優先佇列簡介 (1-1)

* priority + queue 在 queue 中有優先順序

例：可用於病人治療順序(緊急程度、到達時間等)

* 進/出 queue 只有一筆資料

以排序完成 priority queue (1-2)

* 選擇排序：Best case = Worst case = Average case = $O(n^2)$
(selection sort)

* 插入排序：Best case = $O(n)$, Worst case = $O(n^2)$, Average case = $O(n^2)$
(Insertion sort)

* BST：存入，取出時最差為 $O(\log n)$，但有例外 (傾斜樹)
應用於近鄰搜尋 (1-3) (nearest neighbor)

* 進行區域劃分可減少非必要的計算

* 針對特定區域進行分析

初探堆積結構 (1-4)

Heap => min-heap , max-heap

這兩種功能保證 root 為最小/大項，並不比較 left/right child

Insert : $O(\log n)$, Delete $O(1)$

新增資料於堆積結構 (1-5)

* ReheapUp() : 於 bottom 新增資料，再沿走訪路徑進行比較

Insert = $O(\log n)$.

刪除資料於堆積結構 (1-6)

* 刪除資料後需進行維護

* ReheapDown : 將 bottom 補上 root，並再判斷左右子孫

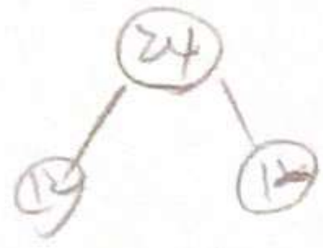
* 刪除中間結點，除了 ReheapDown，並需 ReheapUp

程大是編碼(1-7)

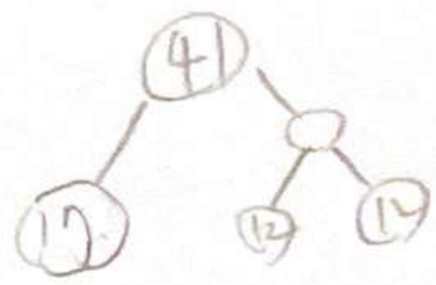
從樹葉開始建一棵樹，not BST

ex. 17, 12, 14, 24, 32

先找兩個最小相加，建一棵樹

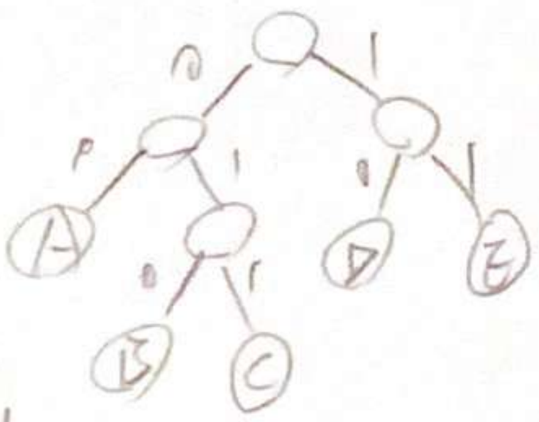


再找最小的加入，並記錄走訪路線(左0右1)



⋮

例：生成以下形狀



A則為(00)，B(010)，C(011)，依此類推

期末資料精確，不能出現有編碼出現在另一編碼之前

新增資料於堆積的範例(1-8)

* heapInsert(), heapDelete(), heapRebuild()

ex. max-heap

→

刪除資料於堆積的範例(1-9)

將 bottom 補上樹根，再 heapRebuild 底建 semiheap，再比較左右 child

將陣列轉堆積 (1-10).

* 依序加入資料並維護，將存在陣列的資料轉成 heap.
* 由最底層子樹開始往上迴圈 / 遞迴

以指標實作堆積結構 (1-11)

Node 除 Left Child & Right Child 指標外，亦可新增 parent 指標方便查

* parent of bottom：一節點 Left Child & Right Child 未填滿，當填滿時
  可利用 parent 往上找
* 記錄
  - 1. 用指標記錄
  - 2. 利用二進位

堆積排序 (1-12)

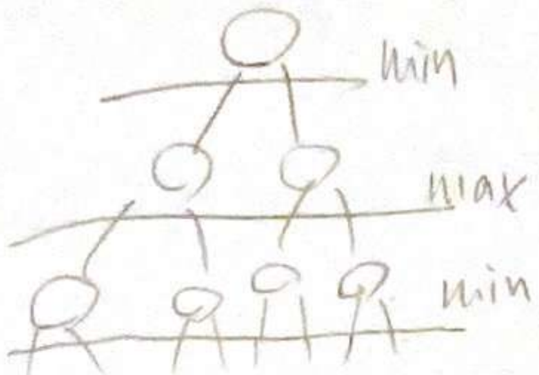
* O(n log n)
* 將一陣列進行 heap 排序可節省空間
* 所需時間與 Merge Sort, Quick Sort 差不多.

堆積變形初探 (2-1)

* 例：當大量飛機起降先後順序

* Doubled-ended Priority Queue — Min-Max heap

Min-Max heap.:



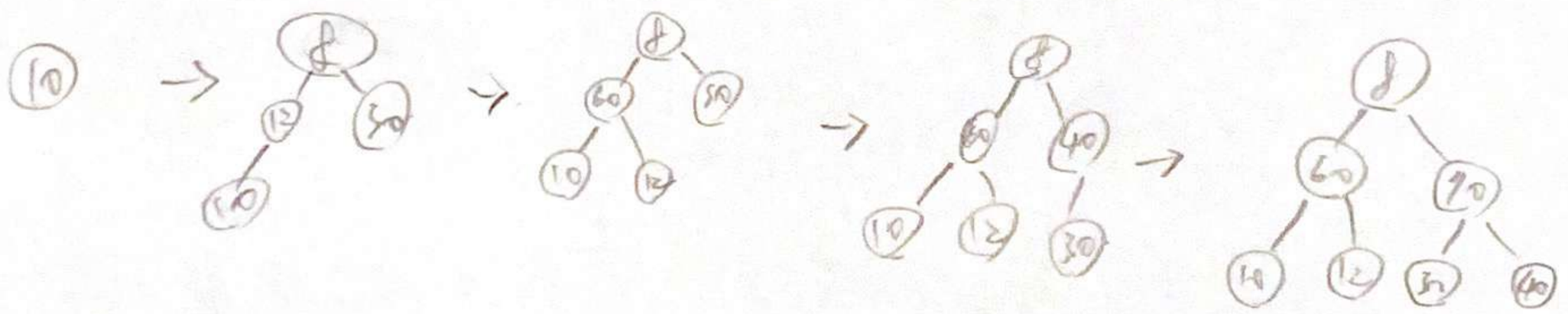
root 為最小值

新增資料於 Min-Max heap. (2-2)

* 先判斷層數為 min / max 層，再與 parent & grandparent 檢查做交換

新增資料於 Min-Max heap. 的練習 (2-3)

ex. 10, 12, 30, 8, 60, 40, 70



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

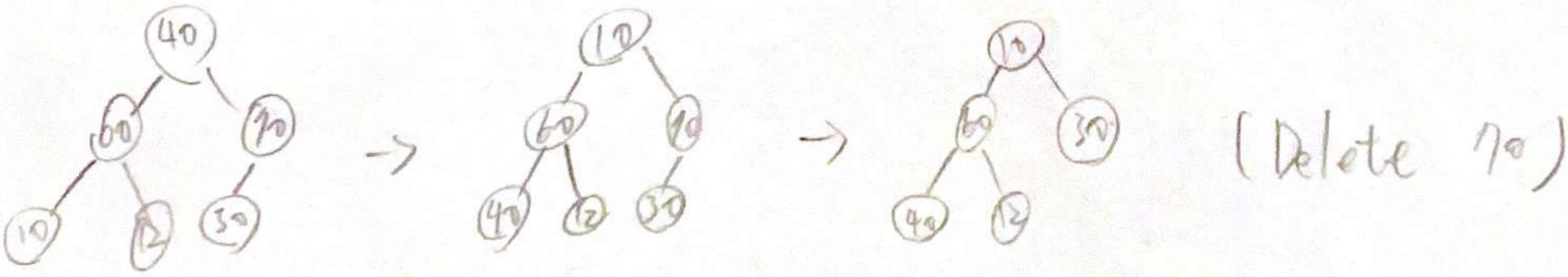
刪除資料於 Min-Max heap. (2-4)

* 刪 root，由 bottom 補上，調整方法同"插入"

* 先檢查 child，再看 min / max 層

但若停在倒數第二層，要再往下檢查

於 Min-Max heap 刪除資料的練習 (2-5).

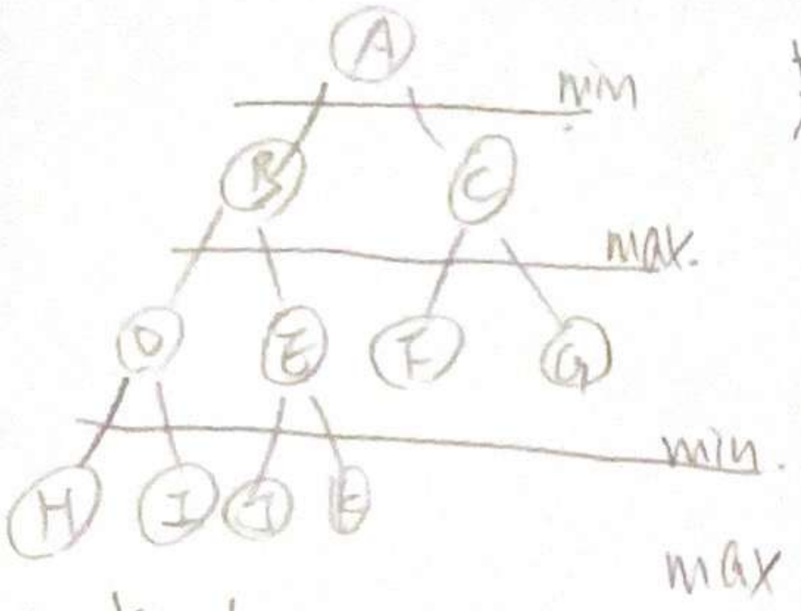


(Delete 70).

求點 i 之 grandchild node => $4i+j$, $j=3,4,5,6$

Min-Max heap 總結 (2-6)

ex, Min-Max

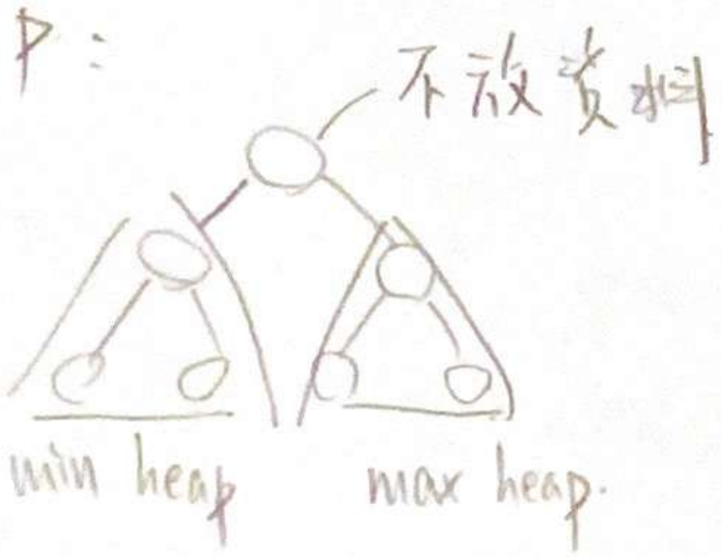


其包含了特性，每一max層節點都有parent

若為Max-Min則相反

新增資料於雙向堆積 (2-7)

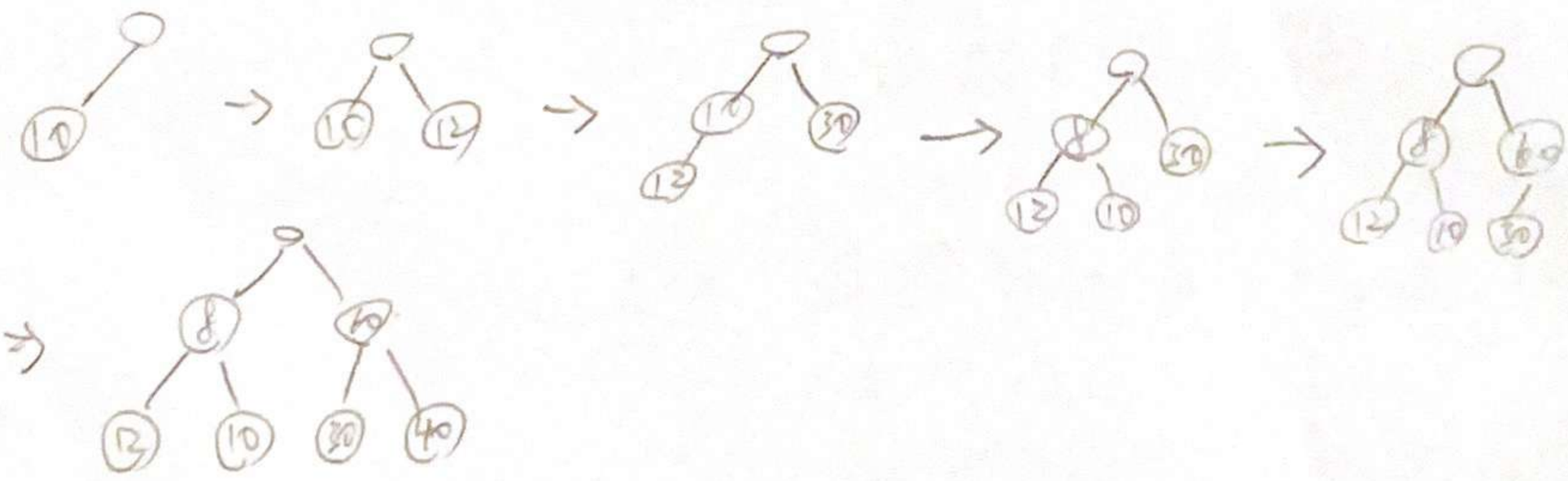
DEAP:



不同於 min-max heap

有兩個獨立的 heap，需找對應點.

ex, 10, 12, 30, 8, 60, 40



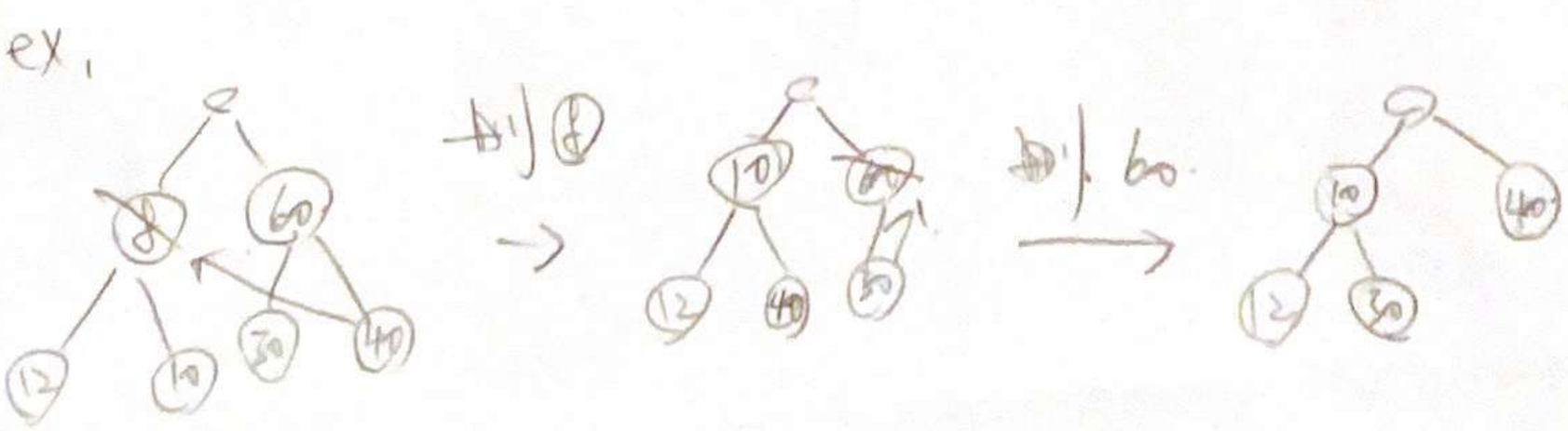
刪除 DEAP 中資料 (2-8)

*刪除 min，用 bottom 遞補，找其子節點，比他小且最小者交換，依此數據，再回插入時的檢查操作.

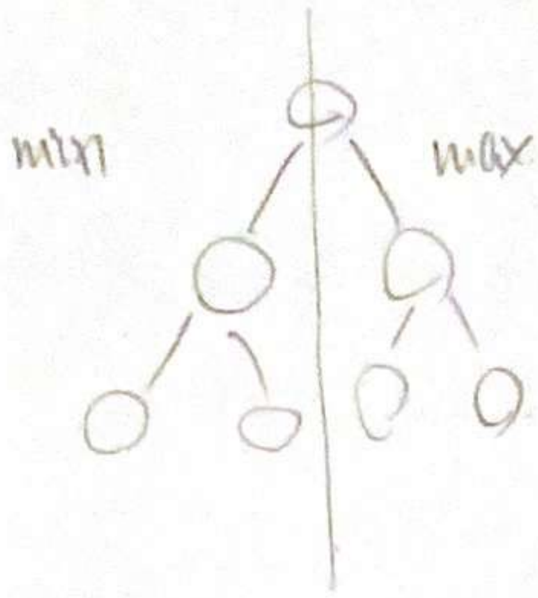
*刪除 max，先往下再往左比較

特例：若往下時走到葉則往左比較，亦需比較同層之child

ex,



DEAP 總結 (2-9)



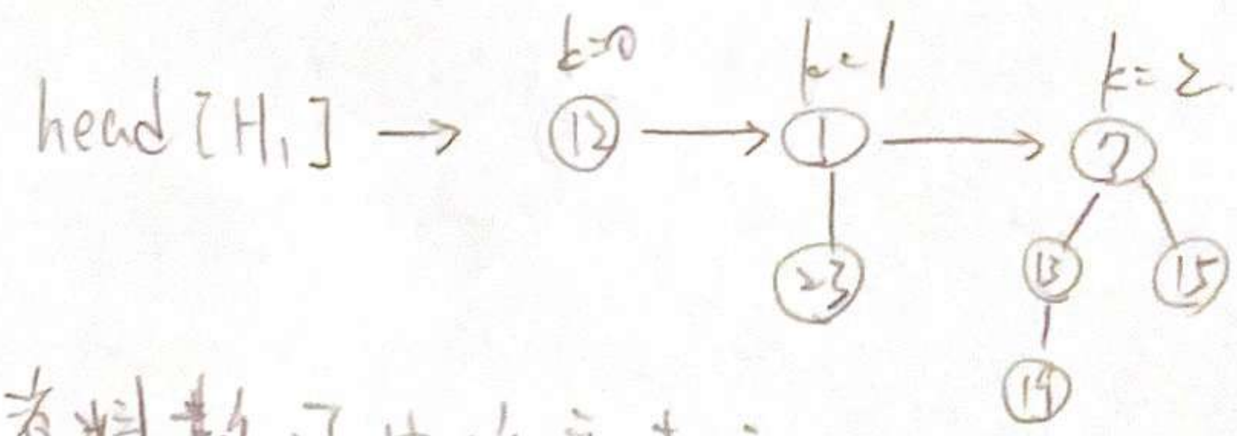
2 棵樹，節點間有對應關係
若往下沒走到 leaf，不用檢查 pair.

堆積結構的應用 (2-10)

* 處理一堆資料 (如：網頁) 時
  External Sort ⇒ (quick + heap) sort.
  先將部分資料建成 DEAP / Min-Max heap.
  其餘利用 quick 放置，直到一筆資料小於記憶體，即可行任意排序
* 當要合併 2 個 Priority Queue 亦用 heap.
* 平衡 2 個人數不平均的隊伍

可合併的堆積結構 (2-11)

* Binomial heap = k 表 child 個數 $O(\log n)$.

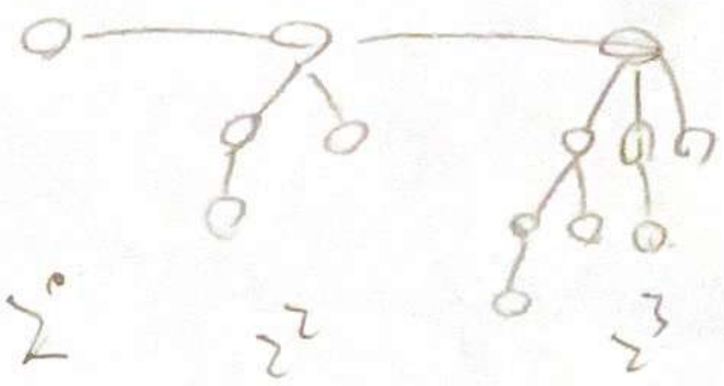


資料數可由次方表示

$7 = 2^0 + 2^1 + 2^2$, $9 = 2^0 + 2^3$

* 根據 k 排序

ex. 13 筆：$13 = 2^0 + 2^2 + 2^3$ ⇒ 3 棵樹.



二項式堆積 (2-12)

* 2相同者則合併，若三相同者，取2較小者合併

堆積結構的總結 (2-13)

* Heap：

  1. Complete tree 2. 路徑上是排序

  Insert(), Delete(), ReheapUp(), ReheapDown()

  O(log n)

  另一次加入 / 一半一半加入

* Doubled-ended Priority Queue:

  Min-Max heap

  DEAP } 相似觀念

* Binomial v.s Fibonacci

搜尋的基本款念 (3-1)

* heap不能用於搜尋
* 例：若有一多國首都人口數表格，可根據越特要求(key)搜尋所求

搜尋機制的基本要素 (3-2)

* 將資料放 class Table ⇒ 包含 Insert, Retrieve... (新增, 刪除, 搜尋)
* 當資料重複時 (duplicate) 需考慮處理方式

根據題目，會有不同的排序要求，可先對資料進行處理，以利
* (是否排序) × (陣列 or pointer) ⇒ 4種

二元搜尋：符合要求建立，對原資料內容，但若改變越特，則需
建立資料結構。

安作方式的不同選擇 (3-3)

* 不同的方法之效率決定應用的時機
* 若處理一個未排之陣列

新增：放最右邊，記錄最後一筆的位置 O(1)

刪除：刪掉後，剩餘的要往前補滿 O(n)
* 若處理一個有排序之陣列

新增及刪除都要進行資料搬動 O(n).
* 二元搜尋已排序的資料：O(log n)
* 用指標且未排序

新增：放在第一個位置，不用搬動資料 O(1)

刪除及搜尋：無排序 O(n)
* 用指標且排序

新增，刪除，搜尋：O(n)

	插入	刪除	搜尋	走訪
未排陣列	O(1)	O(n)	O(n)	O(n)
已排陣列	O(n)	O(n)	O(log n)	O(n)
未排指標	O(1)	O(n)	O(n)	O(n)
已排指標	O(n)	O(n)	O(n)	O(n)
二元樹	O(log n)	O(log n)	O(log n)	O(n)

初探平衡二元樹(3-4)

* 2-3 tree, 2-3-4 tree, AVL tree, Red-Black tree
* 2-3 tree：子結點不是2個就是3個(樹葉往上生長)，能保證樹高會小於同筆資料建成之BST的樹高

簡介2-3 tree (3-5)

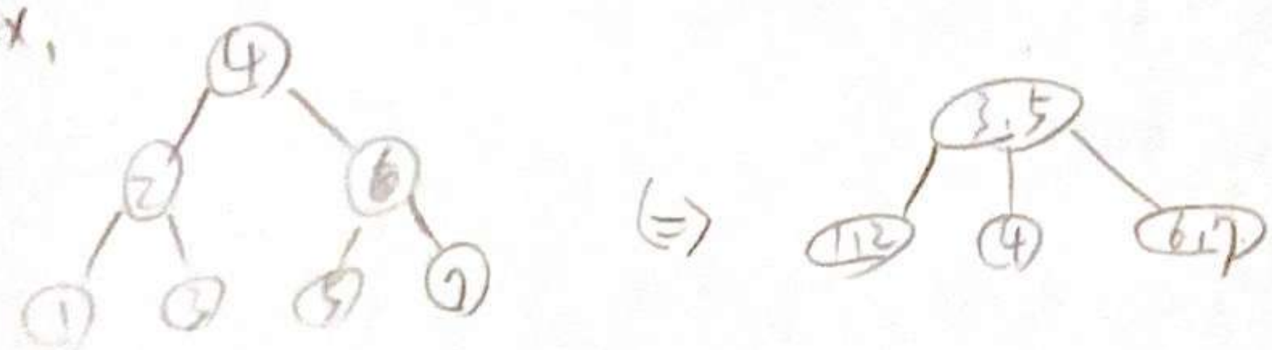
* key數主為子結點數主-1，左小右大
* 若結點存放已滿，則放入後排序，並拋出中間者，往上新增root

ex,

樹的樹高 (3-6)

* tree height會因順序而有所差異

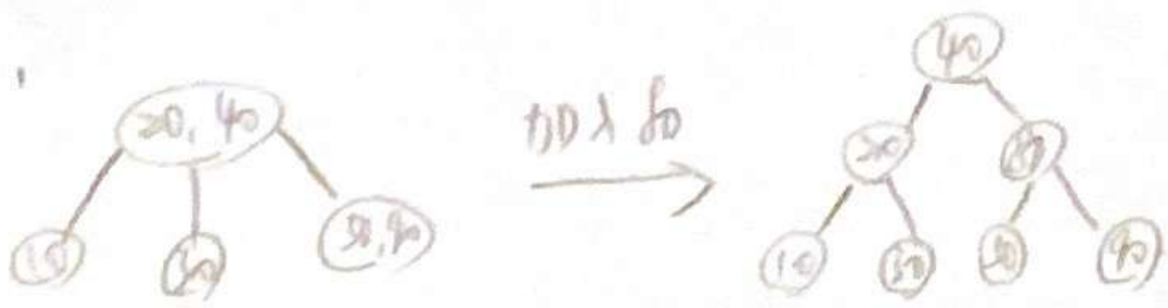
ex,



新增資料於2-3樹的演算法 (3-7)

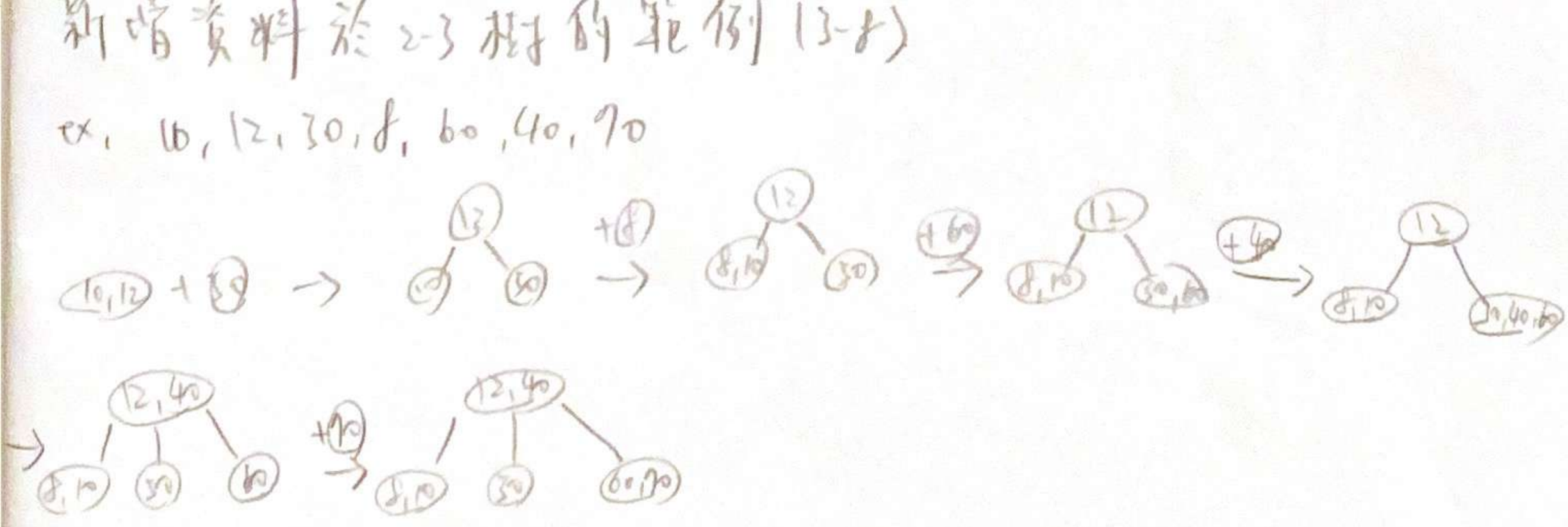
* 找到leaf，從root依key走到NULL
* 加入資料，若空間已滿 → split

ex,



新增資料於2-3樹的範例 (3-8)

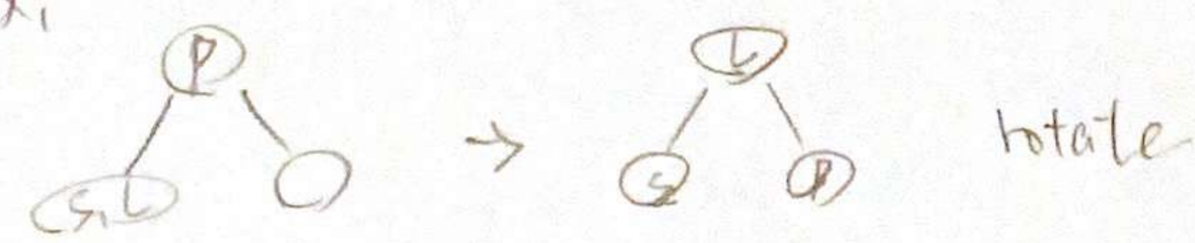
ex, 16, 12, 30, 8, 60, 40, 70

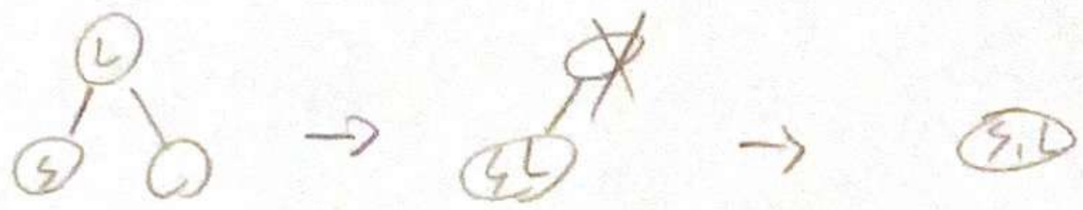


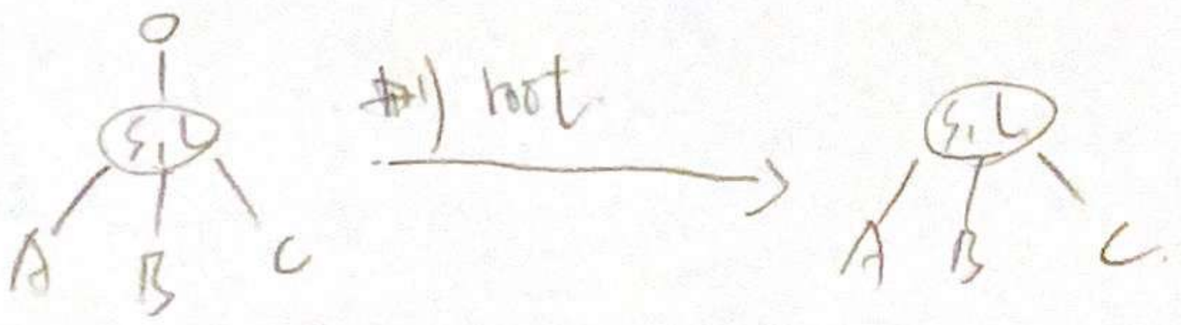
2-3 tree 刪除資料的基本概念.(3-9)

* Redistribute values + Merge (合併)

ex.



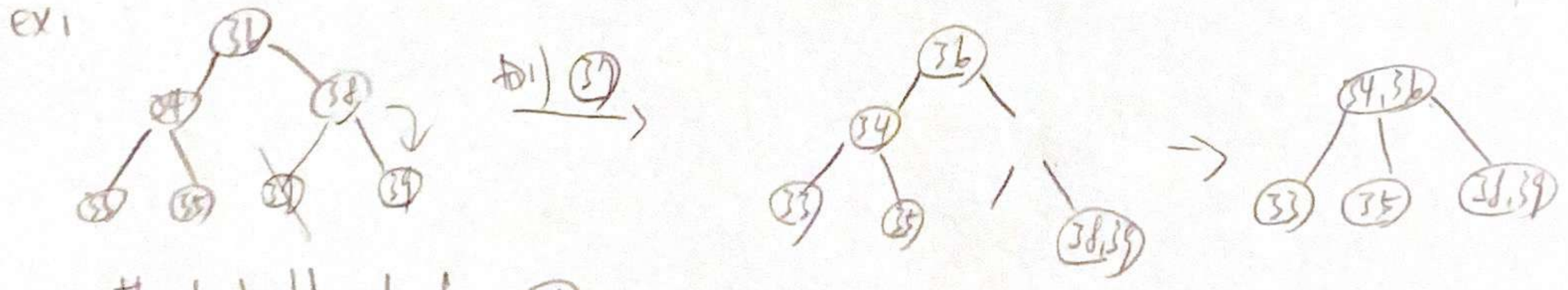




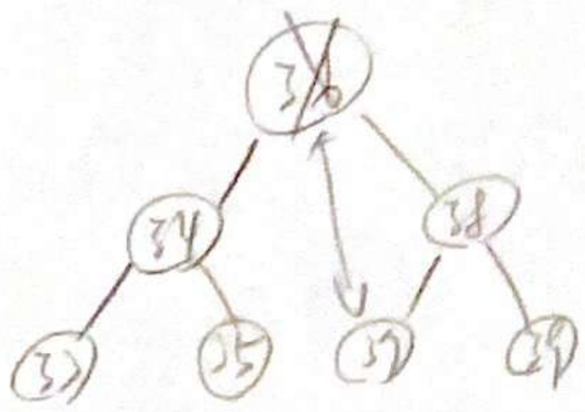
於 2-3 tree 刪除資料的演算法 (3-10)

* 步驟：找到樹葉再刪除的對象 → 刪除
  判斷是否 fix 了 Yes, 重新分配 -> 合併
* 重新分配 & 合併方法判斷 ⇒ 先弟是否有多的資料
  Yes, Redistribute.

ex.



ex. 若刪非 leaf, 36



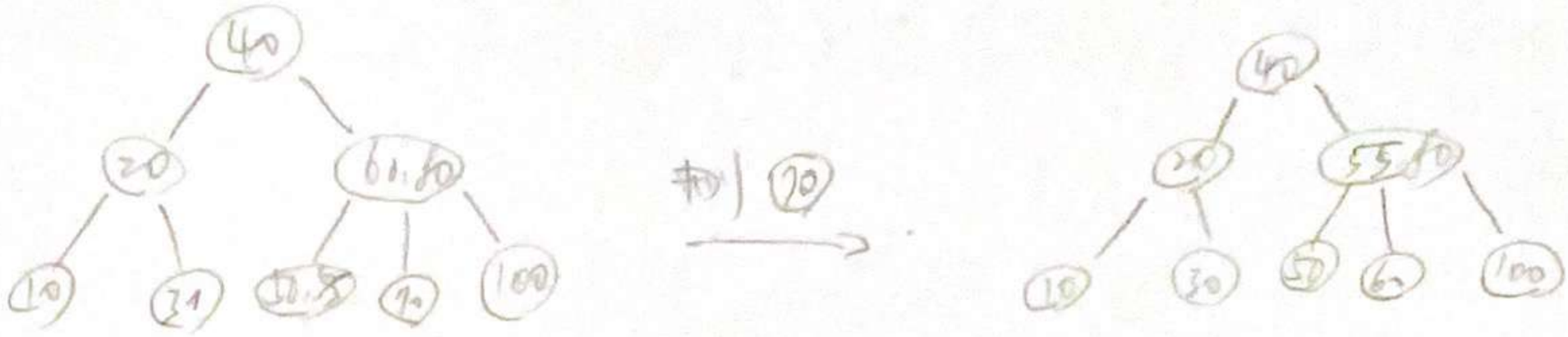
36, 37 swap.

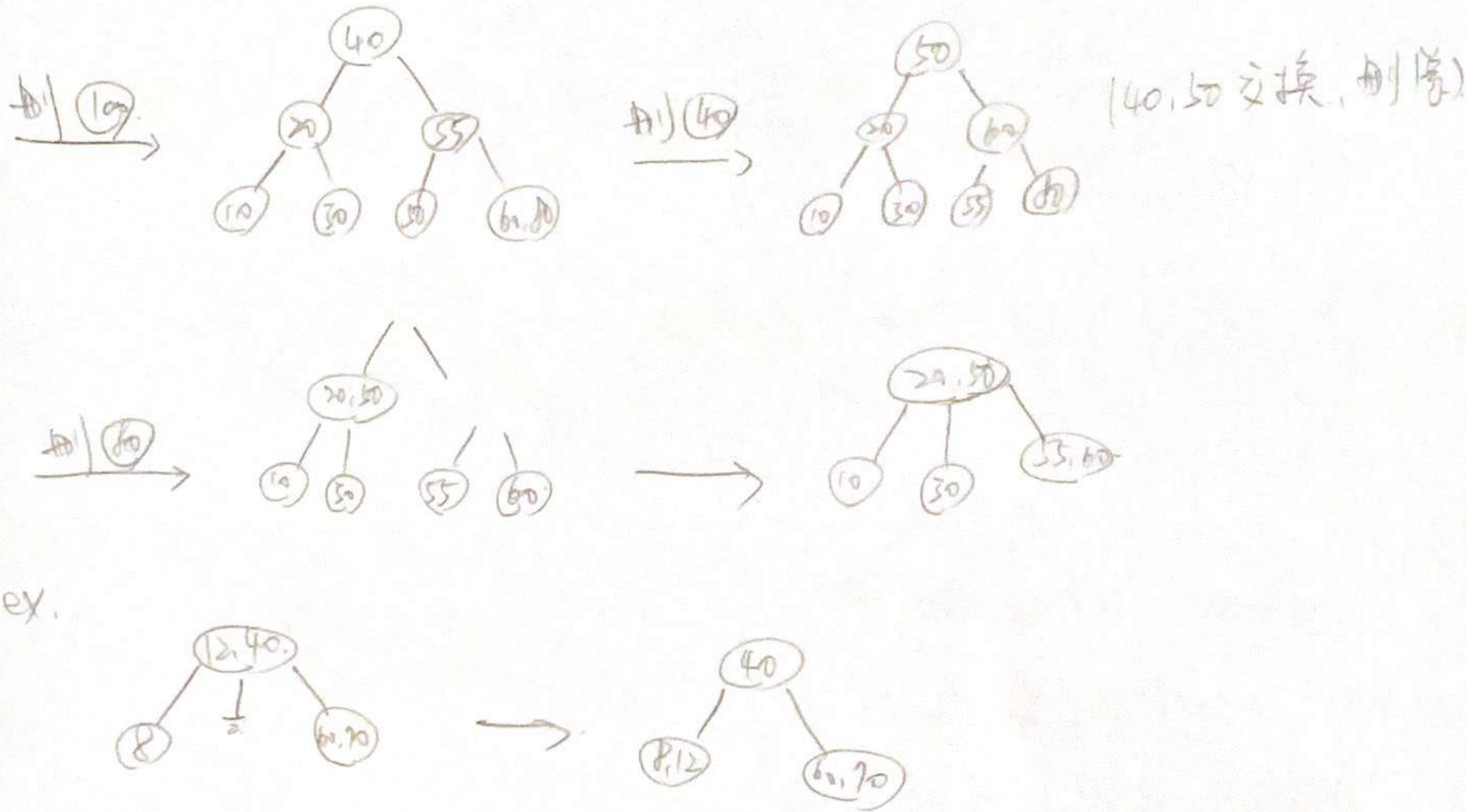
fix: 檢查先弟 ⇒ 有 no 則 rotate

合併: root 往下併至 leaf.

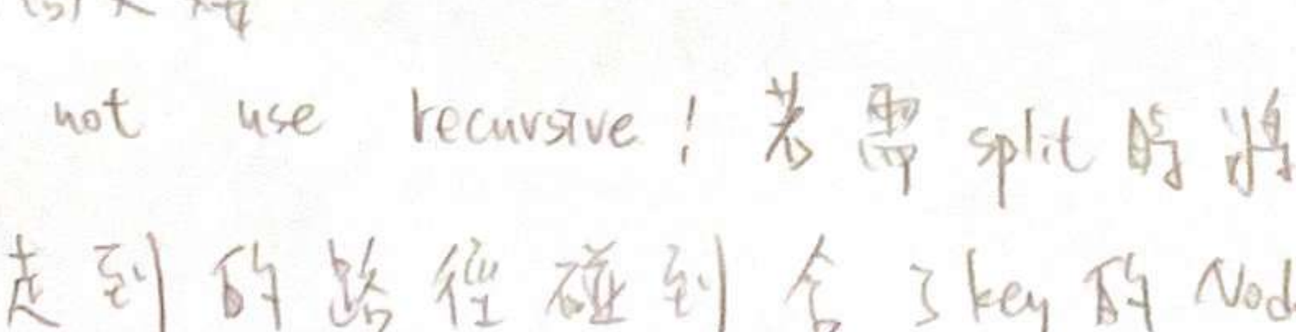
於 2-3 tree 刪除資料的範例 (3-11)

ex.





ex.



2-3 tree 的優點：① 樹高較矮 ② 保持平衡

搜尋時間固定

於 2-3-4 樹新增資料 (3-12).

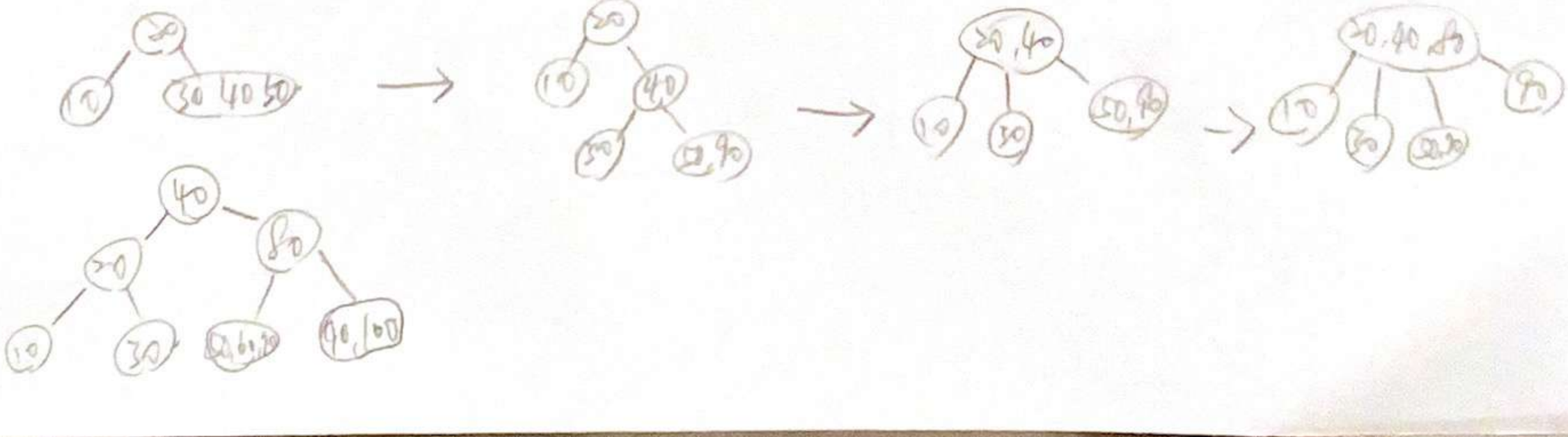
* 相似於 2-3 tree，3 key 4 pointer

⇒ 樹高更矮

* Do not use recursive! 若需 split 的將不知道哪個要往上搬

當走到的路徑碰到含 3 key 的 Node 就先分裂

ex, 60, 20, 30, 40, 50, 90, 80, 70, 60, 100

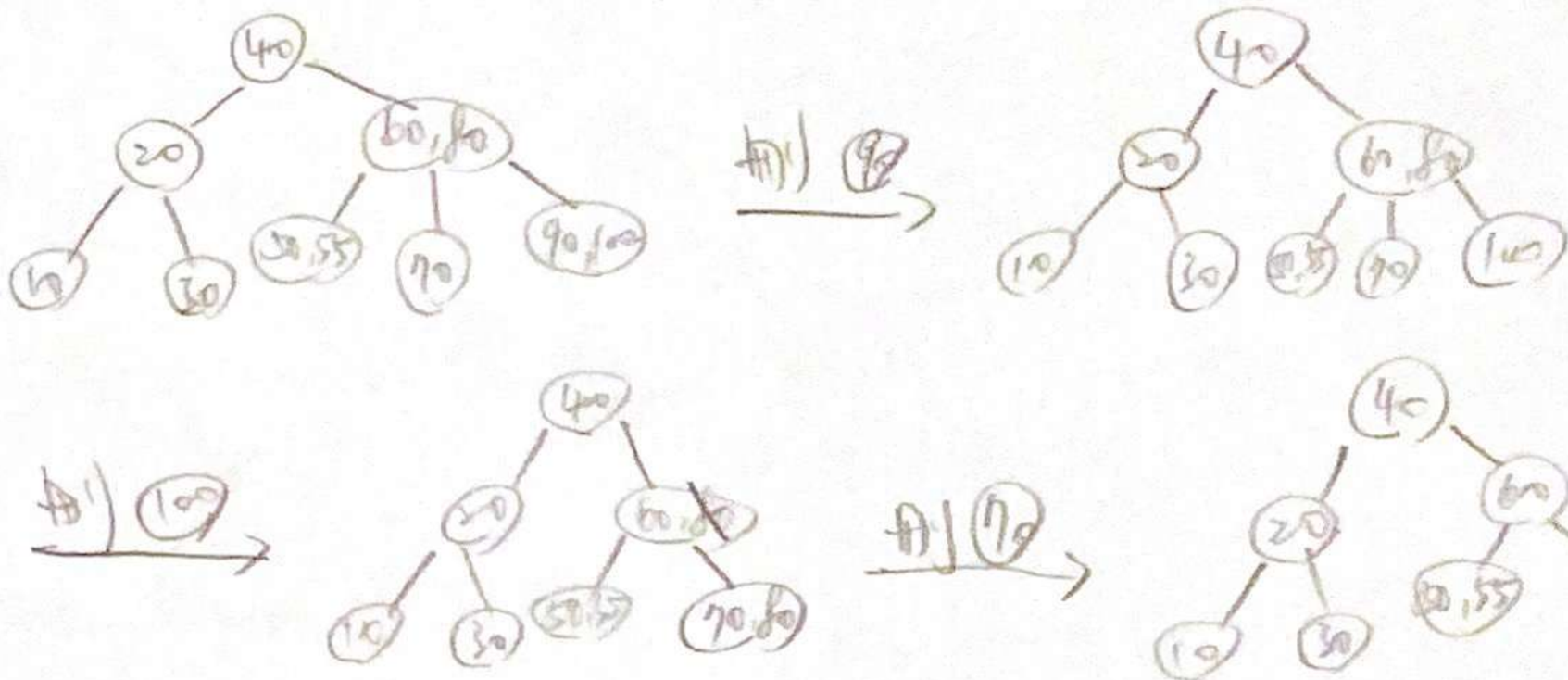


於 2-3-4 樹刪除資料 (3-13)

* 而為"多管閒事"的邏輯，搜尋時檢查將大有1個key的節點合併

* 2-3, 2-3-4 tree：樹高，效率高，但浪費時間

ex.



2-3 樹，2-3-4 樹的缺點 (3-14)

* 不同概念可分為不同家族：

  BST 由上往下

  2-3 & 2-3-4 tree：由下而上建立，且會平衡，利用空間換取時間

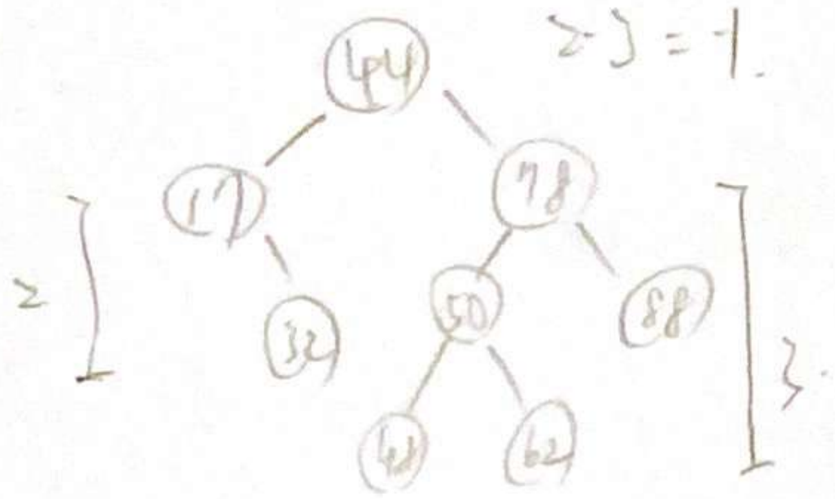
AVL樹 原理 (4-1)

* 平衡的二元搜尋樹，能做到樹高最矮
* 平衡：每個節點的左右子樹樹高差 ≤ 1

若不平衡：rotate

AVL樹的平衡係數 (4-2)

ex.

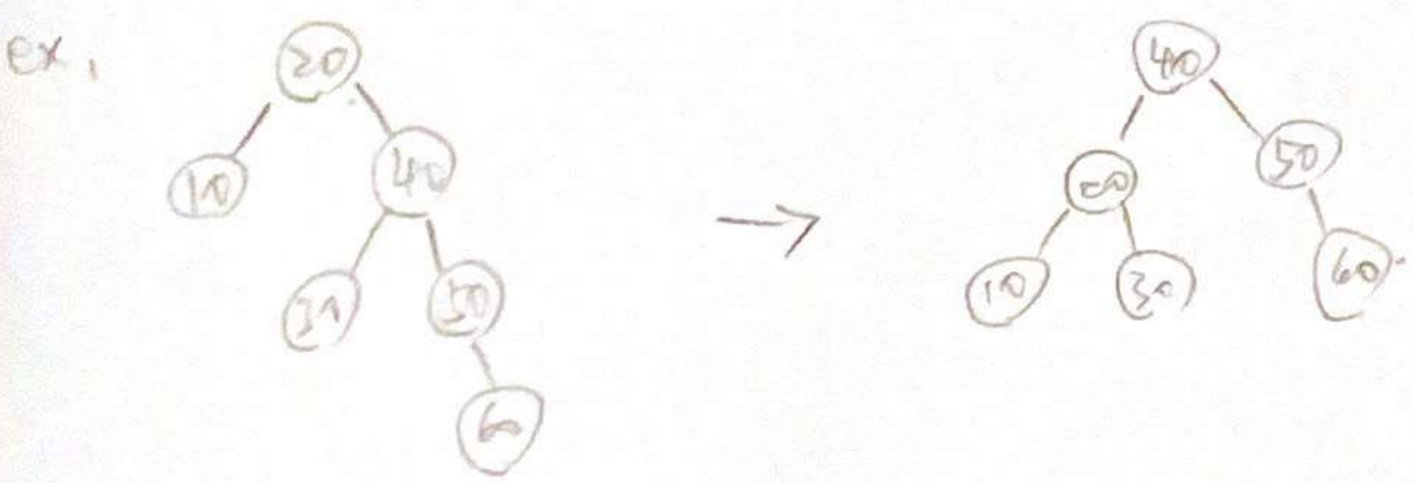


每新增/刪除，都需維護

AVL樹的旋轉 (4-3)

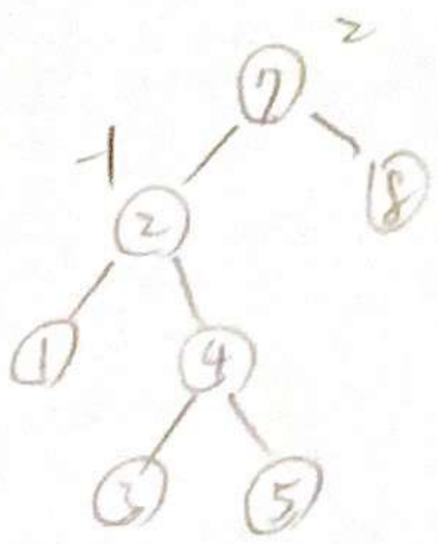
* 單一旋轉 (single)

ex.



* Double rotation

ex.

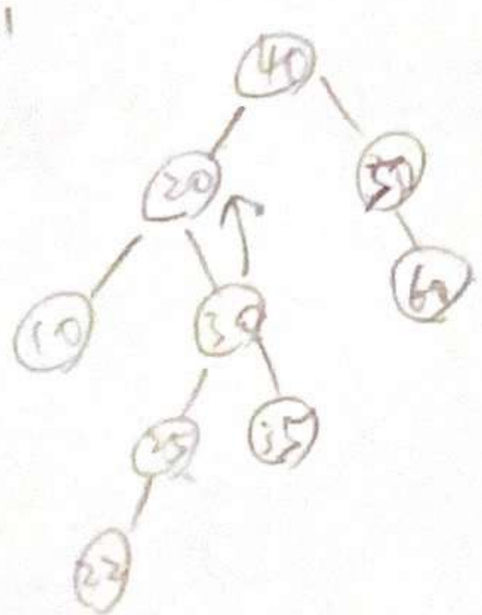


左邊比將重，則 check leftChild 是否同號

是則表示偏重同側，即可做 single rotate

else => double

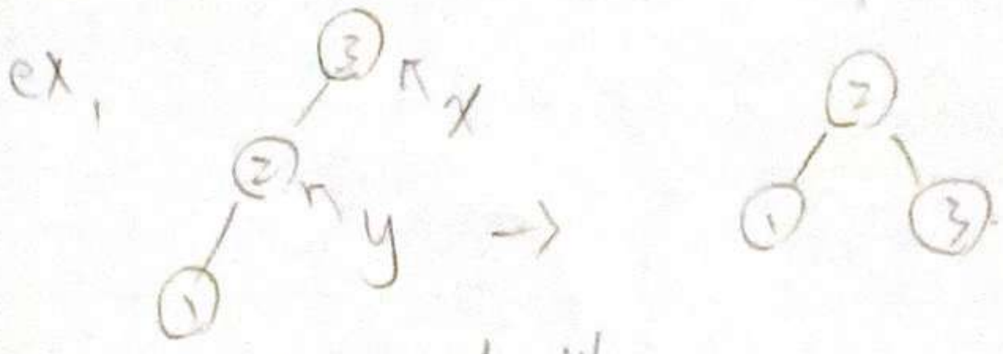
ex.



AVL tree 的旋轉 (4-4)

* 新增 rotate，從 root 往下放資料，並計算左右子樹高的差

ex.



AVL tree 複式旋轉 (4-5)

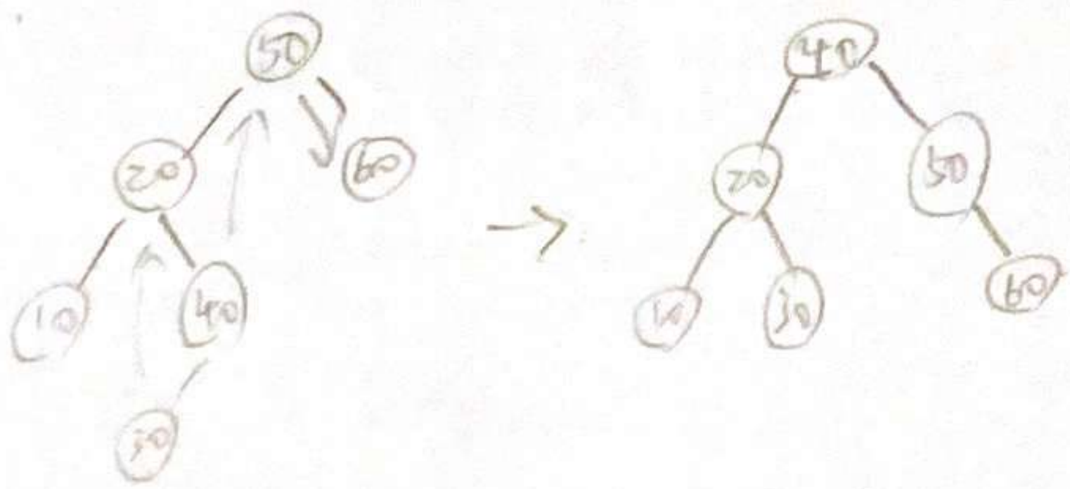
* LR：若子現象子樹分別的差值正負相反，則呼叫 LR，RL 以 double rotate 處理

* 統之將三個節點中的中間位往上放

AVL tree 複式旋轉 (4-6).

LR: x->left = RR(x->left);
return LL(x);

ex.



刪除 AVL tree 的資料 (4-7)

* 刪除方法同 2-3 樹. (swap -> Redistribute)，可能需再旋轉

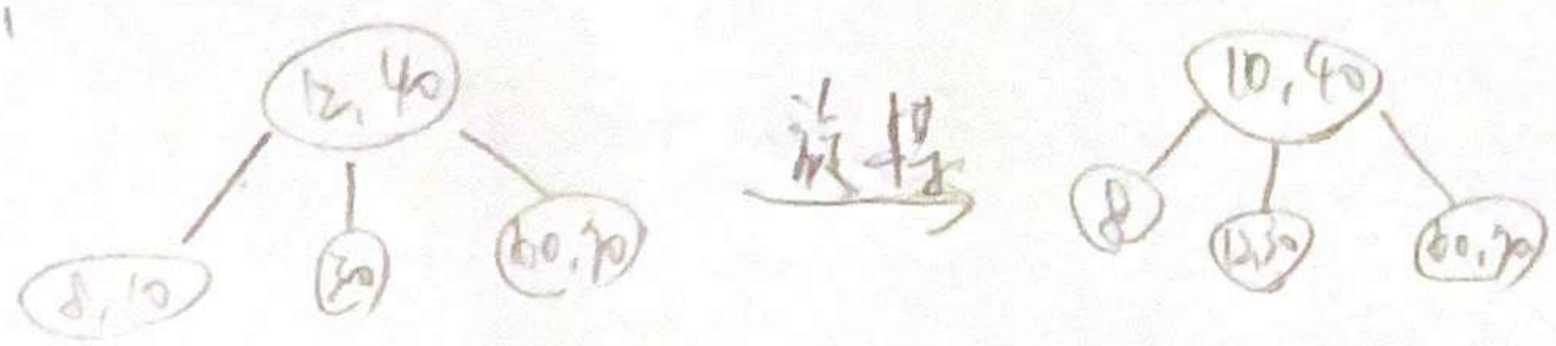
以 AVL 及 2-3-4 tree (4-8)

* AVL：利用 rotate 使樹平衡.

* 2-3-4：將 full 的節點先行 split 以防增加後需 split 狀況

刪除時在走訪到位置的路上，將欠有一個資的 node 合併

ex.



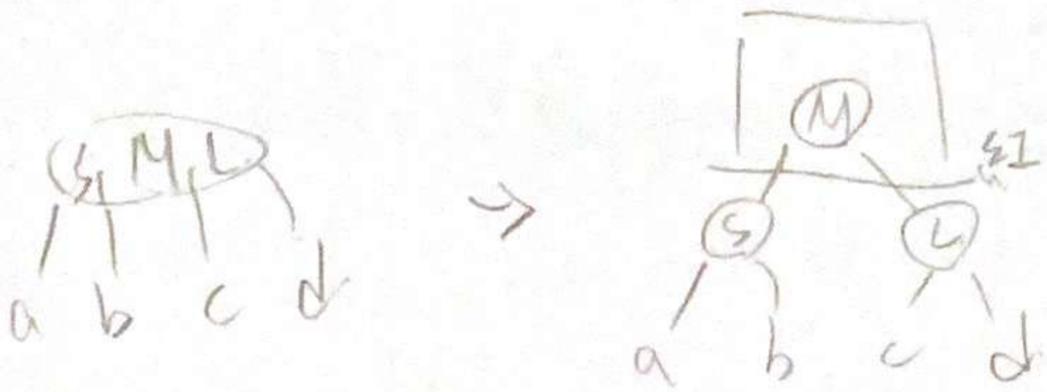
紅黑樹的原理 (4-9)

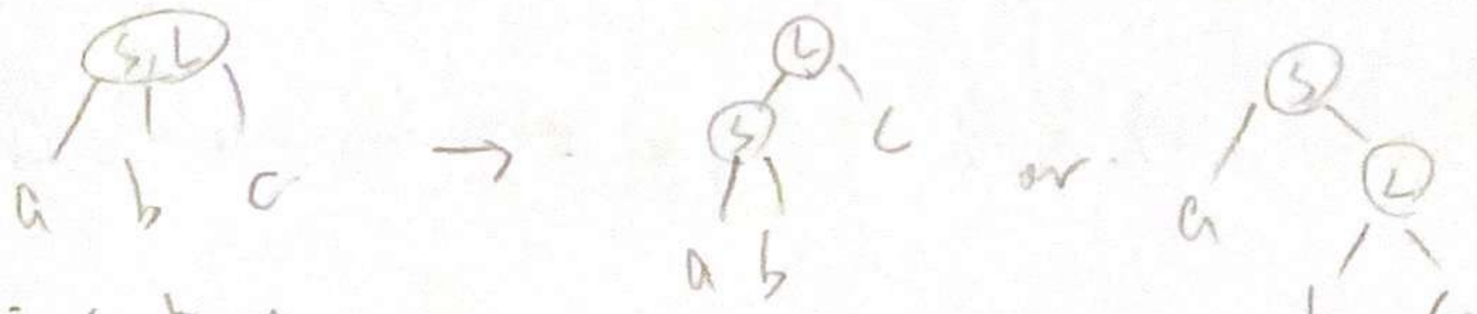
* B- tree (用於海量 Data 處理)

* 結合 2-3-4 & AVL tree 的優點

為 BST 且平衡.

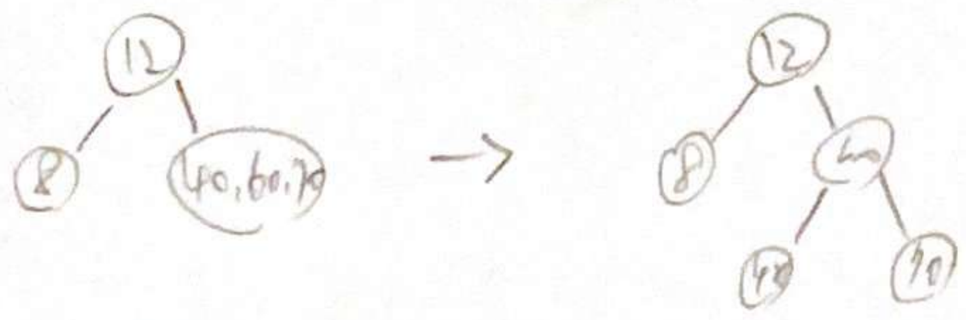
ex.





比較紅黑樹 & 2-3-4 tree (4-10).

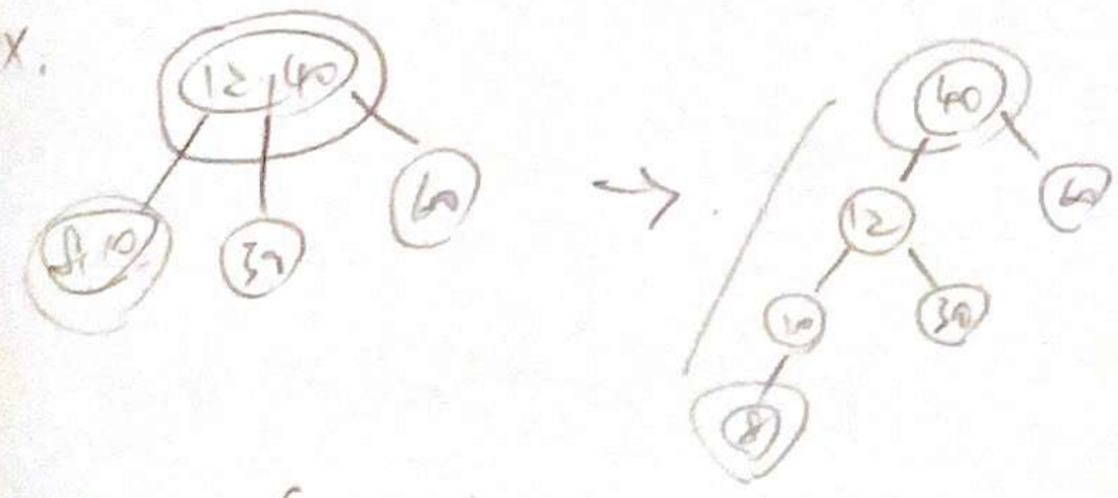
ex.



搜尋的路上所經過的點

一樣多

ex.



點色的長度為原樹高

* 因紅綠個數不限，為不耗時過大，限制走訪時不能連續走2條紅綠

⇒ 若有兩條連續紅綠 ⇒ 旋轉.

紅黑 tree 的 分裂 (4-11)

* 某 Node 之兩個 child link 均為紅：原為 4 node

* 為兩 child 均為黑綠 ⇒ 2-node，但父 parent 為 4 node 的情況

依 LR / RL 可在走訪時記錄

新增資料於紅黑 tree (4-12)

* Red black 為一含有的樹 (二元樹)，新增刪除利用 2-3-4 tree 的概念，利用 AVL 達平衡，並以紅黑判斷是否平衡

* 新增：從樹根往下走至節點，若有兩 Red Child Point 則要處理旋轉 or 改顏色。新增節點 link 為紅色，若新增位置的前一個也紅色則要再處理。

新增資料於紅黑 tree 的範例 (4-13)

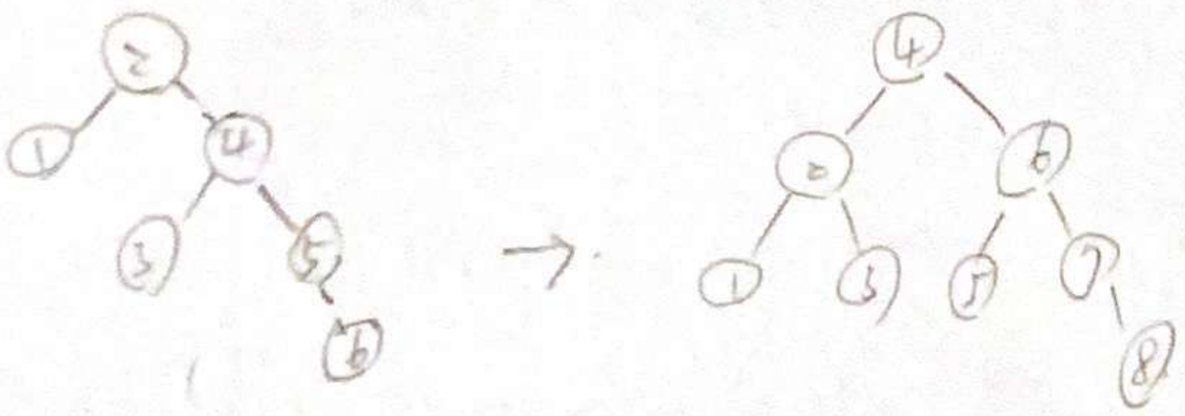
* 若資料不斷加至同側，則會在那側的 link 變紅，將進行旋轉以利平衡

* 所以排序過的資料進入，將不利於建立

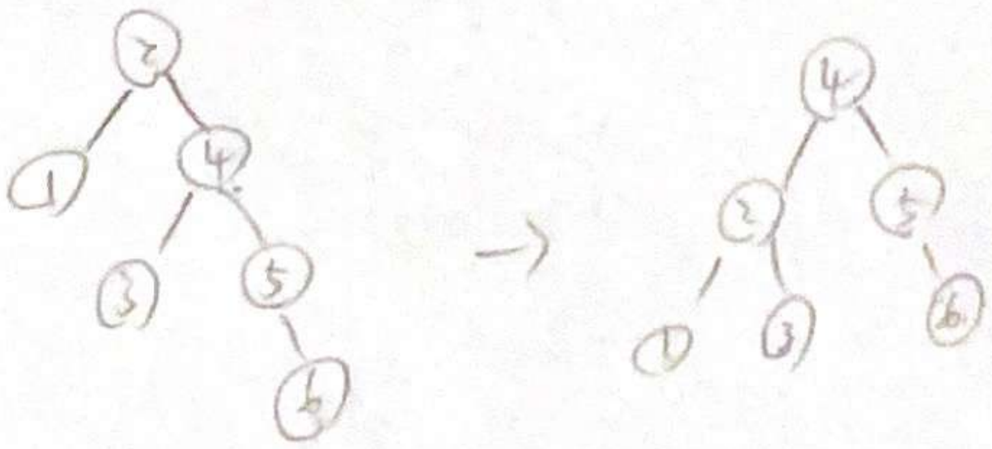
比較紅黑，AVL tree (4-14)

ex, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

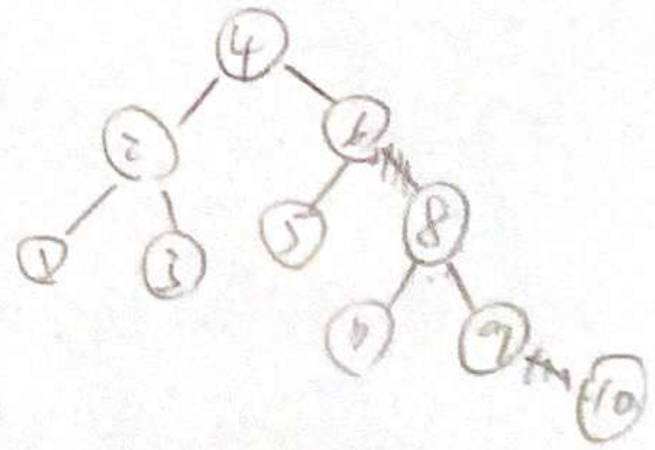
Red black:

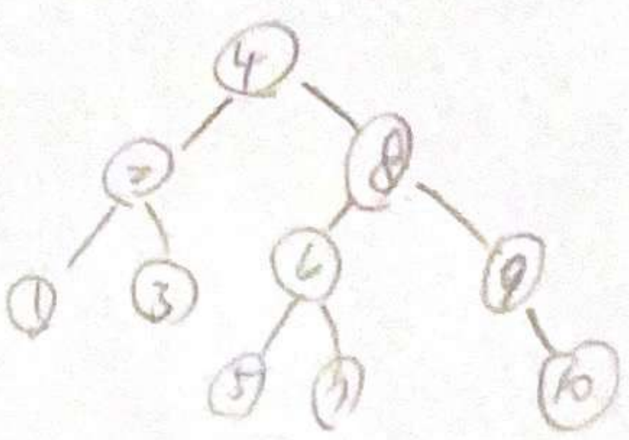


AVL:



於新增刪除時快速





找尋時快速

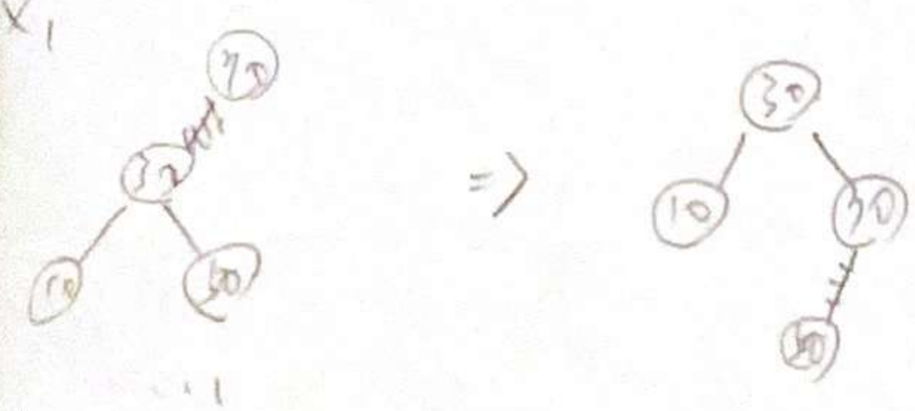
刪除紅黑樹的資料 (4-15)

* 利用旋色 & 旋轉達到目的.
* 將有 2 child 的節點刪除：找出與其鄰近的值交換再刪
* 若只有 1 child：其 parent 必為黑色，用子節點頂補
* 刪除 leaf：① 刪紅色不需再調整 ② 刪黑色：找其兄弟節點達成刪除.
* 刪除點 leaf 且 parent 為紅 (紅→黑，黑改紅，ok)
  (1) without child：改色
  (2) left child：LL + 改
  (3) right child：LR + 改
  (4) two child：以上 + 改 color

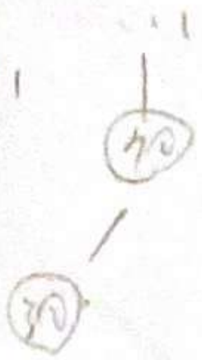
紅黑樹刪除的各種情況 (4-16)

ex.

ex1



ex1



由 70 往上找 red. 以進行 rotate

平衡二元樹統整 (4-17)

* 空間換時間：2-3 tree
* 增加 leaf：二元搜尋.
* 2-3-4 tree 新增刪除效率↑
* AVL：rotate 使樹平衡，搜尋快.
* Red black：綜合 AVL & 2-3-4 tree 的優點.