
  - Greater than Fred, move to Jo and we have found the key
- Searching for Stan
  - Start at the root node Mary
  - Greater than Mary, move to Tina
  - Less than Tina, move to Paul
  - Greater than Paul, no right node, report failure to find key

Adding an Item into a Tree



- To add a node, we first search for the node we are adding
- Searching for Stan
  - Start at the root node Mary
  - Greater than Mary, move to Tina
  - Less than Tina, move to Paul
  - Greater than Paul, no right node, failure to find key...
- Now we add the node where we expected to find it
- Deleting is more difficult...
  - What if we delete Mary?

- Trees can become **unbalanced**
- Especially if we add and remove a large number of nodes
- The height of the tree can grow until the tree becomes a **linked list**
- Special types of self balancing trees have been developed:
  - AVL, Red-Black, etc...

This Weeks Laboratory

- Class Test CR I will be released today!
- This laboratory is one of the worksheets you may be assessed in **Task #1 and/or #2**
- You will be implementing and studying a number of data structures
- It is **very important** as many of the future laboratory worksheets will use the data structures we are going to cover

Next Lecture

- We will be looking at **sorting** in more detail...