1-01

Priority Queue 優先佇列

DATE . . NO.

- Heap 是其中一個做出 priority Queue 最代表性的方法

每一個 priority Queue 在每個時間點提供一個答案，而非一口氣找出

與排序有關

InsertPQ(dataItem, priority)
↓
一筆一筆 push 進去

DeletePQ() or Pull
↓
取出：一次抓一個

把連續幾次抓出的動作合在一起
↳ 按priority 排序的結果

排序演算法

1-02

要新增の資料
⇩ pqInsert(): O(1)

Selection Sort : Unsorted List() ⇒ 並不是最好
⬇ pqDelete(): O(n)
由小到大排序

增の info
⇩ pqInsert(): O(n)

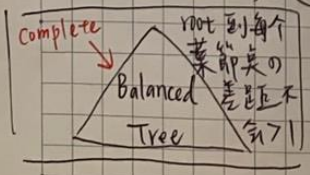
Insertion Sort : Sorted List()
⬇ pqDelete(): O(1)

增の info
⇩ pqInsert(): O(log n) 最理想 最差：O(n)

Tree Sort : BST → 可用來做排序
⬇ pqDelete(): O(log n) O(n)

增の info
⇩ pqInsert(): < O(n)

Heap → 沒有真の去做排序 但可滿足這樣の需求
⬇ pqDelete(): < O(n)



- Binary Search Tree
  worst case：樹不平衡
- Heap：保證一定是平衡の樹

增の info
⇩ pqInsert(): O(log n)
min-Heap
⇩ pqDelete(): O(log n)
只保證樹根是最小的

I-03
• Priority Queues の 応用
Nearest Neighbor? NN 問題
情況：地圖中找最小距離
↳ 先將全部分矩形区塊，在一个区中找最小距離の city，再丟回原有資料中比較，若結果最小的是 city 而不是区塊則是答案，反之則繼續找，而不用針對每个 cities 做計算.

增の info
⇩ pqInsert(): O(log n)
max-Heap
⇩ pqDelete(): O(log n)

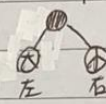
I-04
• Heap 性質（稻草堆）適合用陣列實作
① complete binary tree、balanced
② 每條路徑（每个樹葉到 root）可看成每个 sorted lists

I-05 新增 info：ReheapUp　pqInsert(): O(log n)
↳ 過程：一个 bubble 往上追溯 sort の 回合.
新增的資料位置：bottom（陣列的下一個、資料量的位置）
新增又需要檢查加入的那根稻草

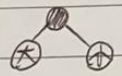
1-06

刪除 info：Reheap Down　　pqDelete()：O(log n)　worst case

- 刪樹根：constant time O(1)
  - 後↳ 放入 bottom ⇒ 不要放到維構，影響層重最小
  - ↳ 再往下調整 reheap. O(log n)

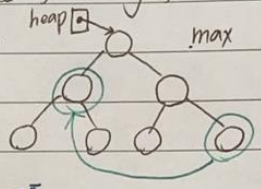
往下時：max-heap ⇒ 

if 大 > 父，swap

min-heap ⇒ 

if 父 > 小，swap

- 刪中間節點 pqDelete()：O(log n)
  - 要考慮往上

ex: 

bottom 拿過去後未必會比要刪的父節點小，就要考慮是否往上

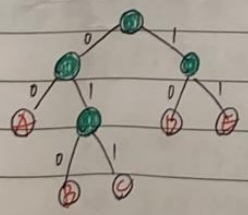
1-07 Huffman Coding 霍夫曼編碼

由樹葉建樹，最後產生樹根

1. 找最小的2個樹葉，建成一個小樹，其根的鍵值為2個樹葉相加。
2. 建出的小樹的根與剩下的樹葉，看成新的樹葉，再重複步驟1

每一回合都會產生新的節點出來，最後產出一個 二元樹

application：網路 or 影像的壓縮方法



Code

A: 00	D: 10
B: 010	E: 11
C: 011	F: 101

Code ⇒ 加or解密的金鑰

往下走時記左是0、右是1，最後用來表示符號。

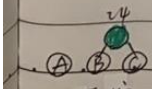
若有 D:10，則不能有 F:101 這樣的出現，因為若看到10時不能確定是D還是F，要等下一個才知道。

↳ 任何一個 code 都不能是另外一個 code 的 prefix.（前面一部分）

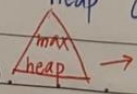
解碼時比較即時

加密花的時間 > 解密的
↳ 通常

Character A B C D E
頻率 17 5 12 7 32

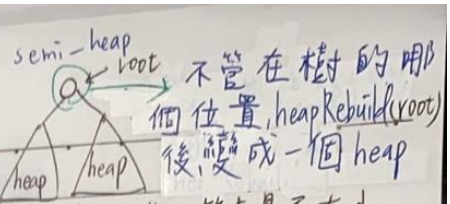


heap O(log n)



→ Delete D Insert

壓縮時把出現頻率最高的代碼設短一點更省空間(位元)



不管在樹的哪個位置，heapRebuild(root)後，變成一個 heap

1-08 新增資料於堆積

- semi-heap 樹根不對其他都對的 heap

新增的位置 bottom 就是 size()，新增後檢查與其父節點是否大小正確，父節點公式：(index-1)/2，保證最大是在樹根(小)，剩下沒有保證

1-09 刪除資料於堆積

1. 複製 bottom 到要刪的節點
2. 刪掉 (pop) bottom，變成 semi-heap
3. heapRebuild 往下把 semi-heap 變成 heap

1-10 將陣列直接轉成 heap

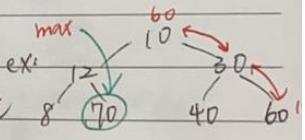
呼叫 heapRebuild(i)，i = 2、1、0 (只有3層的例子)

↳ 第一個非樹葉節點，且是在下角 last internal node

2 把右子樹變 heap，由下往上

1 把左 2 2

↳ 不能從 0、1、2 做：∵不能保證 Max or Min 會到 root



1-11 以指標實作堆積結構

左小孩、右小孩、父節點 指標

insert：在邊較複雜：∵parent OF Bottom 要移動

POB 往上找若有 右邊有兄弟則 POB 變為其右兄弟

↳ 沒有則表在最左邊

1-12 堆積排序

heap 能最大 or 最小，∵ 問 n次最大/最小 ⇒ 排序結果

↳ 每次刪除 log n ⇒ O(n * log n)

- 優點：實作上可寫在同一個陣列裡運作

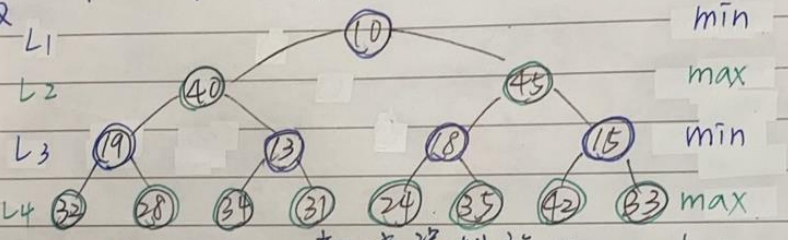
刪除時 Max/Min 跟 bottom swap，再把 size -1，就不會用到本要刪除的

2-01 堆積變形 Variations of Heap

- Double-ended Priority Queues (DEPQ)
  - Min-Max Heap
  - Double-ended Heap (DEAP)
- Forest (union) of Heaps ⇒一群Heaps合在一起成一個Heap
  - Binomial Heap
  - Fibonacci Heap

Min-max Heap —complete Binary Tree

- DEPQ



2-02 Min-max Heap：Insert 新增資料於 Min-max heap

①看加入的是單數層(min層) or 偶數層(max層)

②看是否要和其父節點 swap

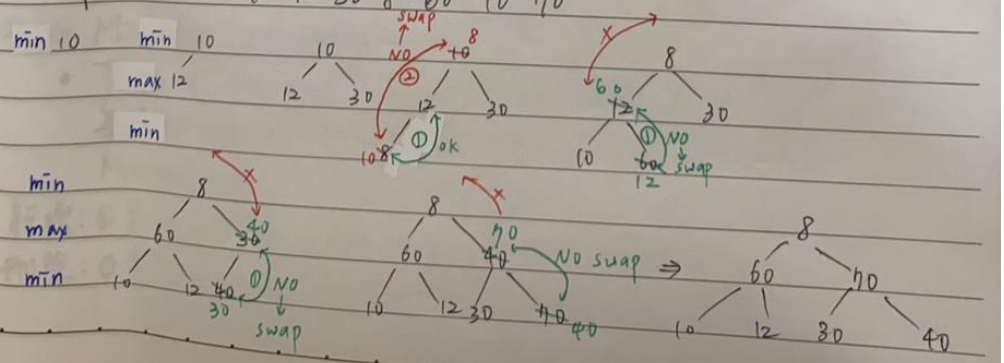
③在其所在的奇(偶)數層往上 reheap

↳與其祖父節點.

公式：{[(index-1)/2]-1}/2

2-03 Insert data into Min-max Heap practice

- input order：10 12 30 8 60 40 70



index	0	1	2	3	4	5	6
value	8	60	70	10	12	30	40

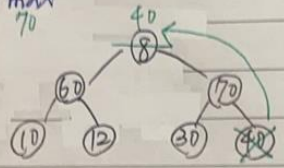
2-04 Min-max heap: Delete the Smallest

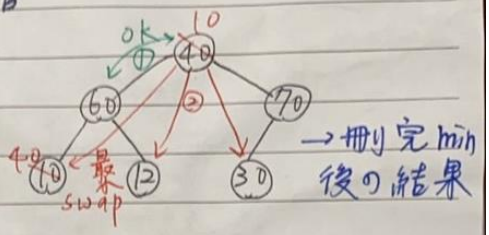
- 刪最小的
  1. 刪 bottom 取代最小的
  2. 看和其小孩的大小是否正確要 swap?
  3. 不管 step 2. 是否有 swap，只要檢查刪掉的 heap (即 min heap) 是否要 swap
  只有一條路徑 ⇒ $O(\log n)$

2-05 Min-max heap 的刪除練習

- 刪 min & max

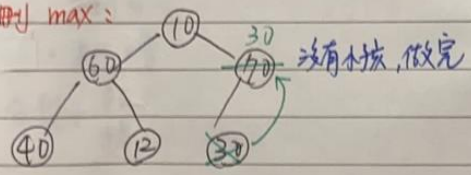
刪 min：





→ 刪完 min 後的結果

刪 max：



- 如何判斷在 min or max 層：取 log2 取整數
  公式：level = ((int)floor(log2(i+1)) % 2)
- 找祖父節點
  if (i-1)/2 > 2) grandparent = (i-3)/4

2-06 Min-max heap 總結

- Three 4-way trees 有3棵樹所組成，存在一起
  每個max heap裡的節點，都有個 min heap 的父節點
  Min-max heap = 1 min-heap + 2 max-heap

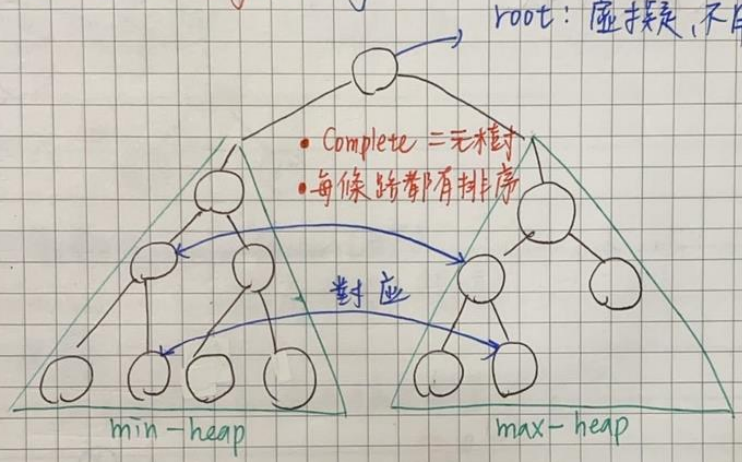
新增：① 先跟其父檢查回再和祖父 heap 檢查

刪除：① 先檢查父節點 ② 看是往下移動 ⇒ 與孫子比較

DATE . . NO.

2-07 新增資料於雙向堆積

- Double-ended Priority Queue (DEPQ)
  - Insert any key
  - Delete the smallest key
  - Delete the largest key



- DEAP: Insert

1. 檢查兩个 Heap 的連接關係 Left < Right
   not 父子，而是左右關係
2. 往上檢查 reheap

2-08 DEAP: Delete the Smallest

1. 用 bottom 取代 min
2. 往下 reheap ∵ bottom是大的值
3. 檢查對應節点 ⇒ Left < right
4. 若左右 swap，再看是否往上 reheap

- 特殊情況：左樹比右樹大、多一層
  新增入右樹後，不能只檢查對應節点，若左樹對應結点下還有一層，要 check

DATE NO.

2-09 DEAP Summary

- 空的根 + min-heap + max-heap
- max-heap 裡的每的節点都有一個對應節点在 min-heap.
  當 min-heap 節点沒有對應 max-heap 節点可比較，則和對應 Index 的父節点比較.

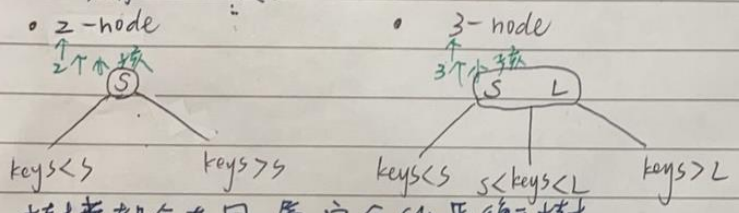
Unit 3.

heap 不適合用來搜尋. binary search 較佳

- 平衡二元樹
  - 2-3 tree
  - 2-3-4 tree
  - AVL tree
  - Red-black tree

  (2-3 tree, 2-3-4 tree 類似; AVL tree, Red-black tree 類似)

- 2-3 樹
  - 只有 2-node 及 3-node



樹葉都會在同一層, 完全的平衡樹

- 新增:

分裂: 加入有 2 個 key 的 node 時, 由小到大排序, 中間值往上提, 小的及大的分別是中間值的 2 個小孩 (分裂)

- 樹高 2-3 樹高最差情況: 每個 node 都只有一個 key.
- 刪除:

① 先重新分配 ② 再合併.

↳不會破壞結構

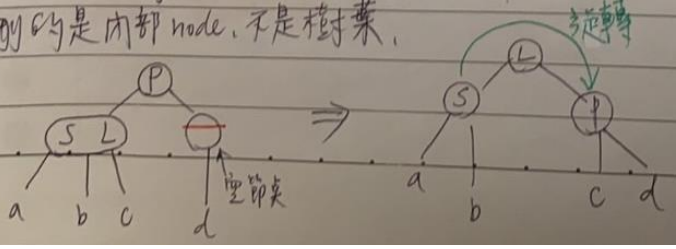
當刪完所有 keys, 造成 node 空掉時

兄弟的 2 個 keys 及父節點

① 先找大小最近的兄弟, 找有 2 個 keys 的 與其旋轉, 中間值 偏父節點, 小在左, 大在右.

② 若無有 2 個 keys 的兄弟, 合併其父節點及其兄弟到其兄弟

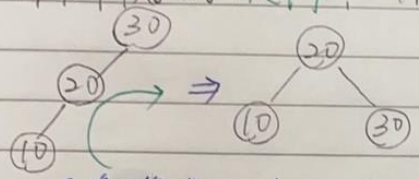
若刪的是內部 node, 不是樹葉.



Unit 4

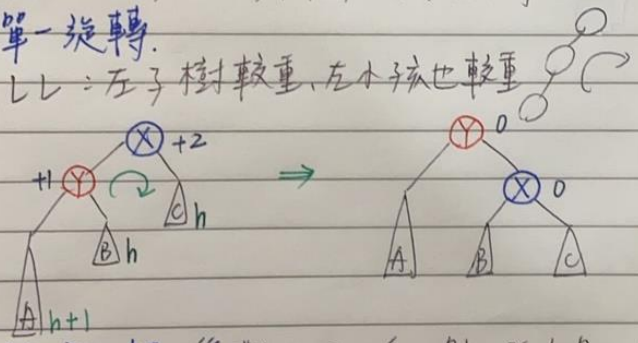
- **AVL 樹** 平衡的 BST
  是 BST 中高度最矮的，不是傾斜的

1. 每次 insert or delete 後，check 是否 balanced
   平衡：任何一節点の左子樹與右子樹相差≤1（高度）
2. 不平衡則 **旋轉**：中間值上提，小放左，大放右



- **平衡係數 Balance Factor (BF)**
  BF (a node) = 左子樹高 − 右子樹高

- **單一旋轉**
  LL：左子樹較重，左小孫也較重



- **複式旋轉**：係數一正一負 RL or LR