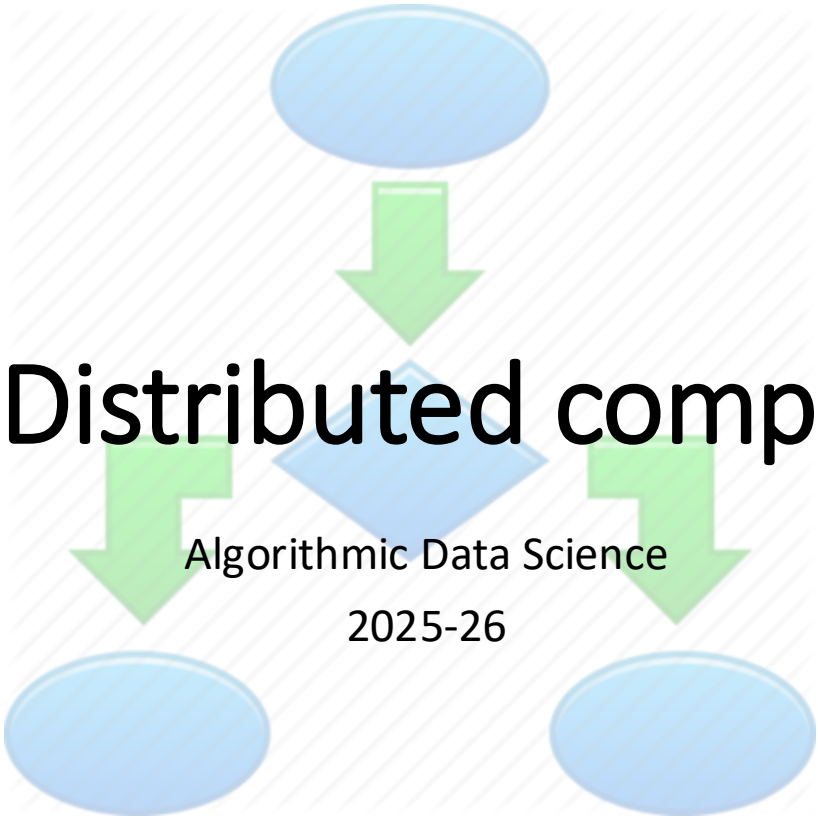


Week 5: Distributed computation

Algorithmic Data Science

2025-26

UNIVERSITY OF SUSSEX

Dr Adam Barrett

Warm up / revision

1. Suppose a dictionary has a good hash function. What is the time to access a random item from it?

(a) O(1) (b) O(log n) (c) O($\sqrt{n}$) (d) O(n)

2. Recall that a hash collision refers to when 2 or more keys hash to the same `bucket' (slot). Chaining resolves this by placing all items that hash to the same slot into a linked list. Suppose now a dictionary has a less good hash function, and on average each used slot has 3 items in it (and at most 10 items in it). What is the access time now for a random item?

(a) O(1) (b) O(3) (c) O(n) (d) O(n^3).

3. Now suppose things are even worse and any 2 keys picked at random have a 1 in 3 chance of hashing to the same slot. What is the access time now?

(a) O(1) (b) O(n) (c) O(3n) (d) O(n^3).

Warm up / revision

1. Suppose a dictionary has a good hash function. What is the time to access a random item from it?

(a) O(1) (b) O(log n) (c) O($\sqrt{n}$) (d) O(n)

2. Recall that a hash collision refers to when 2 or more keys hash to the same `bucket' (slot). Chaining resolves this by placing all items that hash to the same slot into a linked list. Suppose now a dictionary has a less good hash function, and on average each used slot has 3 items in it (and at most 10 items in it). What is the access time now for a random item?

(a) O(1) (b) O(3) (c) O(n) (d) O(n^3).

3. Now suppose things are even worse and any 2 keys picked at random have a 1 in 3 chance of hashing to the same slot. What is the access time now?

(a) O(1) **(b) O(n)** (c) O(3n) (d) O(n^3).

Main topics per week

Week	Topic
1	Data structures and data formats
2	Algorithmic complexity. Sorting.
3	Matrices: Manipulation and computation
4	Processes and concurrency
5	**Distributed computation**
6	Similarity
7	Map/reduce
8	Graphs/networks
9	Graphs/networks, PageRank algorithm
10	Databases
11	*independent study*

Overview for today

Sorting revisited: Heaps and the heapsort algorithm

Bit more on parallelism and concurrency:

- Heaps for process queuing.
- CPUs and GPUs for parallelism.

Distributed computation

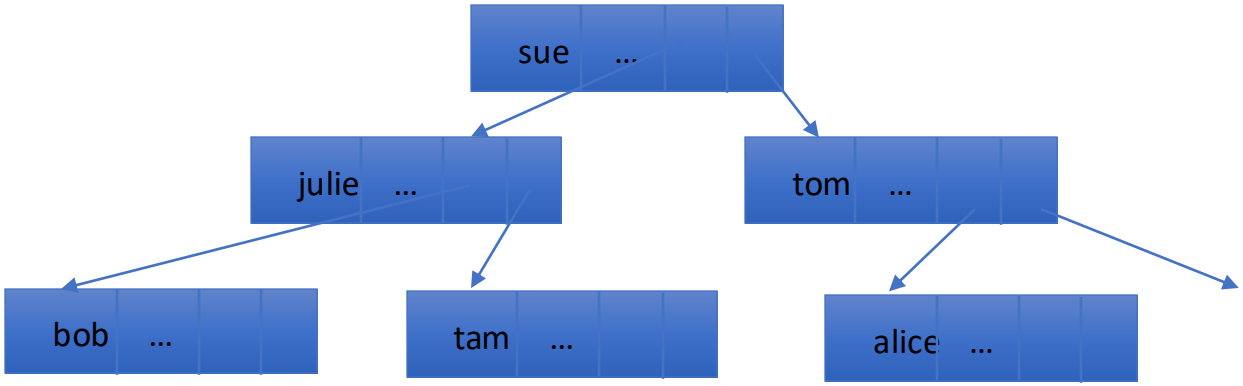
- What are data centres?
  - Basic set-up and fault tolerance
  - Briefly, synchronization and Remote Procedure Call for communication.
- Essentials of cybersecurity - authentication.
  - Hashing
  - Encryption

Recap of last week

- **Processes** are programs (sequences of instructions that often take inputs and generate outputs), which run either in **parallel** or **concurrently** sharing CPU cores.
- Processes may have one or multiple **threads** — which are lightweight processes themselves.
- Processes have a **life cycle**, which includes creation, being ready, running, waiting, and termination.
- Processes can be synchronised/coordinated via locks or other methods.

Heaps and the heapsort algorithm

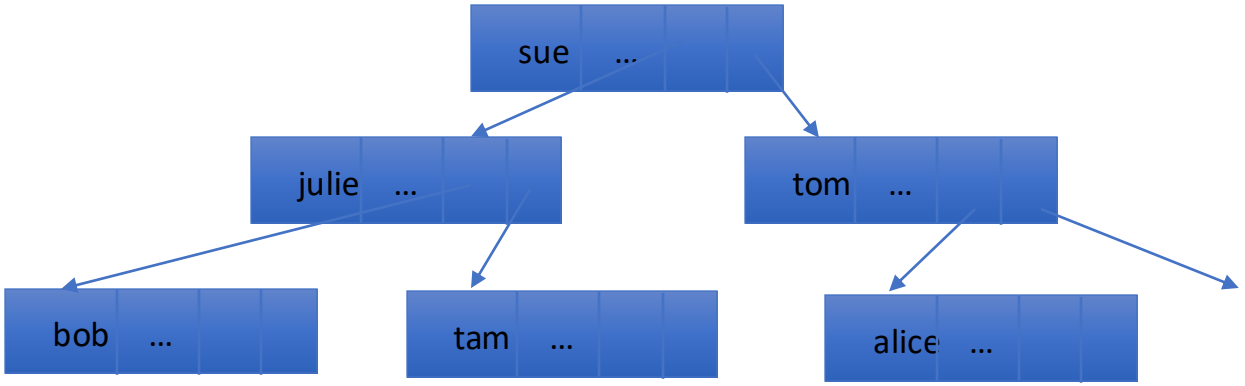
- Heaps are **binary trees** which satisfy the **heap property**: Value[parent(i)] ≤ Value[i]



- This is a complete binary tree. Does it satisfy the heap property?

Heaps

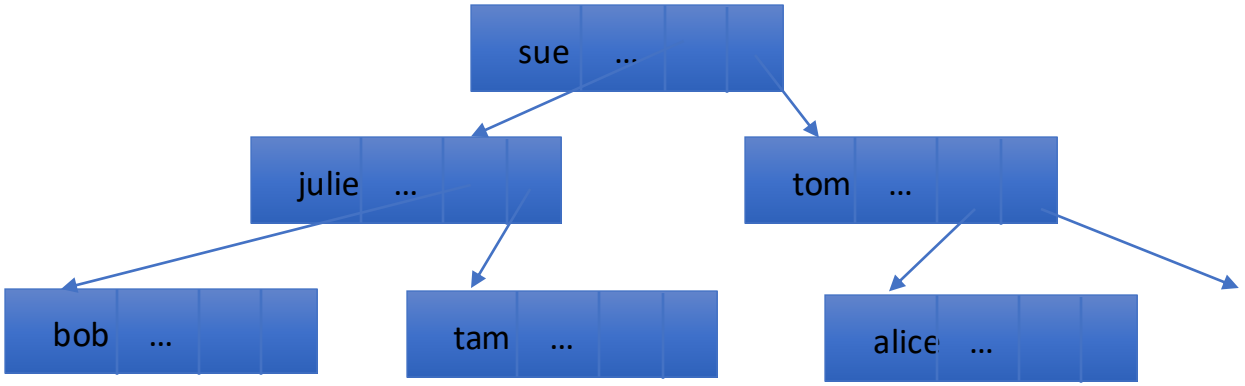
- Heaps are **binary trees** which satisfy the **heap property**: Value[parent(i)] ≤ Value[i]



- Run heapify on all non-leaf nodes in a bottom-up fashion i.e.
  - heapify([tom, julie, sue])

Heaps

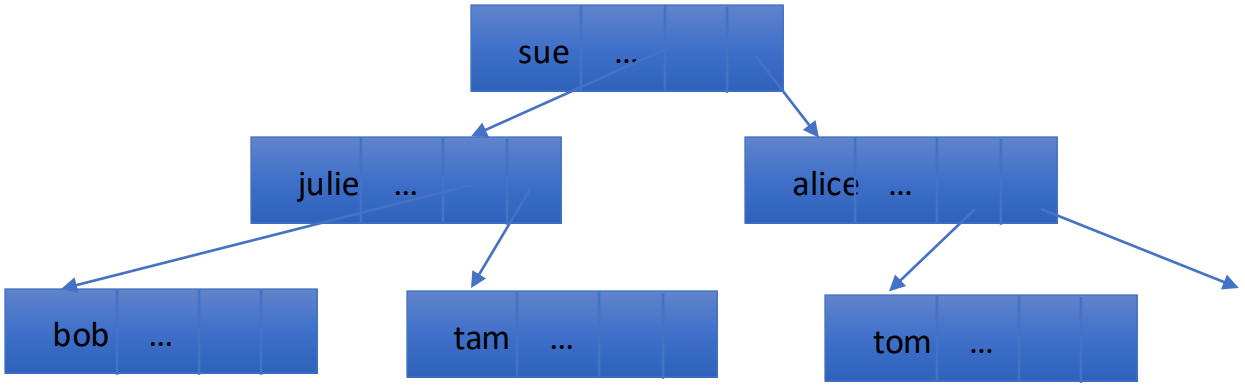
- Heaps are **binary trees** which satisfy the **heap property**: Value[parent(i)] ≤ Value[i]



- heapify(tom) :- tom > alice so swap these nodes

Heaps

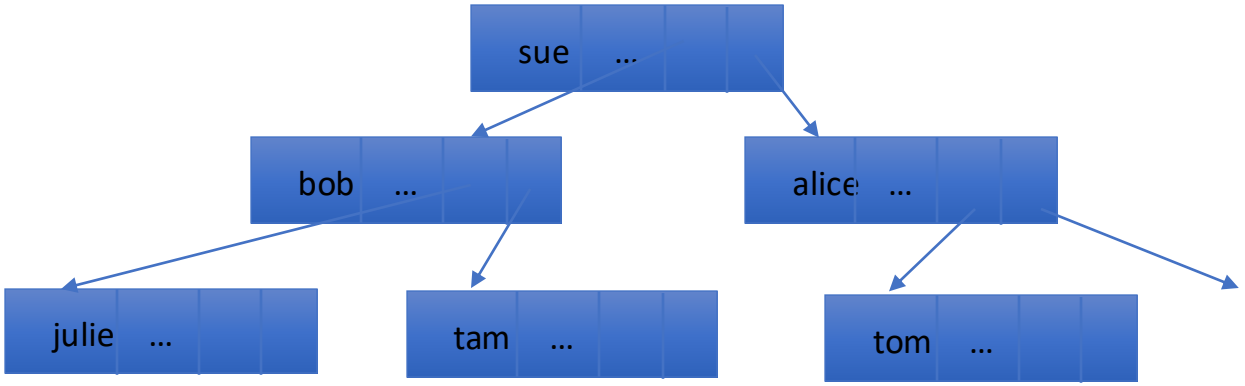
- Heaps are **binary trees** which satisfy the **heap property**: Value[parent(i)] ≤ Value[i]



- heapify(julie) :- Out of Julie Bob and Tam, Bob is first in alphabet, so swap Bob and Julie

Heaps

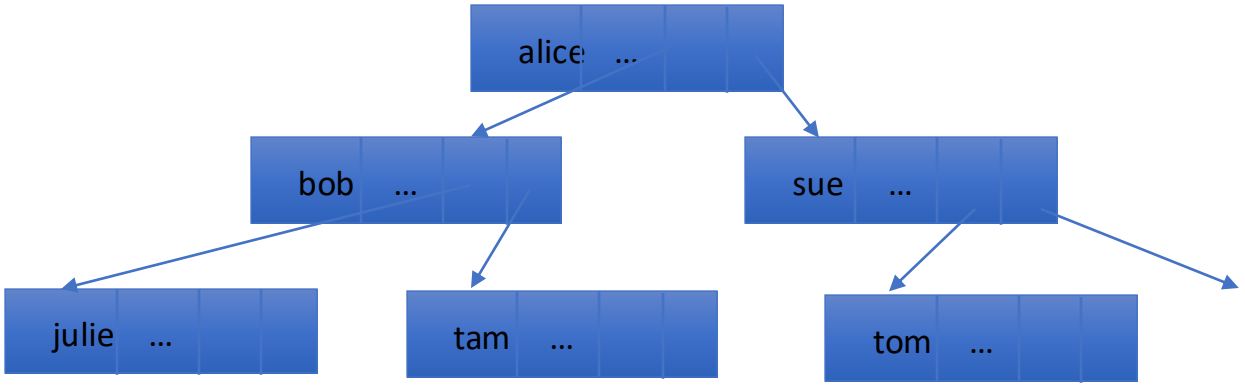
- Heaps are **binary trees** which satisfy the **heap property**: Value[parent(i)] ≤ Value[i]



- heapify(sue) :- Out of Sue Bob and Alice, Alice is first in alphabet, so swap Sue and Alice

Heaps

- Heaps are **binary trees** which satisfy the **heap property**: Value[parent(i)] ≤ Value[i]



  - heapify(sue) :- Sue is before Tom in the alphabet, so can stay put.
  - Heap property now satisfied.

Heapsort algorithm

newlist=[]
While heap non-empty
    Append element at top of heap to **newlist**.
    Take the element at the bottom of the heap, and put it at the top of the heap.
    Run **heapify**

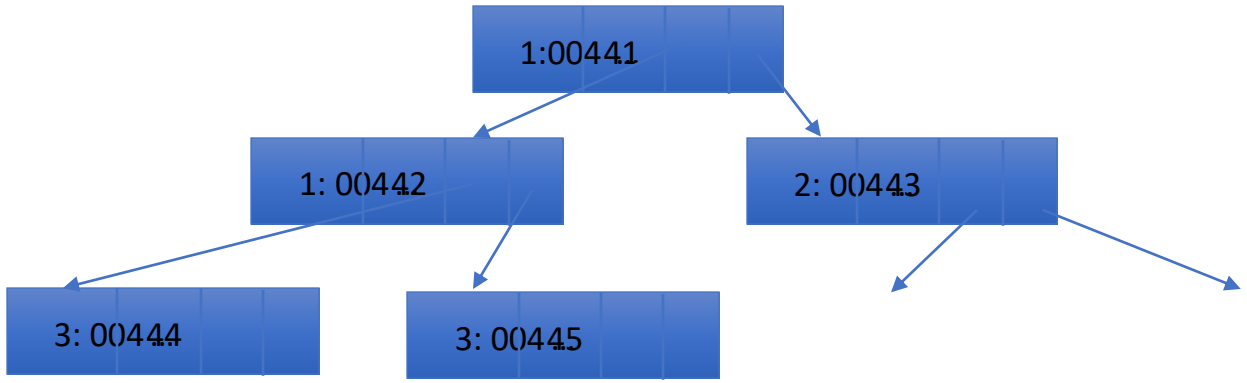
- O(*n* log *n*) to build a heap
- O(*n* log *n*) to extract sorted list from a heap.

- Can use heaps to efficiently implement a '**priority queue**':
  - Action done on top element.
  - Heapify to find new top element
  - Repeat

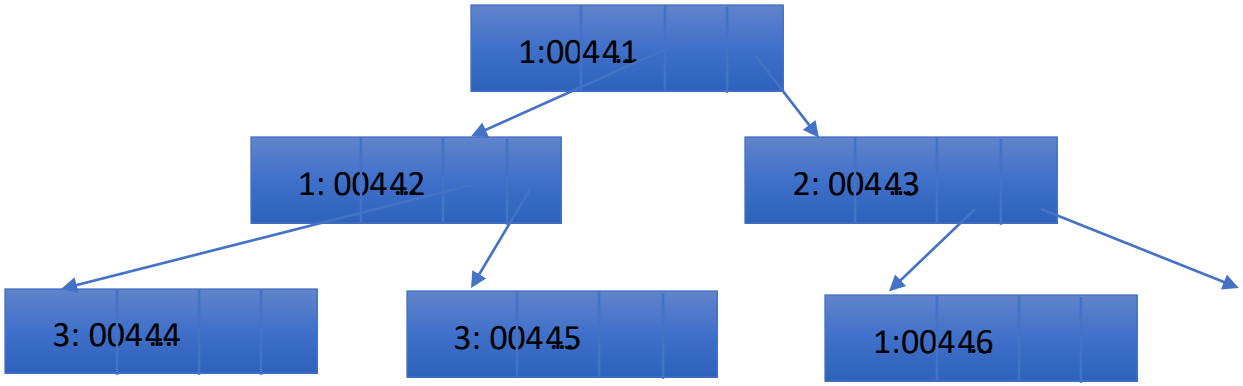
Priority queues as heaps

- Recall: heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]

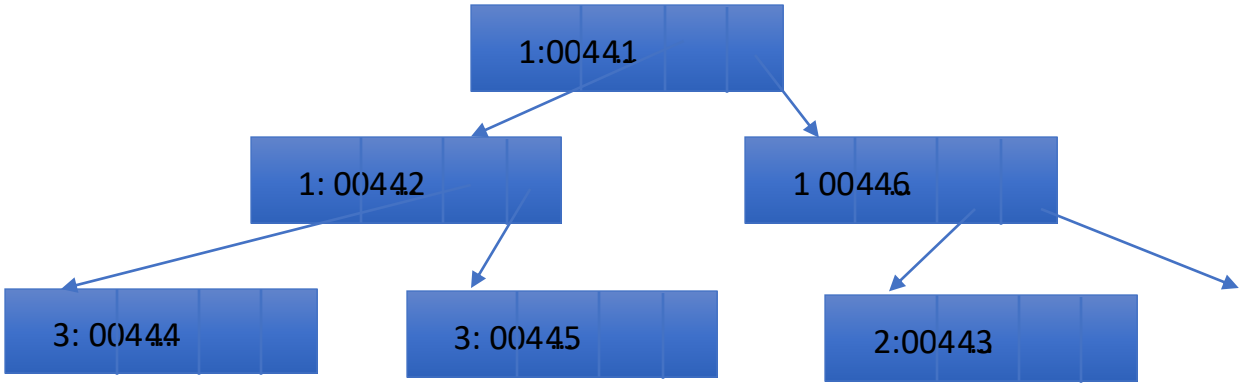
- Process IDs can be arranged in a heap, with priority labels attached to them, here, 3 levels of priority: 1 high priority, 2 medium, 3 low.



A new process is added to the queue and finds its place

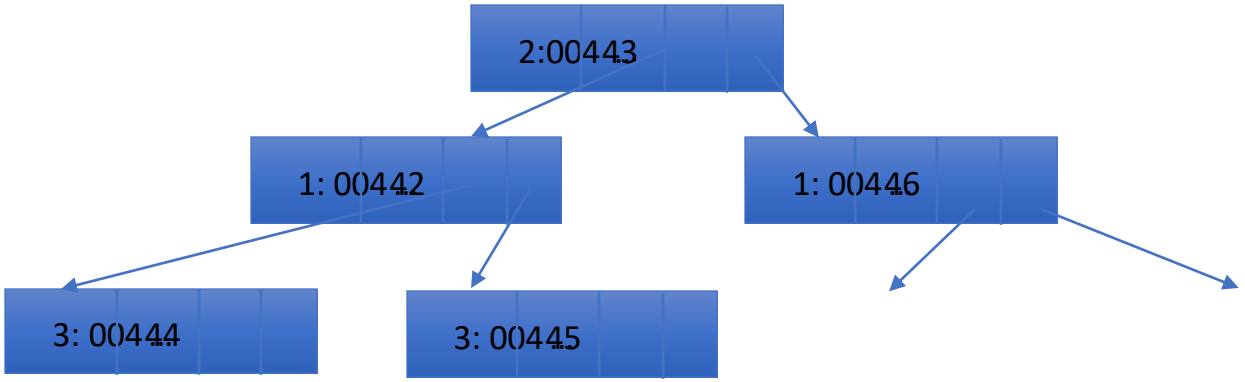


A new process is added to the queue and finds its place



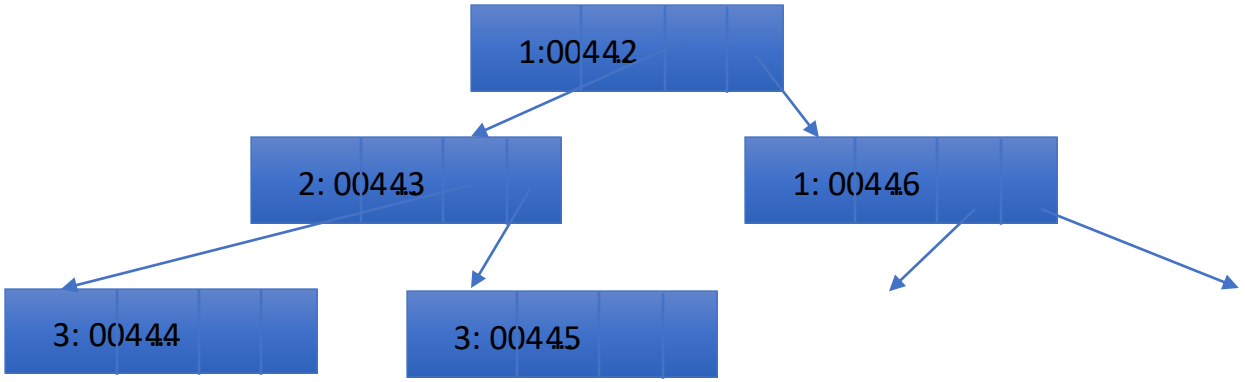
The process at the top of the heap enters the CPU and is removed from the heap. . .

- Then the process at the bottom of the heap is placed at the top and is sifted.

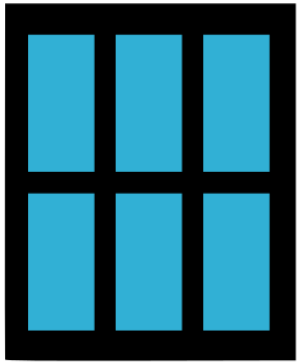


The process at the top of the heap enters the CPU and is removed from the heap. . .

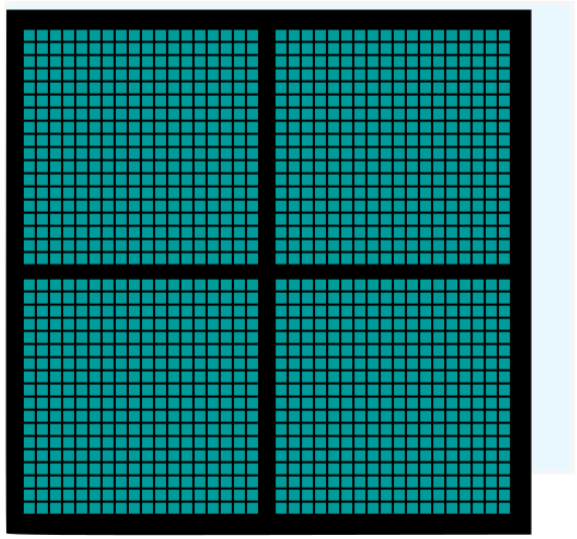
- Then the process at the bottom of the heap is placed at the top and is sifted.



CPUs and GPUs (Graphic processing unit)



CPU
Multiple Cores



GPU
Thousands of Cores

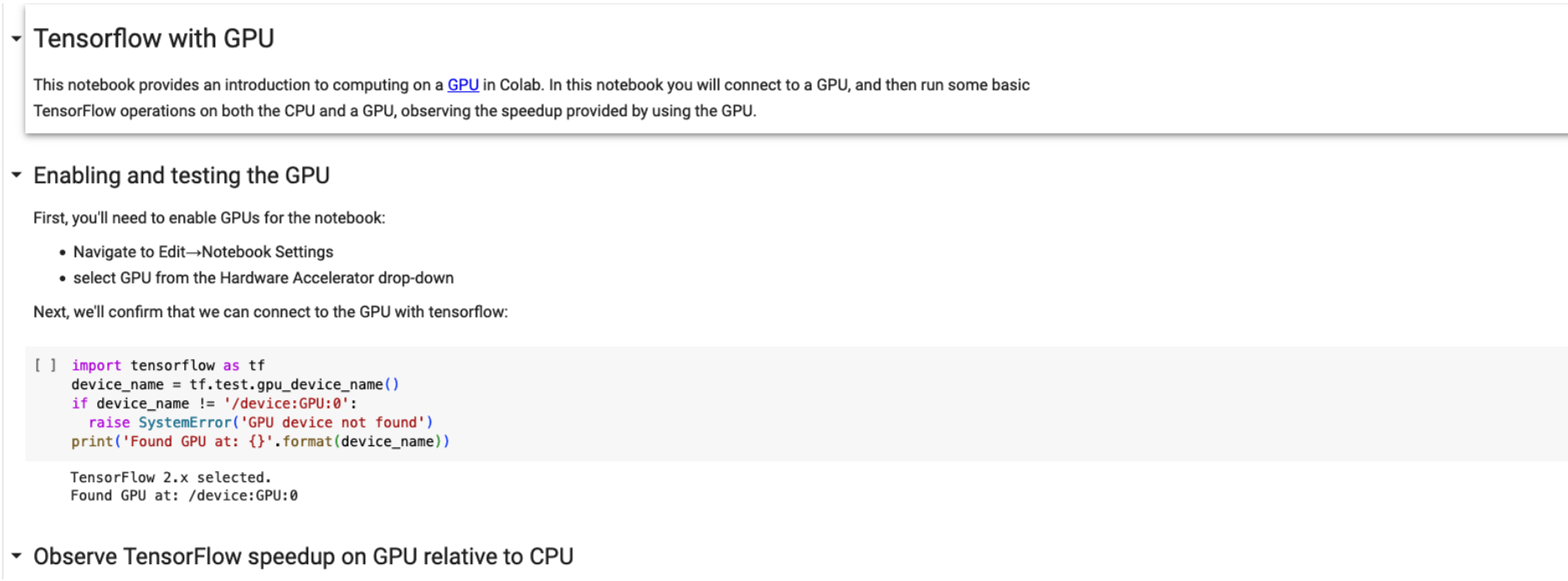
From teachcomputerscience.com

CPUs vs GPUs

CPU	GPU
Larger memory capacity	Smaller memory capacity
Slower at loading memory in from outside – low bandwidth	Quicker at loading in memory from outside - high bandwidth
Does better when process involves lots of data update.	
Faster clockspeed, but fewer cores	Slower clockspeed, but more cores.
Will be better for **task parallelism**: different tasks done on a big chunk of data simultaneously.	Good for **data parallelism**: repetitive simple task done on many small chunks of data. E.g. basic mathematical operations on arrays can be done in parallel. Useful in machine learning.

GPU demo on Google Colab

- High-end laptops now have GPUs for computing.
- The lab machines in Chichester 1 have GPUs.
- You can access GPUs on a Google server at Colab for free- link below for basic machine learning demo.



Tensorflow with GPU

This notebook provides an introduction to computing on a GPU in Colab. In this notebook you will connect to a GPU, and then run some basic TensorFlow operations on both the CPU and a GPU, observing the speedup provided by using the GPU.

Enabling and testing the GPU

First, you'll need to enable GPUs for the notebook:

- Navigate to Edit→Notebook Settings
- select GPU from the Hardware Accelerator drop-down

Next, we'll confirm that we can connect to the GPU with tensorflow:

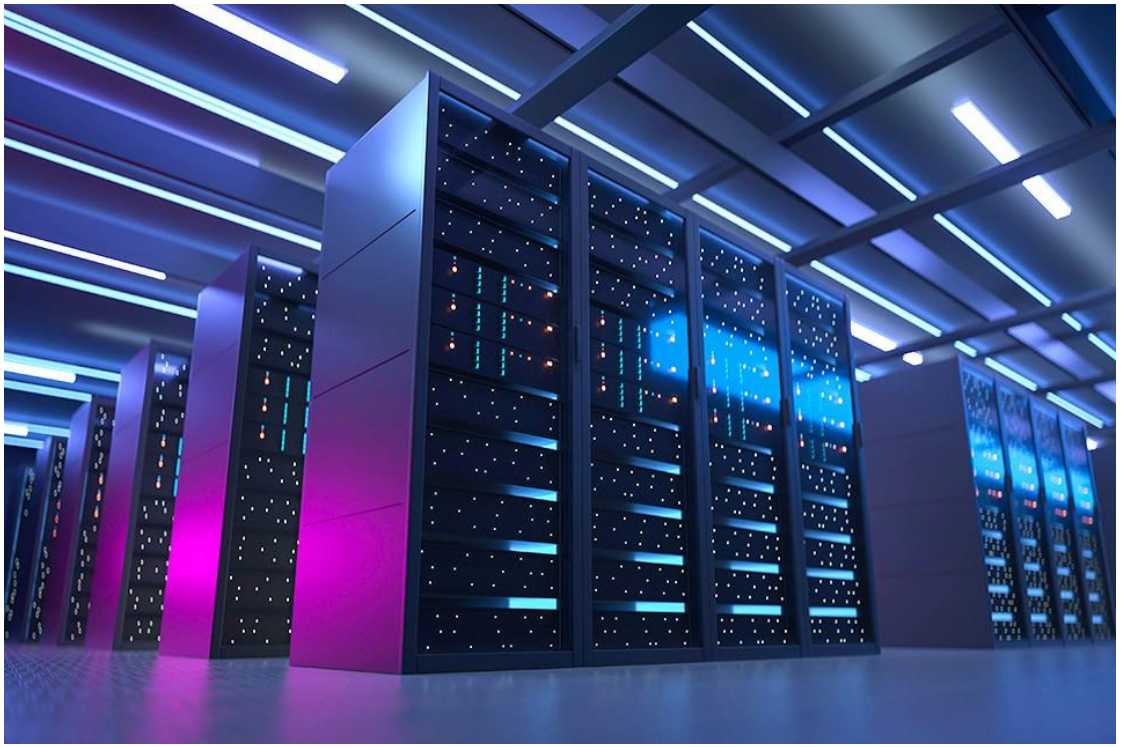
```
[ ] import tensorflow as tf
    device_name = tf.test.gpu_device_name()
    if device_name != '/device:GPU:0':
      raise SystemError('GPU device not found')
    print('Found GPU at: {}'.format(device_name))
```

```
TensorFlow 2.x selected.
Found GPU at: /device:GPU:0
```

Observe TensorFlow speedup on GPU relative to CPU

https://colab.research.google.com/notebooks/gpu.ipynb

What is a data centre



What is a data centre

As data scientists, your networks will be:

- A link into a Data Centre, and then inter-machine links within a Data Centre.
- High bandwidth: 1 GBit/s or higher
- Low latency
  (apart from desktop to data centre).

You will be a ***client***, and the data centre will be a ***server.***

What is a data centre

As client, you may use “machines” that are:

- Within the data centre
- Often virtual, running on a physical machine
- Most likely running versions of Linux
- Your processes will send messages to each other to communicate.
- These message will use TCP (transmission control protocol) for reliable delivery

Client-server architecture

Scenario: a server offers services desired by (potentially many) clients. All requests from clients and services provided by the server are delivered over a network.

How it works:

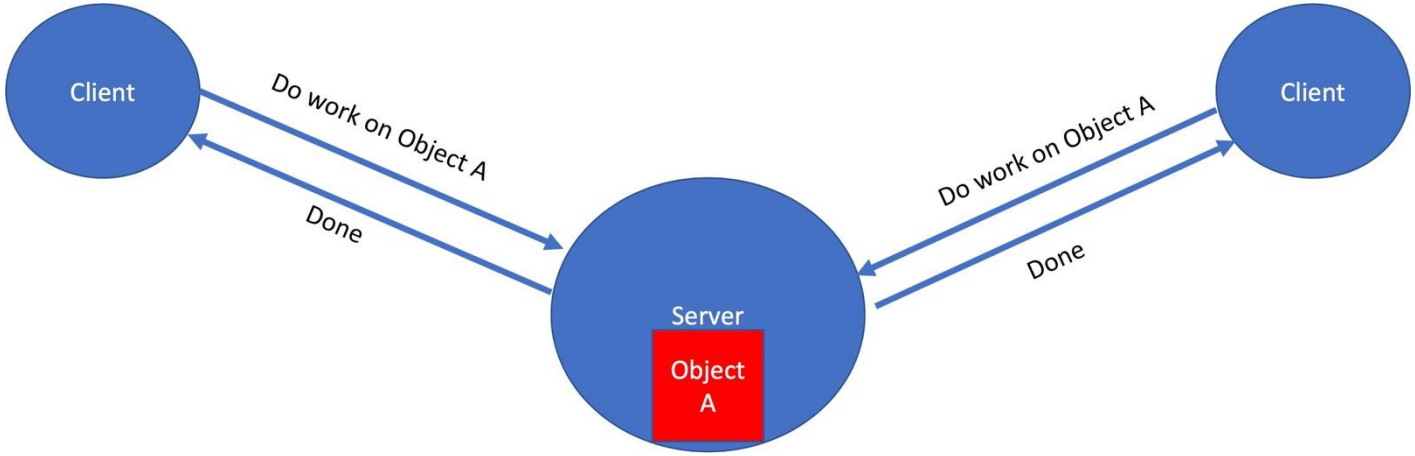
- First, the client sends their request via a network-enabled device.
- Then, the network server accepts and processes the user request.
- Finally, the server delivers the reply to the client

… this is similar to how processes share a CPU core!

Synchronisation across machines

Data centres are usually accessed and used by multiple users/clients.

How can we ensure that the different machines don't corrupt *Object A*?

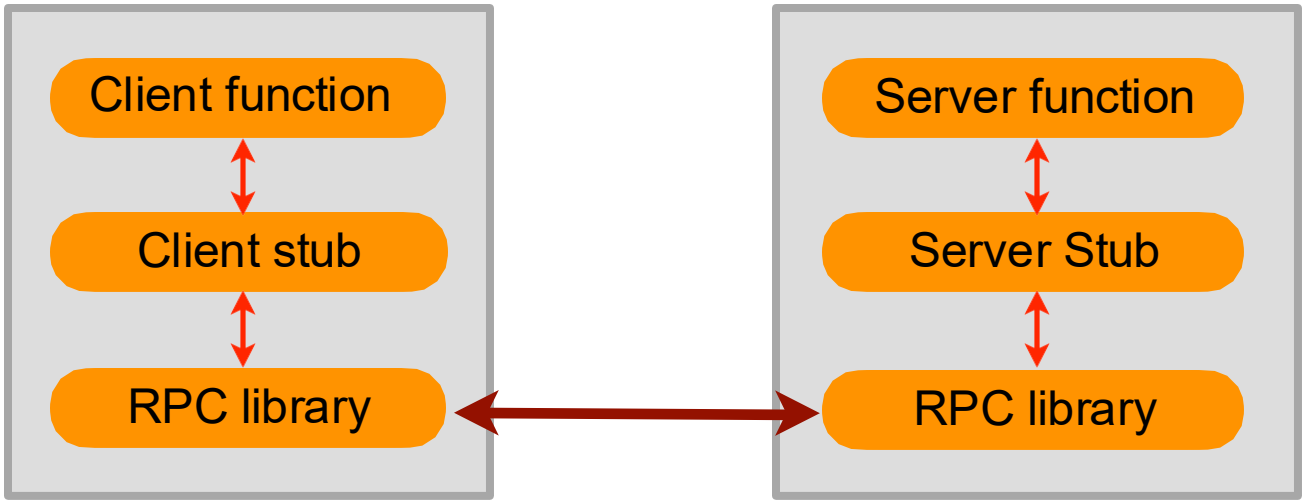


Synchronisation across machines

- Atomic (all or none) send/receive of messages.
- Use e.g. file locking (similar to locks for threads on single CPU…)
  - Often distributed systems hold multiple copies of the same file. So a lock implies messages notifying that all copies of the file are locked as well.
  - Writing only takes place after confirmation of local locking has been made.

Remote Procedure Call

Call a procedure on a remote machine



The stubs act as intermediaries that help different components or systems understand and communicate with each other, even when there are differences in data formats, representations, or languages.

Fault-tolerant distributed systems

If a machine fails, then we can still provide a service
— > **key idea**: have ***redundancy***

- For example, multiple machines providing the same service (such as storing data).
- Probability of total failure (such as data being lost) is reduced, since data is replicated across multiple machines.
- If probability of failure is p for a given machine, then probability of loss of service with n machines is p^n, and the availability of the service is $1 - p^n$.

- If the mean time between failure for 3 machines is 5 days, and repair time is 4 hours, then assuming independence of failure:

$$p = \frac{4}{5 \times 24} = 0.03, \quad \text{the availability is } 1 - 0.03^3 = 99.96\,\%$$

- Replicate data and monitor operations to enable fault-tolerance.

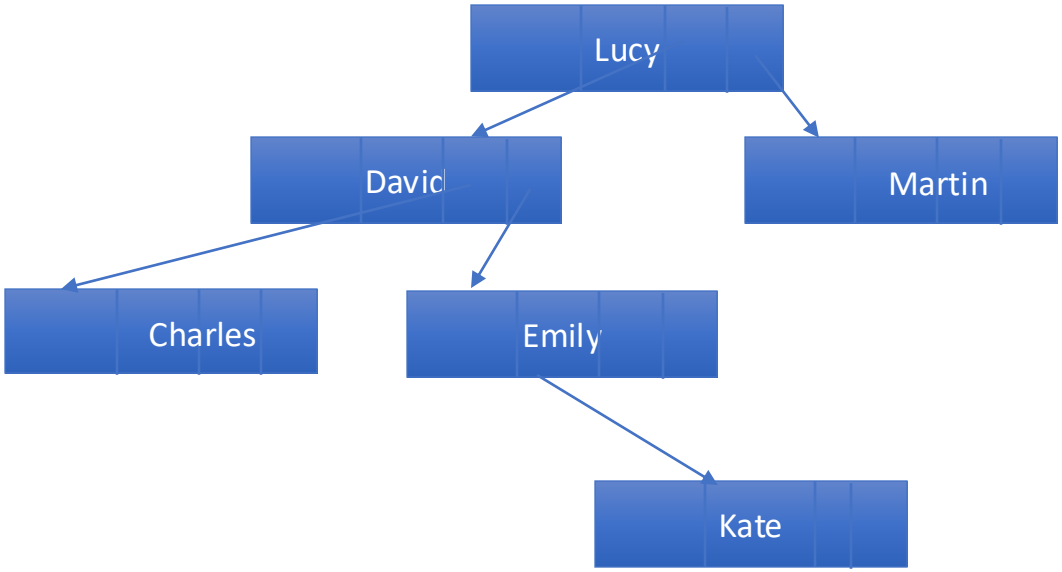
The Google File System (GFS)

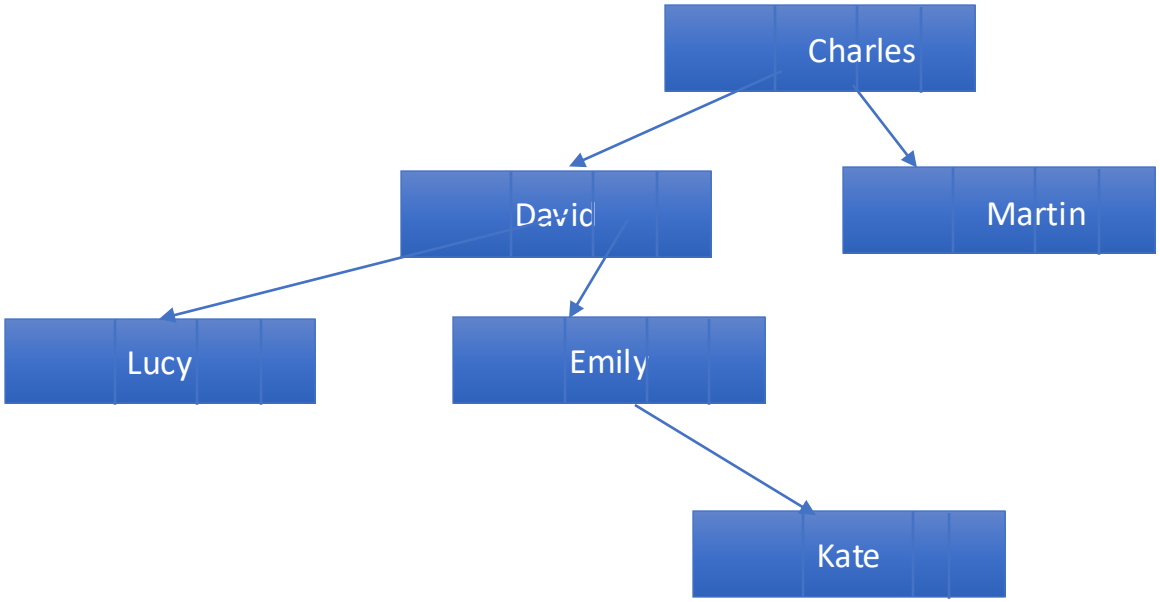
Example: GFS is designed for massive data that rarely gets updated

- 64MB chunks are replicated over chunk servers.
- Meta Data (including where each chunk is, and which chunk is what file) is stored on master servers.
- Master servers monitor the work of chunk servers.

Exercise

Heapify is applied to extract the file for the person whose name comes first in the alphabet. List the operations that are applied to the binary search tree, and draw the heap that is obtained.




Charles
David
Martin
Lucy
Emily
Kate

Securing machines

Security is made of three key pieces:

Authentication: identify who each user is

Authorisation: control who is allowed to do what

Enforcement: ensure that users only do what they are allowed to do

Authentication

Most common approach to implement — via passwords

Acts as a shared secret between two parties. Since only I know password, machine can assume it is me.

Problem: One system must keep copy of secret, to check against password. What if malicious user gains access to this list of passwords?

Solution: Transformation on data that is difficult to reverse

– **hash functions**, aka **secure digest functions**.

Hash Functions

A hash function $h = H(M)$ has the following key properties:

1. Given M, it is easy to compute h.

2. Given h, it is hard to compute M.

3. Given M, it is hard to find another M' such that $H(M)=H(M')$

-Usually, any small change in M generates a big change in h. . .

Typical hashing functions: MD5, SHA-1, SHA-256, SHA-512.

NB. This is a different use for hash functions than we met before, but it's the same properties we're after here too.

Hashing in Python

```
a = 'hello world'

h = hash(a)


import    hashlib

a = 'hello world!'

h_md5 = hashlib.md5(a.encode('UTF-8'))

h_sha1 = hashlib.sha1(a.encode('UTF-8'))
```

Password management via hash

- Consider a user needs to authenticate in a server using a password via a public channel.

  - Password should not be sent publicly.

  - List of passwords should not be stored — could be hacked!

- Idea:

  - Server only store a list of hashes of passwords.

  - Users only send hash of their password, which is then compared against the list.

Hashed-Message Authentication

Can we use message digests to provide authentication?

- Provide a shared secret K to both ends of communication.
- Generate an authentication code for message m, denoted by $(K|m)$.
- Hash the message authentication code, $H(K|m)$.
- Send message m and hash $H(K|m)$.

Receiver then regenerates $H(K|m)$ and compares with the received hash. Only knowledge of the shared secret and an unchanged message could generate the authentication code

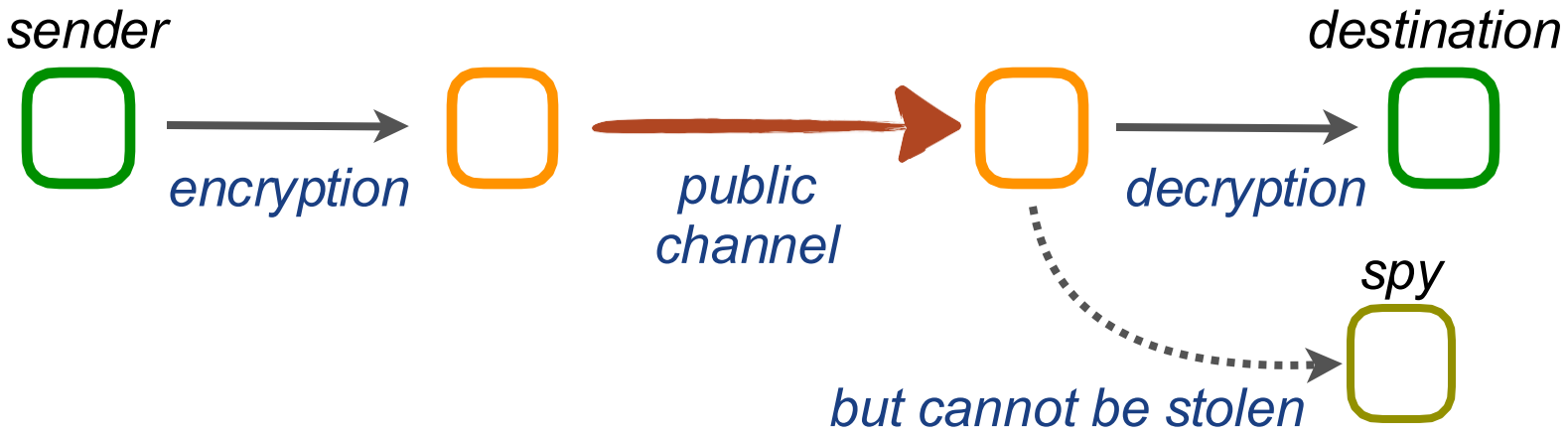
→ Provides authentication and integrity!

- Practical algorithms (e.g. HMAC) use two keys and two hashes to provided added security: $H(K_2 \mid H(K_1|m))$.

Hashing vs encryption

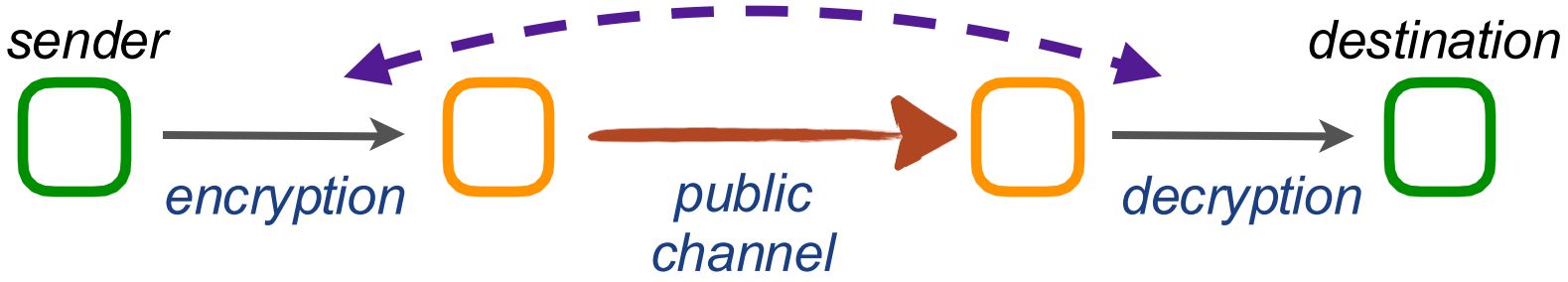
Hashing: a one-way process — easy to do, hard to undo.

Encryption: a two-way process, where decoding is possible under conditions…



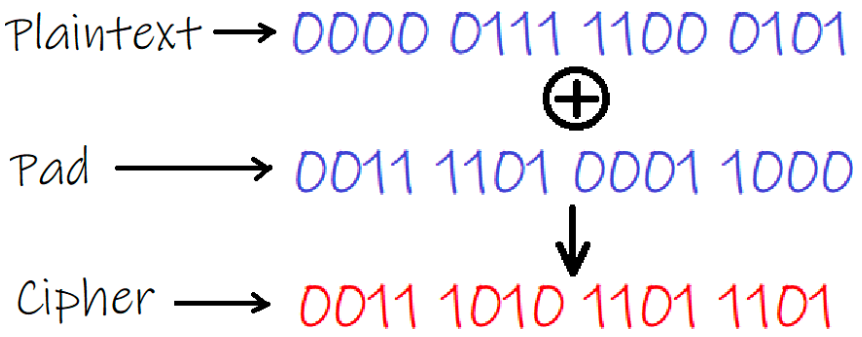
Symmetric encryption

Symmetric: same key is used for coding and decoding



Simple, but not very flexible…

Example — *one-time pad* (OTP)

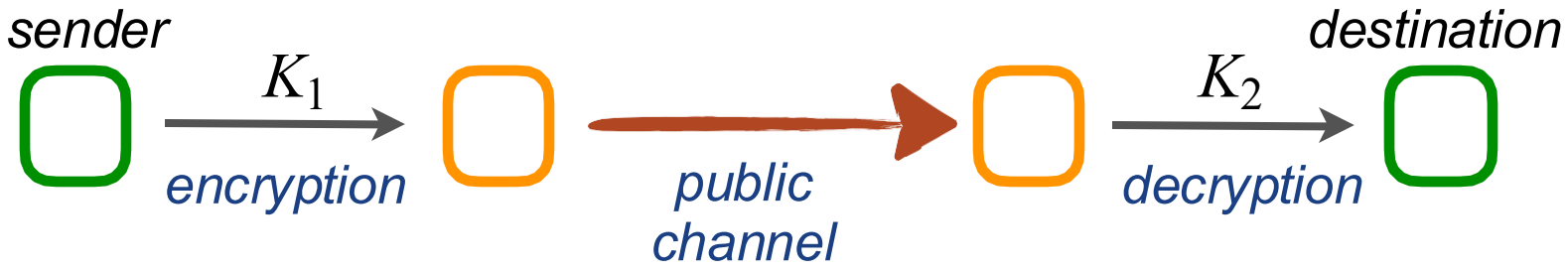


⊕ here is the XOR operator:

a⊕b =1 if precisely one of a and b is 1
else a⊕b =0

Private/public key encryption: 1 way

Asymmetric: different keys are used for coding and decoding.



Usage 1: private transmission of data

if a receiver makes K_1 public and keeps K_2 private, people can encrypt using K_1 to send information privately.

Usage 2: authentication

if a sender keeps K_1 private and makes K_2 public, people can use K_2 to decrypt and hence authenticate the sender.

Private/public key encryption: 2 way

Consider two users, Alice and Bob:

- Both users create a pair (private-public) of keys, and share their public key via an open channel.
- To encrypt, Alice first encrypts with her private key, and then with Bob's public key.
- Bob decrypts the message using Alice's public key and his private key.

This scheme achieves two goals at the same time:

- **Privacy:** only Bob can decrypt the message because only he has his private key.
- **Authentication:** the fact that the message was successfully decoded using Alice's public key authenticates that is coming from her.

Exercise

What is the run-time, in O notation of breaking a numerical password of length n by trying every possible combination?

(a) $O(n^{10})$ (b) $O(10^{\log n})$ (c) $O(10^n)$ (d) $O(n!)$

Exercise (Revision)

Consider the following function, *functionA*, which takes as input a positive integer n. Inside *functionA*, the function call *random_integer*(10) outputs a random positive integer between 1 and 10.

```
def functionA (n):
    list1 = [ ]
    i=n
    while i > 0:
        x=random_integer(10)
        list1.append(x)
        i -=1
    return list1
```

What is the asymptotic run-time of this program in O notation, for large n?

(a) O(i) (b) O(n) (c) $O(i^2)$ (d) $O(n^2)$.

Exercise (Revision)

Consider the following function, *functionA*, which takes as input a positive integer n. Inside *functionA*, the function call *random_integer*(10) outputs a random positive integer between 1 and 10.

```
def functionA (n):
    list1 = [ ]
    i=n
    while i > 0:
        x=random_integer(10)
        list1.append(x)
        i -=1
    return list1
```

What is the asymptotic run-time of this program in O notation, for large n?

(a) O(i) **(b) O(n)** (c) $O(i^2)$ (d) $O(n^2)$.

Conclusions

- **Client-server** architecture is pervasive in distributed systems. Most data science centres are accessed in that way.
- **Distributed systems** have their own ways to offer computational services, coordinate users, and guarantee robustness.
- Security is crucial in distributed system for controlling who can do what (**authentication**) and who can access what information (**privacy**).
- **Encryption** is a key technology to enable both authentication and privacy in distributed systems.