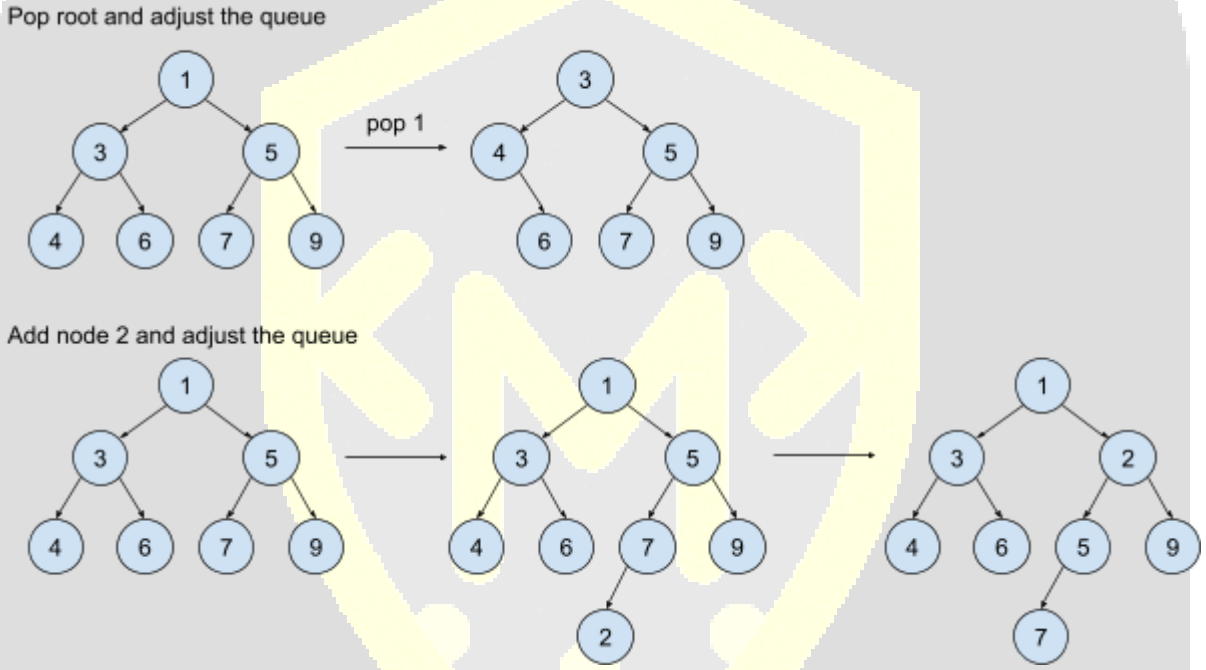
建議對演算法與數據結構已經有足夠熟悉的學員跳過此章節

主要原因是我們課程的設計核心為實際去模擬當你真實遇到該問題時會有什麼樣的想法，但講義內容其實已經有部分暗示解法，但實際遇到問題時並不會知道接下來的題目會用到解法的方向。

Priority queue

原理

Priority queue 主要是讓priority最高的元素優先出來，通常是用 heap 來實作，而 heap 又分為 min heap 及 max heap，通常會是以 binary tree 的方式呈現，在 min heap 中根的值一定是最小，且他的所有子樹也都會保持根是最小值。當我們今天要加一個新的點進到 min heap 裡，我們會先把它放在最下面（本來是 null 的位置），然後再藉由和 parent 比較大小一路調整往上位移。此時間複雜度為 $O(tree的高度)$，因此若將該 binary tree 每一次盡量填滿，tree 的高度就會是 log n，因此複雜度就會是 $O(tree的高度)$。每次從跟 pop 出最小值後需要將新的最小值移到根，一種方式為一路往下維護他的子樹（若左子樹數值比右子樹小就把左邊的點一上去然後往下維護左子樹）。Max heap 的做法跟 min heap 一樣只是根放的是最大值。



另一種實作min heap的方法

我們剛剛介紹到min heap，可以藉由把下面的點推上來實作，但這時會發現，我們很難去確定說接下來要補值要補到哪個節點，及未來重複做刪除或加入節點時會不會造成該二元樹不平衡(實際上即便不平衡，他也可以維持在O(logn)的複雜度，此處的n為總共進入過heap的資料量)，因此又另一種做法為將最後一個節點補到根的位置

最後一個節點補到根

為了簡化這個問題，實務上我們通常使用另一種做法來實作 Min Heap，並在刪除最小值時採取以下做法：

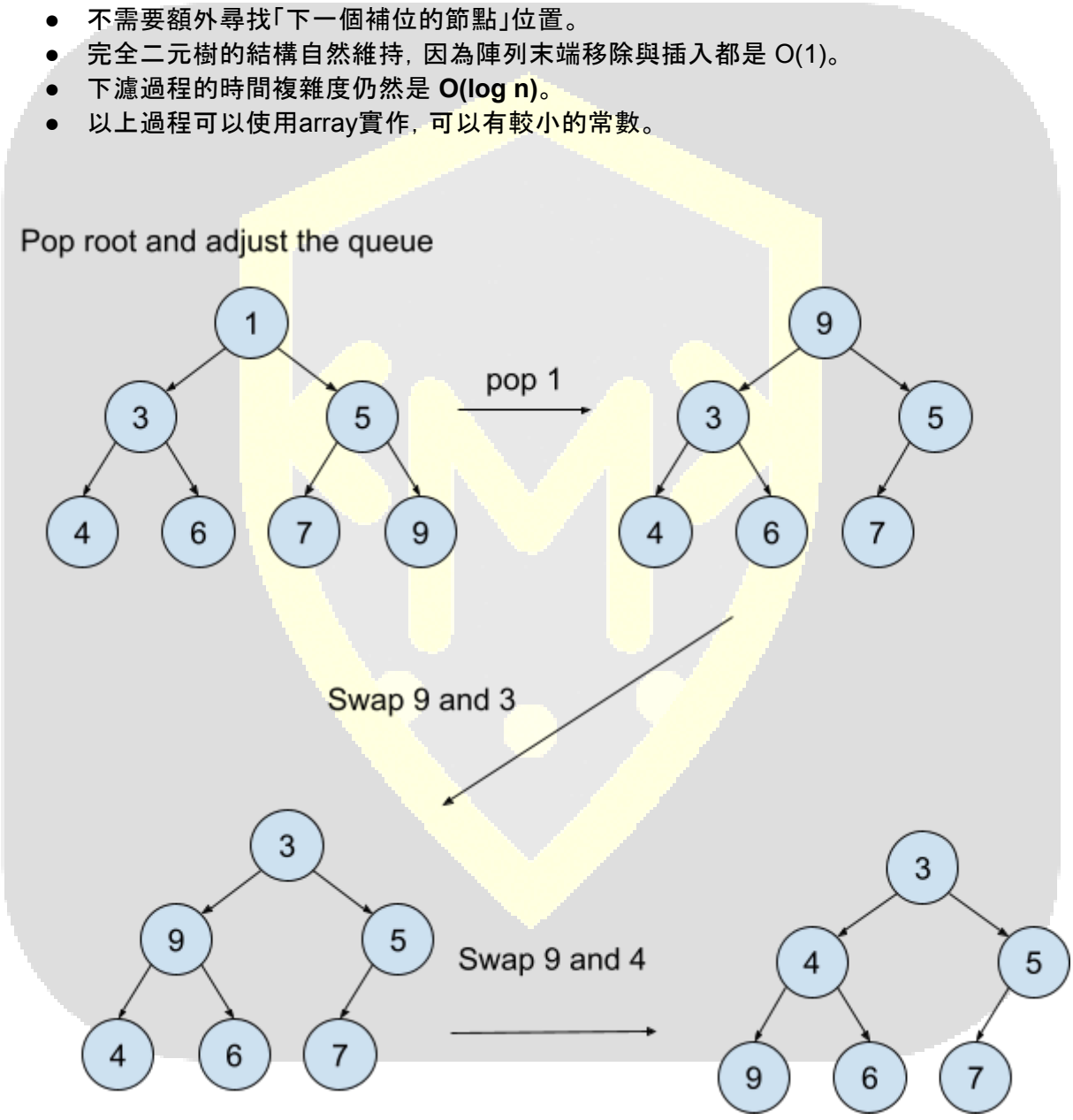
1. 取出根節點（最小值）— 這就是我們要返回的結果。

2. 將最後一個節點(陣列尾端)移到根的位置 — 這樣可以確保陣列仍然代表一棵完全二元樹，因為只是「移走尾端」再「補到開頭」。

3. 從根節點往下調整(**Heapify**) — 依序與左右子節點比較，與較小的那個交換，直到滿足 Min Heap 條件。

這個方法的好處：

- 不需要額外尋找「下一個補位的節點」位置。
- 完全二元樹的結構自然維持，因為陣列末端移除與插入都是 O(1)。
- 下濾過程的時間複雜度仍然是 **O(log n)**。
- 以上過程可以使用array實作，可以有較小的常數。



○