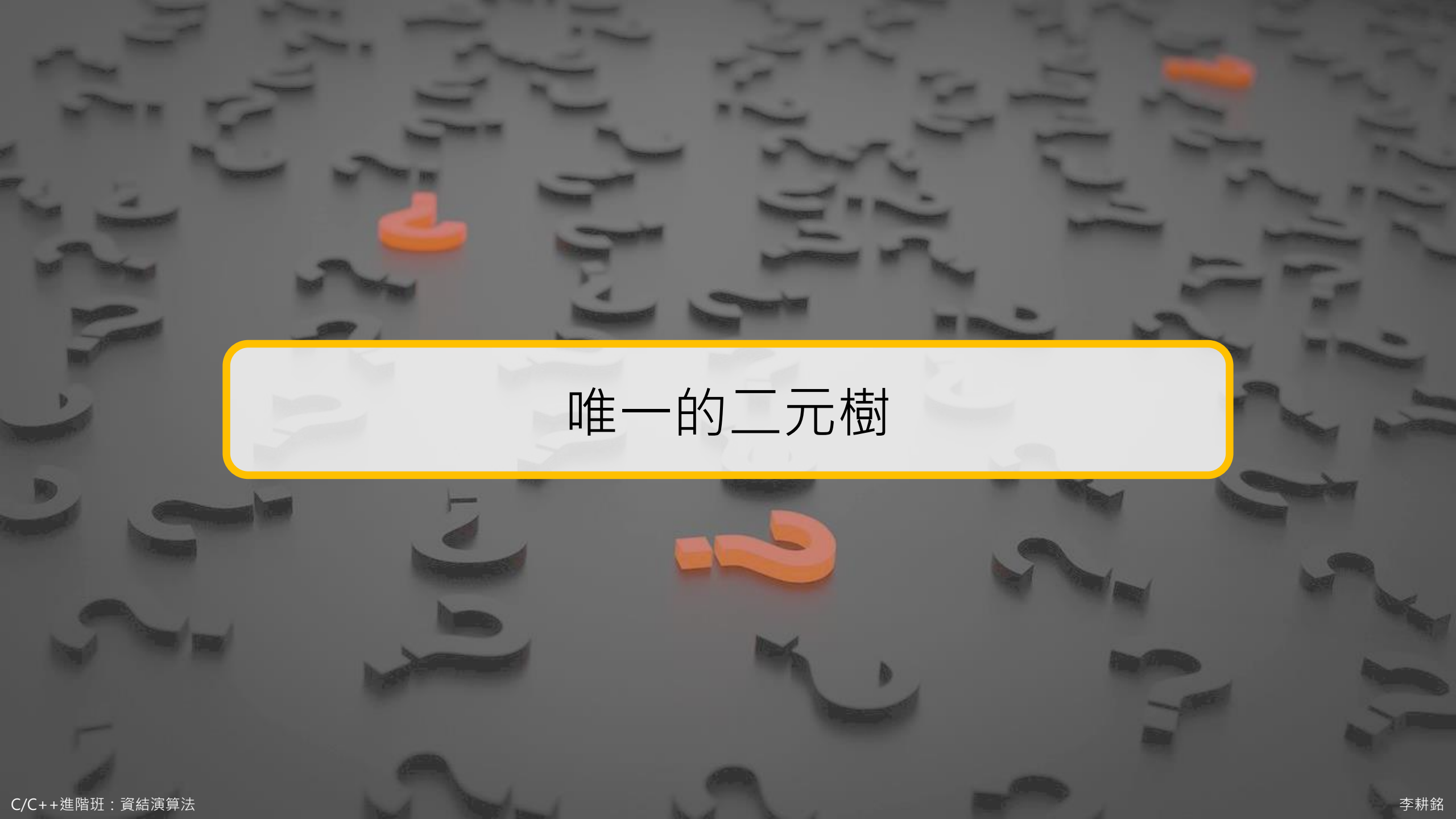
C/C++ 進階班
資料結構

二元樹相關應用
(More about Binary Tree)

李耕銘

課程大綱

- 唯一的二元樹
- 一般樹化成二元樹
- 二元堆疊
- 霍夫曼編碼 Huffman Coding
- 決策樹 Decision Tree
- 樹 (Tree) 與圖 (Graph)



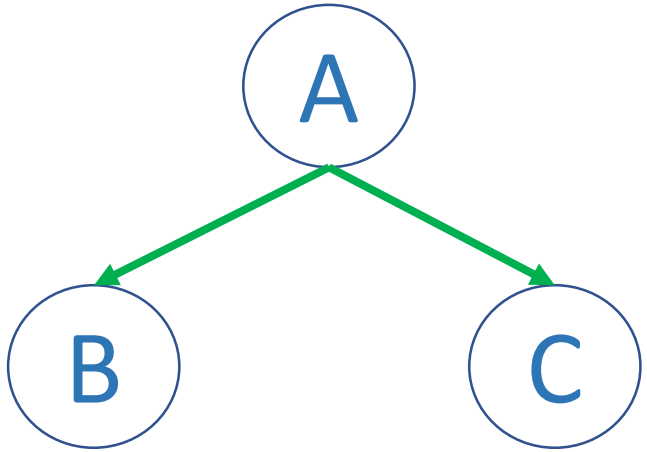
唯一的二元樹

唯一的二元樹

下列狀況可以決定唯一的二元樹

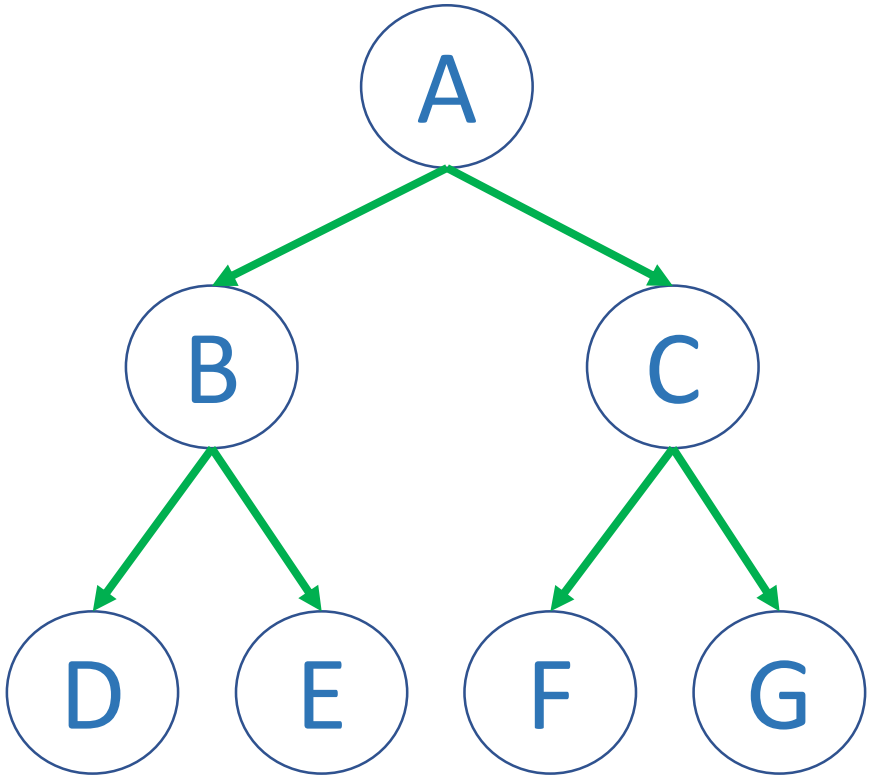
1. 中序與前序
2. 中序與後序

前序與後序的結果無法決定唯一的二元樹



唯一的二元樹

- 中序與前序
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG
- 中序與後序
  - ✓ 中序：DBEAFCG
  - ✓ 後序：DEBFGCA
- 前序與後序
  - ✓ 前序：ABDECFG
  - ✓ 後序：DEBFGCA



唯一的二元樹

- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

唯一的二元樹

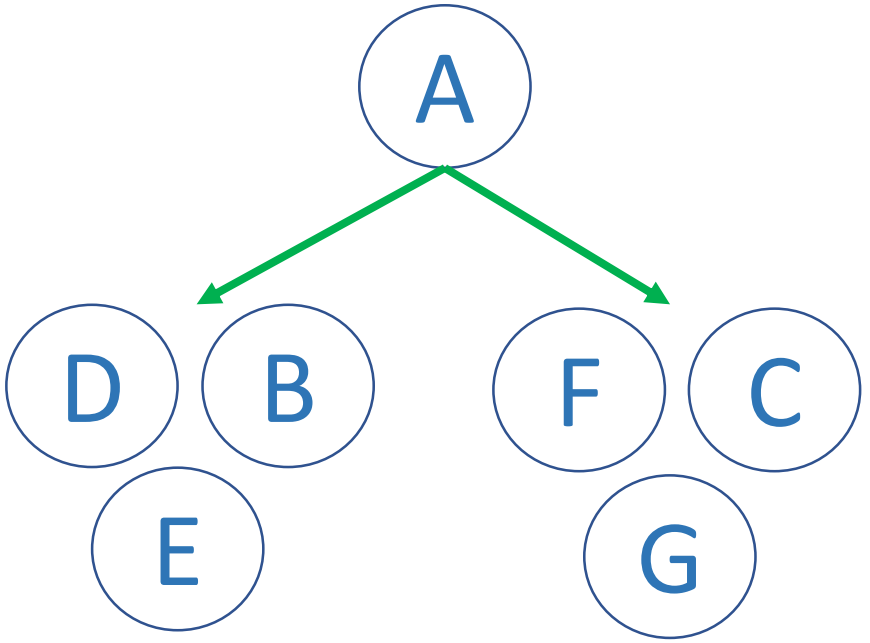
- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

1. 以前序決定根節點 A

唯一的二元樹

- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

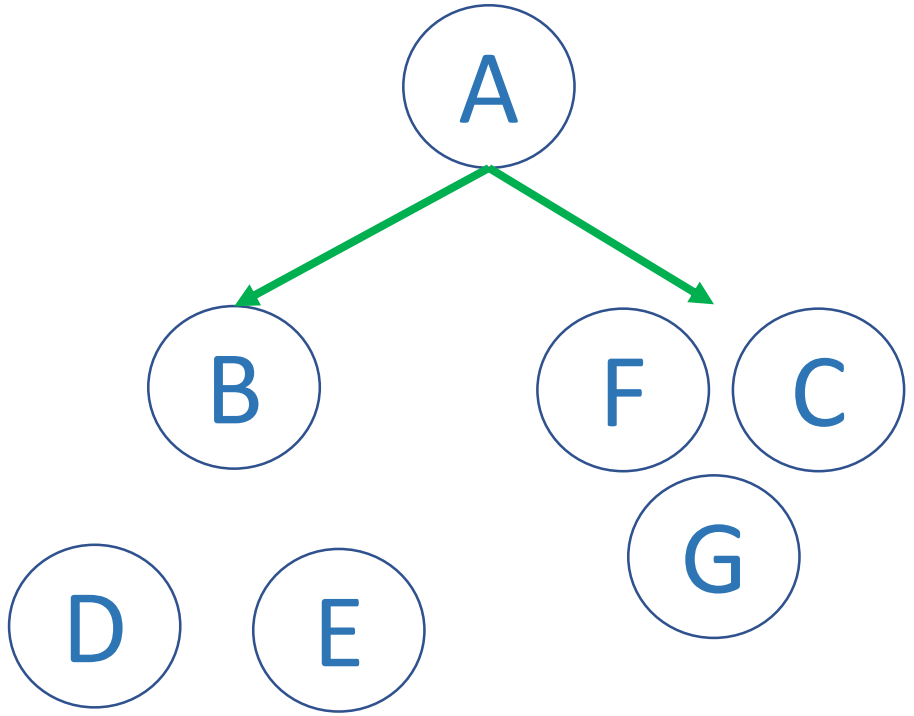
1. 以前序決定根節點 A
2. 以中序分成左右 (DBE)A(FCG)



唯一的二元樹

- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

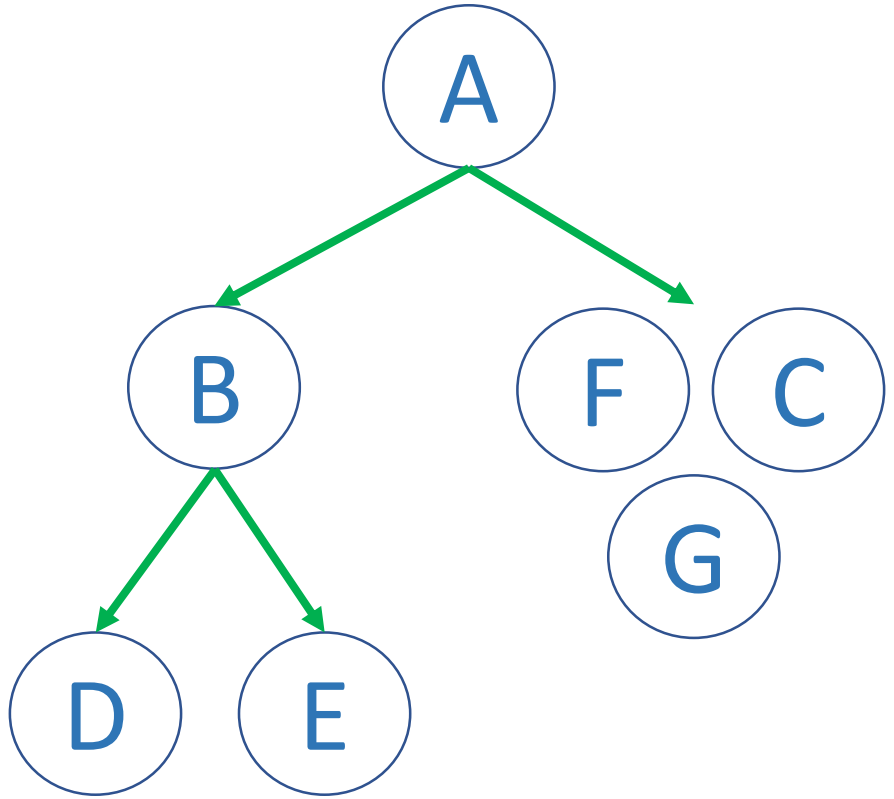
1. 以前序決定根節點 A
2. 以中序分成左右 (DBE)A(FCG)
3. 以前序分出左子樹的第一節點 B



唯一的二元樹

- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

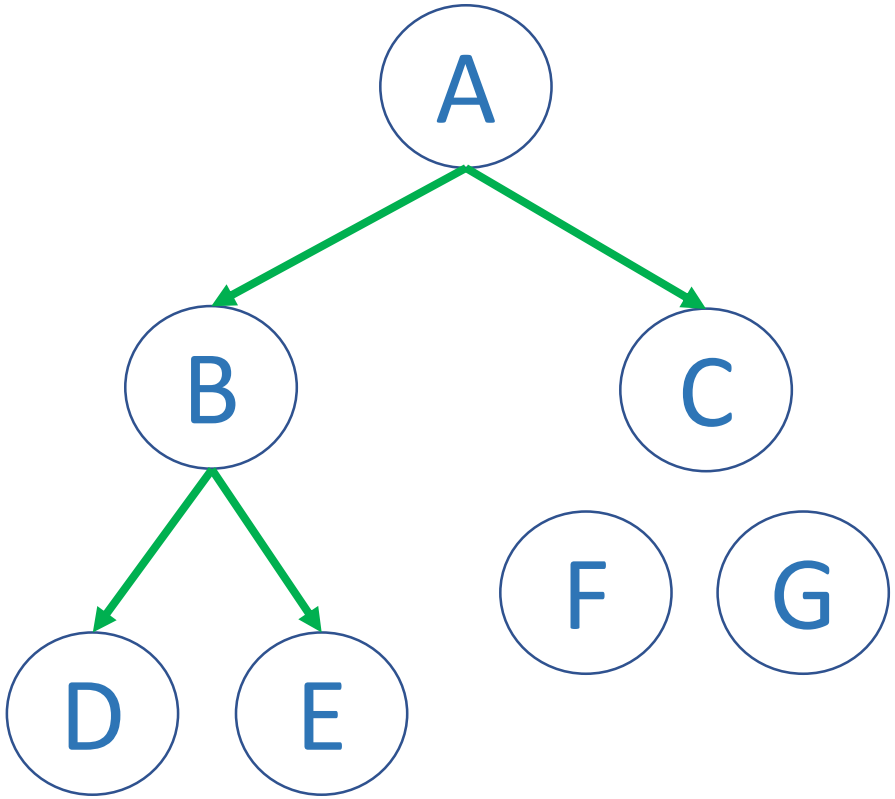
1. 以前序決定根節點 A
2. 以中序分成左右 (DBE)A(FCG)
3. 以前序分出左子樹的第一節點 B
4. 以中序切出 B 的左右子樹 D、E



唯一的二元樹

- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

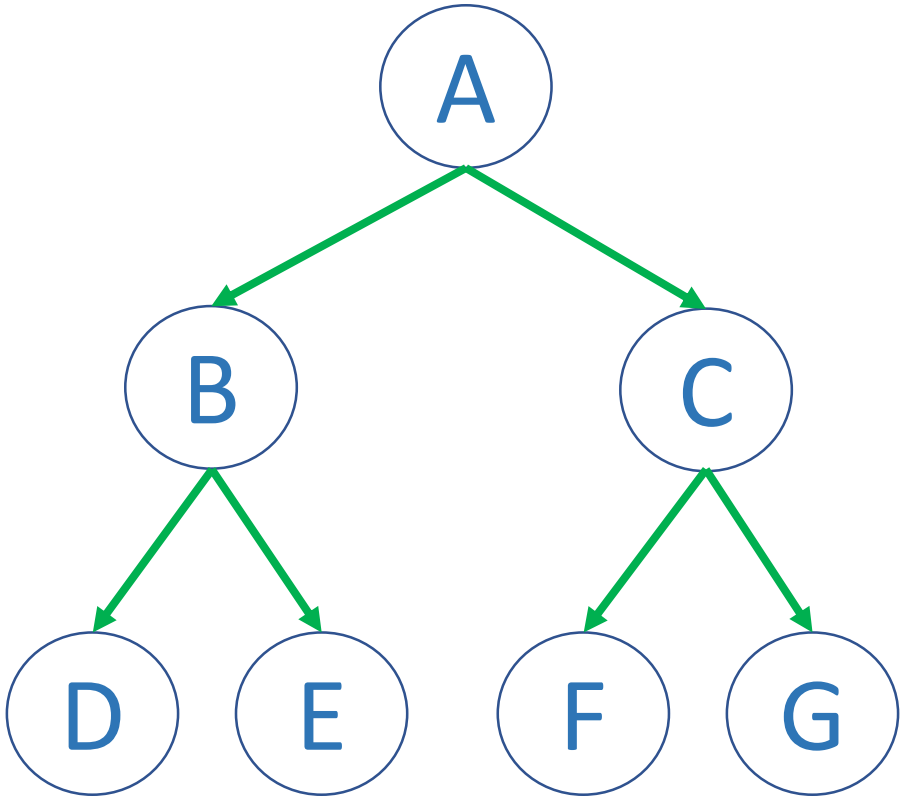
1. 以前序決定根節點 A
2. 以中序分成左右 (DBE)A(FCG)
3. 以前序分出左子樹的第一節點 B
4. 以中序切出 B 的左右子樹 D、E
5. 以前序分出右子樹的第一節點 C



唯一的二元樹

- 以中序與前序決定二元樹
  - ✓ 中序：DBEAFCG
  - ✓ 前序：ABDECFG

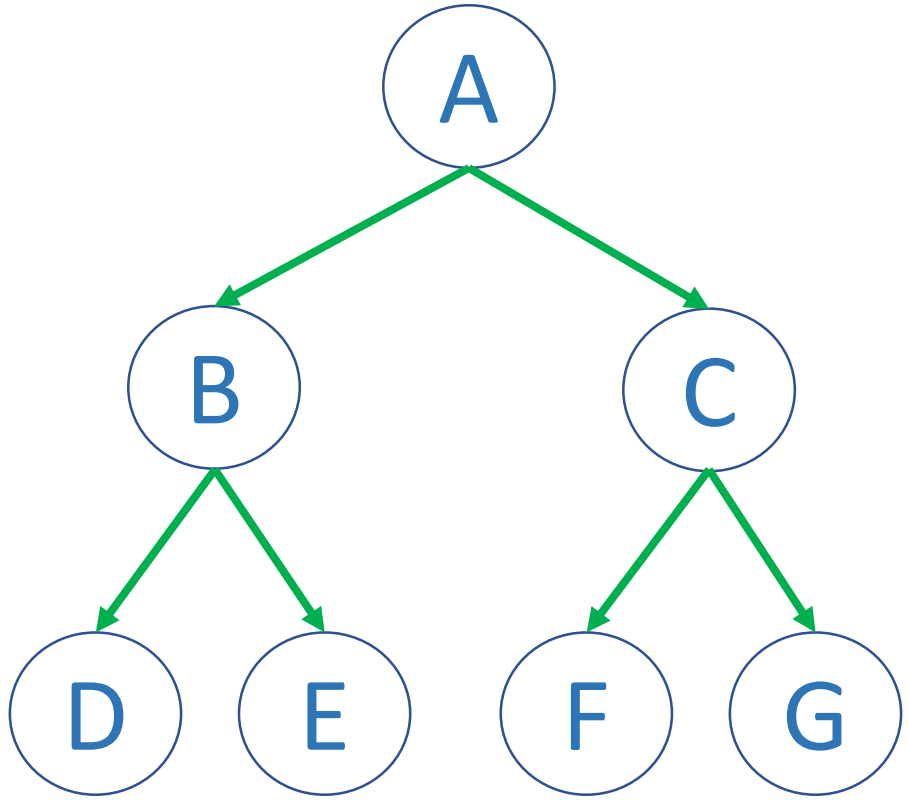
1. 以前序決定根節點 A
2. 以中序分成左右 (DBE)A(FCG)
3. 以前序分出左子樹的第一節點 B
4. 以中序切出 B 的左右子樹 D、E
5. 以前序分出右子樹的第一節點 C
6. 以中序切出 C 的左右子樹 F、G



唯一的二元樹

- 前序：分出根節點
- 中序：分出左右子樹
- 後序：分出根節點

- 中序一定要有，前序後序擇一即可



Practice

Mission

給定一二元樹的中序與後序排列如下：

1. In-order：ACFGBDE
2. Post-order：AGFCDEB

請試著重建這棵二元樹並畫出之。

Practice

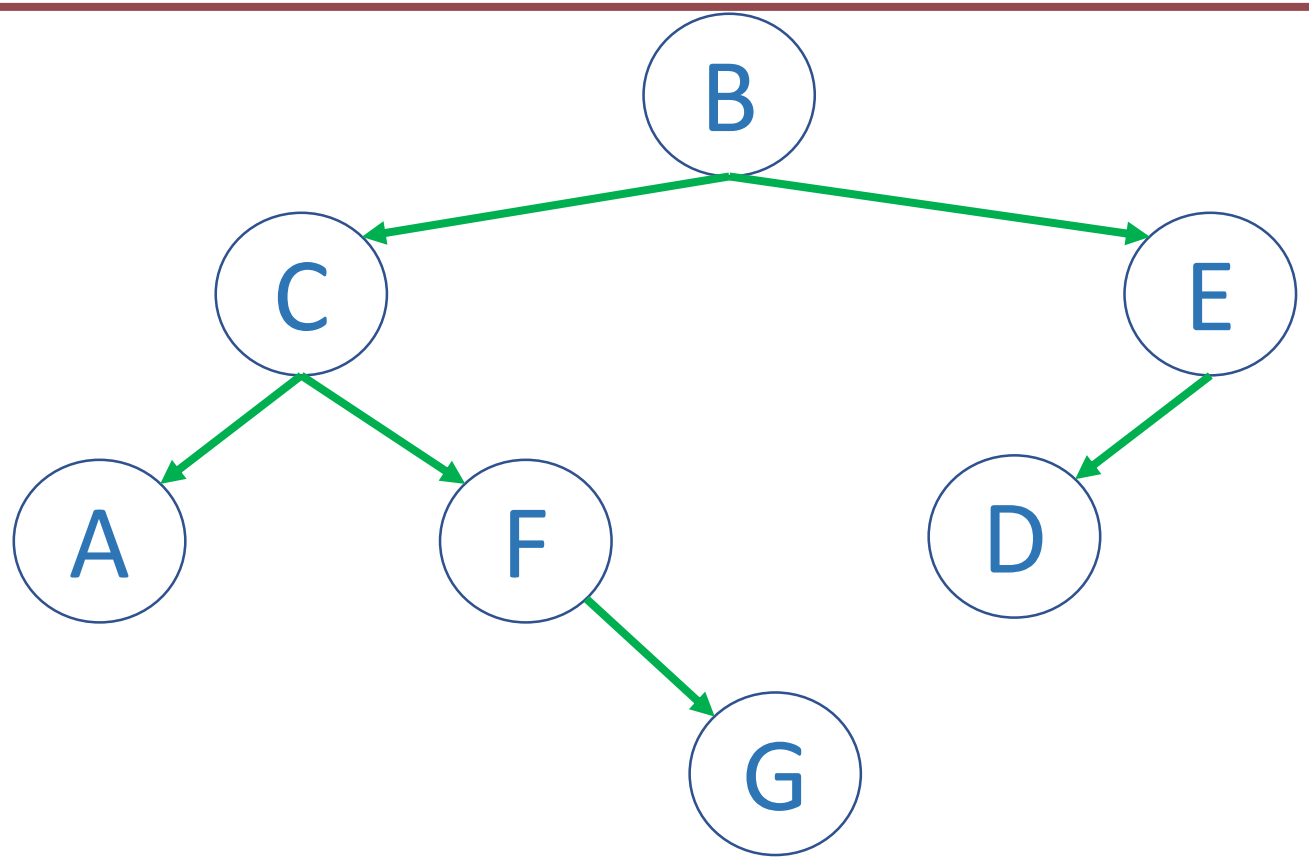
Mission

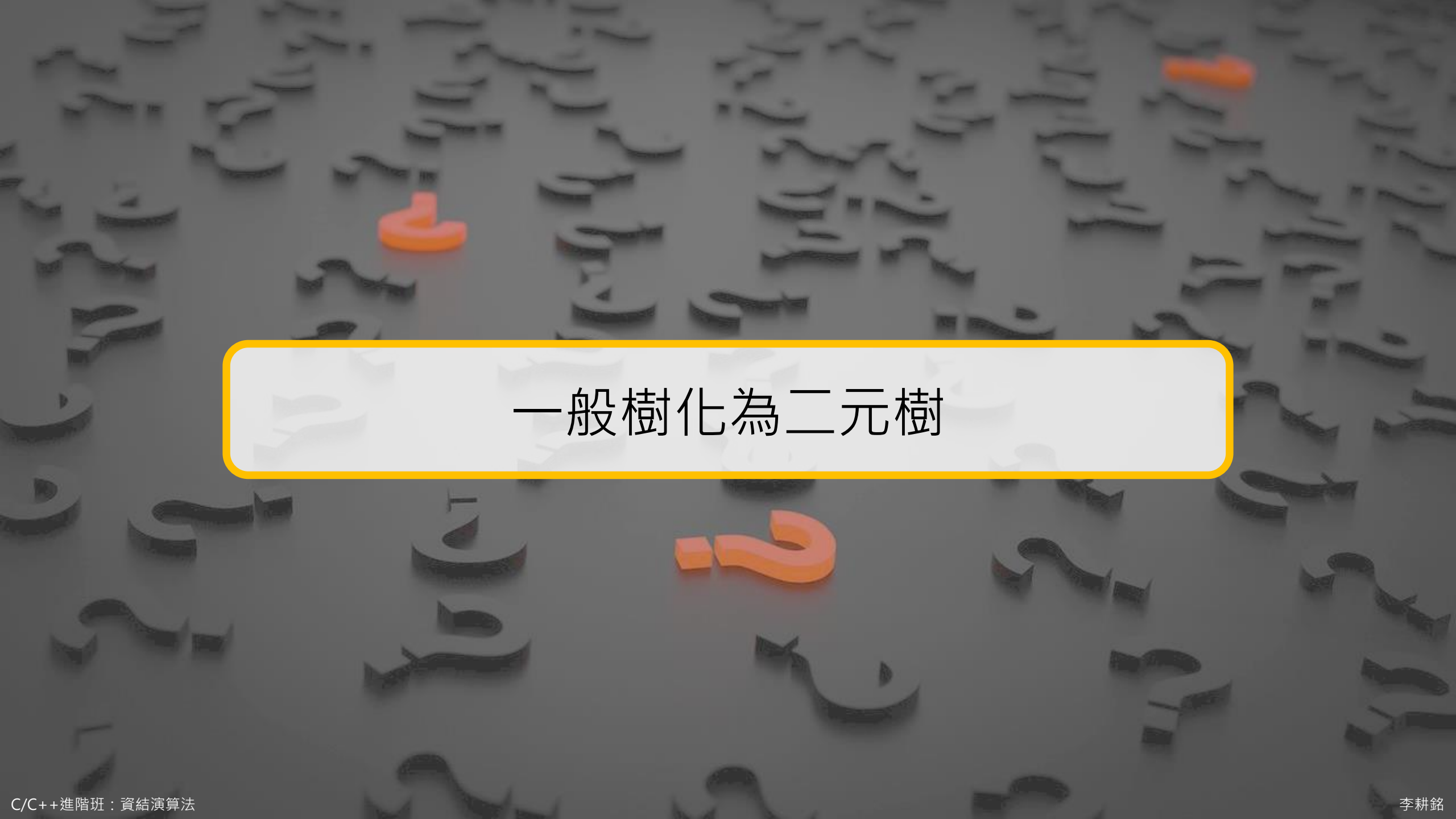
1. In-order：ACFGBDE
2. Post-order：AGFCDEB

Practice

Mission

1. In-order：ACFGBDE
2. Post-order：AGFCDEB

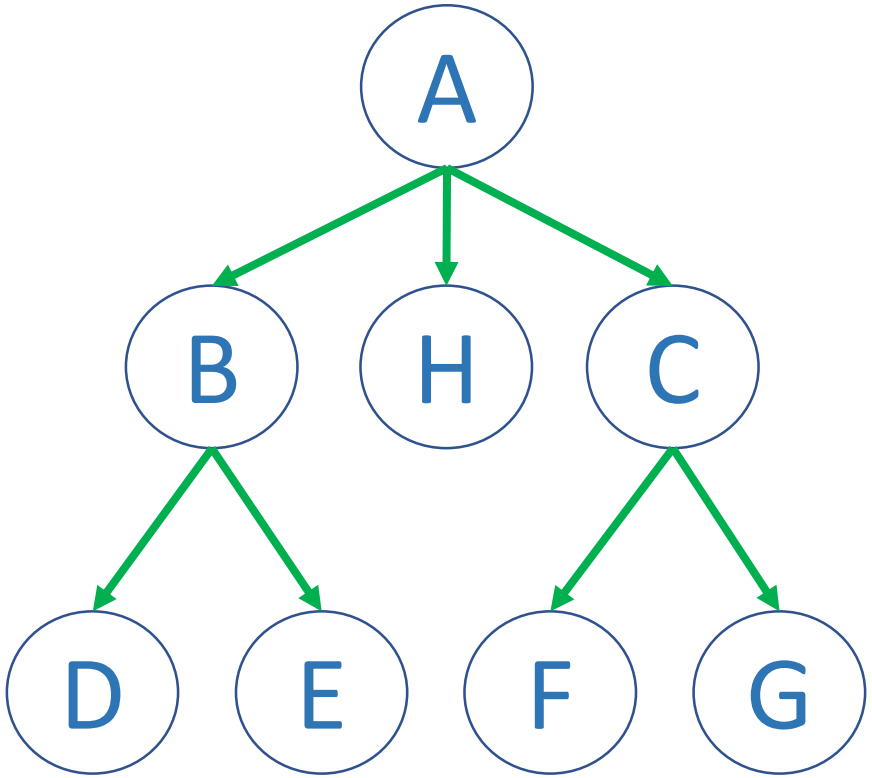




一般樹化為二元樹

一般樹化為二元樹

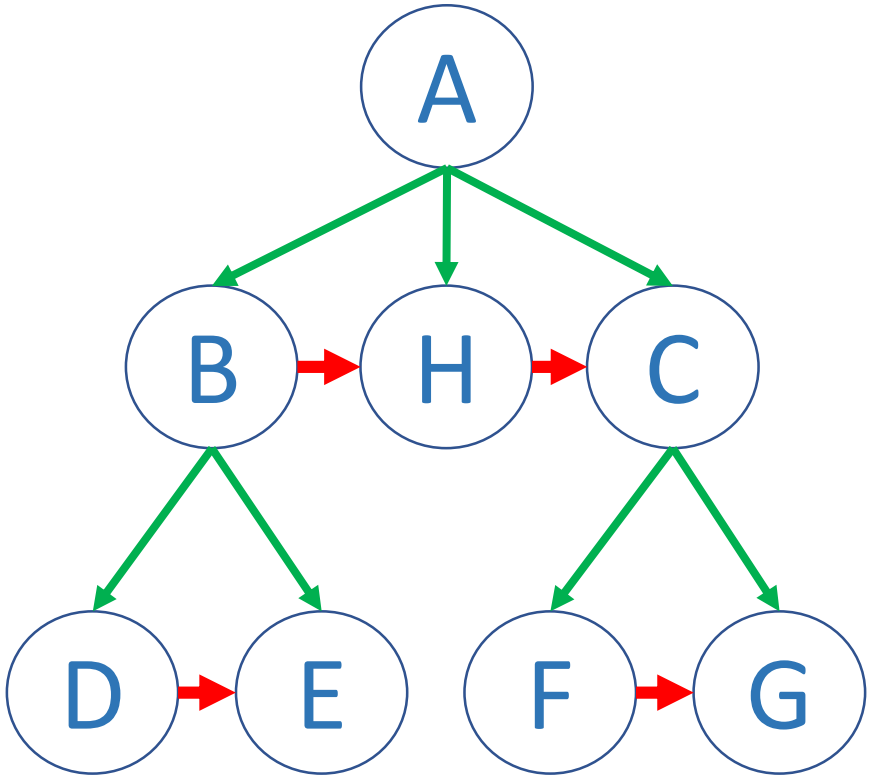
- 回憶：一般樹與二元樹的最大差異
  - ✓ 二元樹最多只有兩個分歧
- 要將一般樹化為二元樹
  - ✓ 刪去多餘連結，只保留/新增一個



一般樹化為二元樹

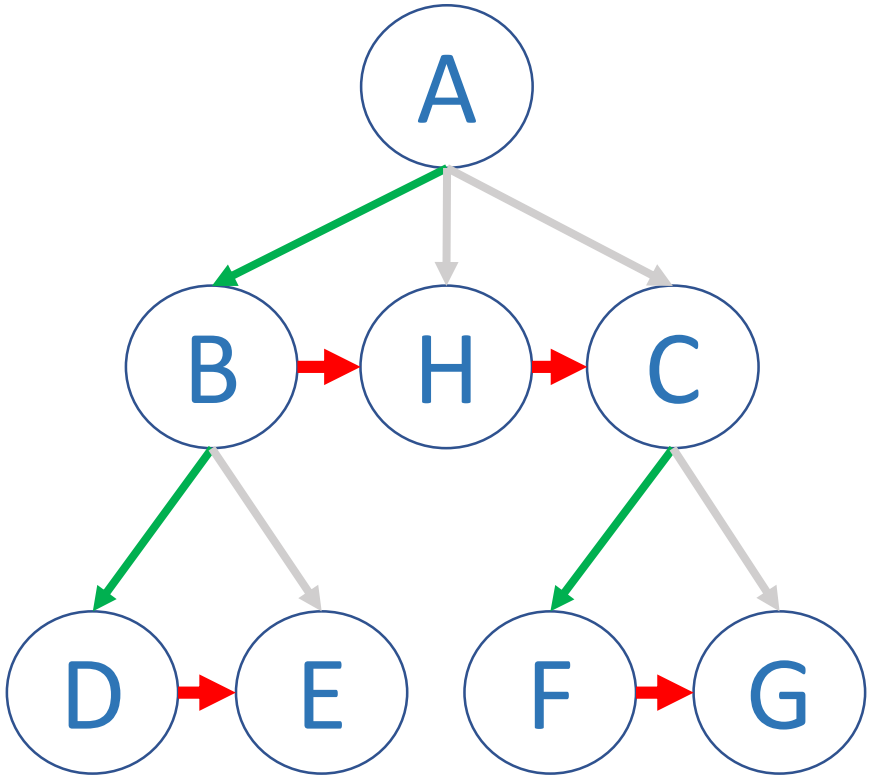
- 步驟

  1. 將兄弟節點們用平行線連接



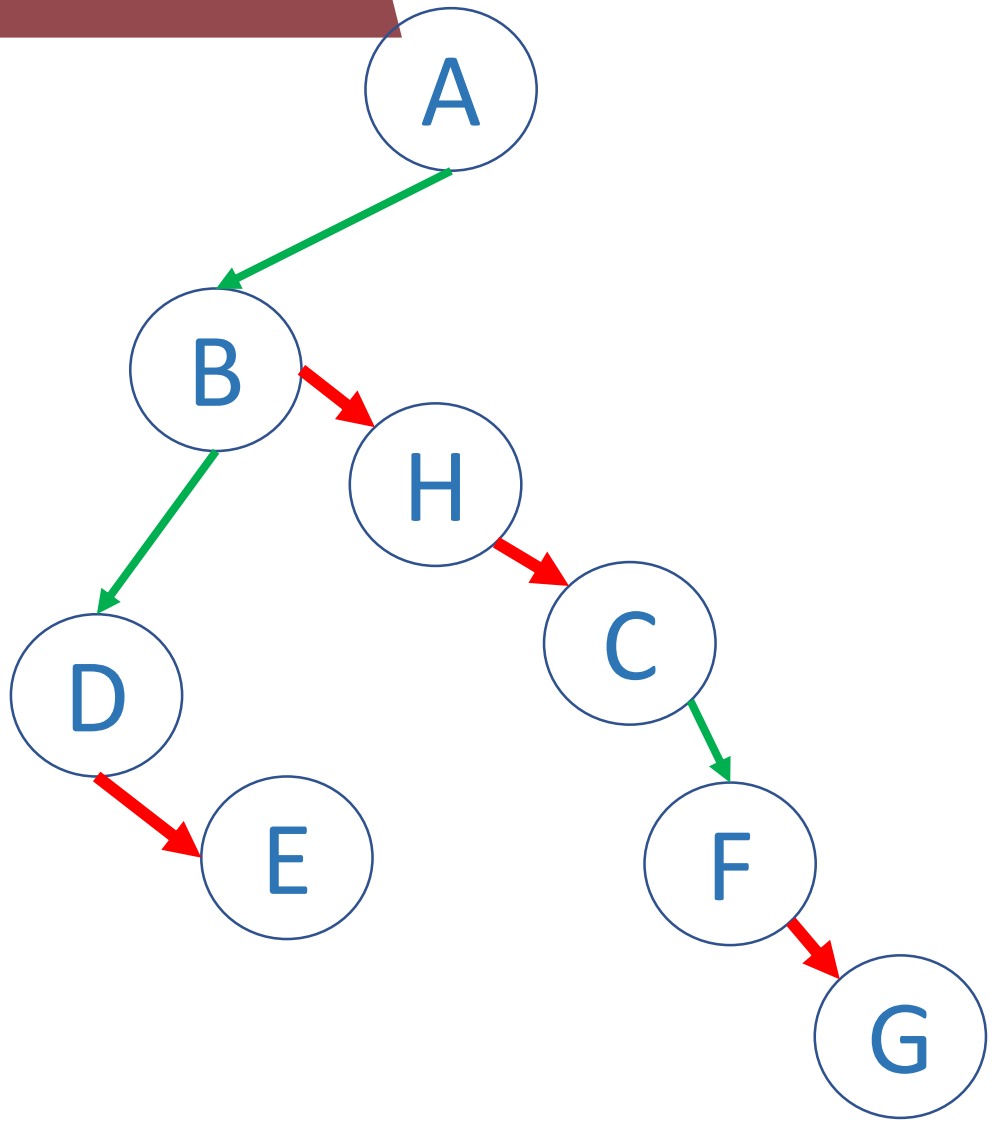
一般樹化為二元樹

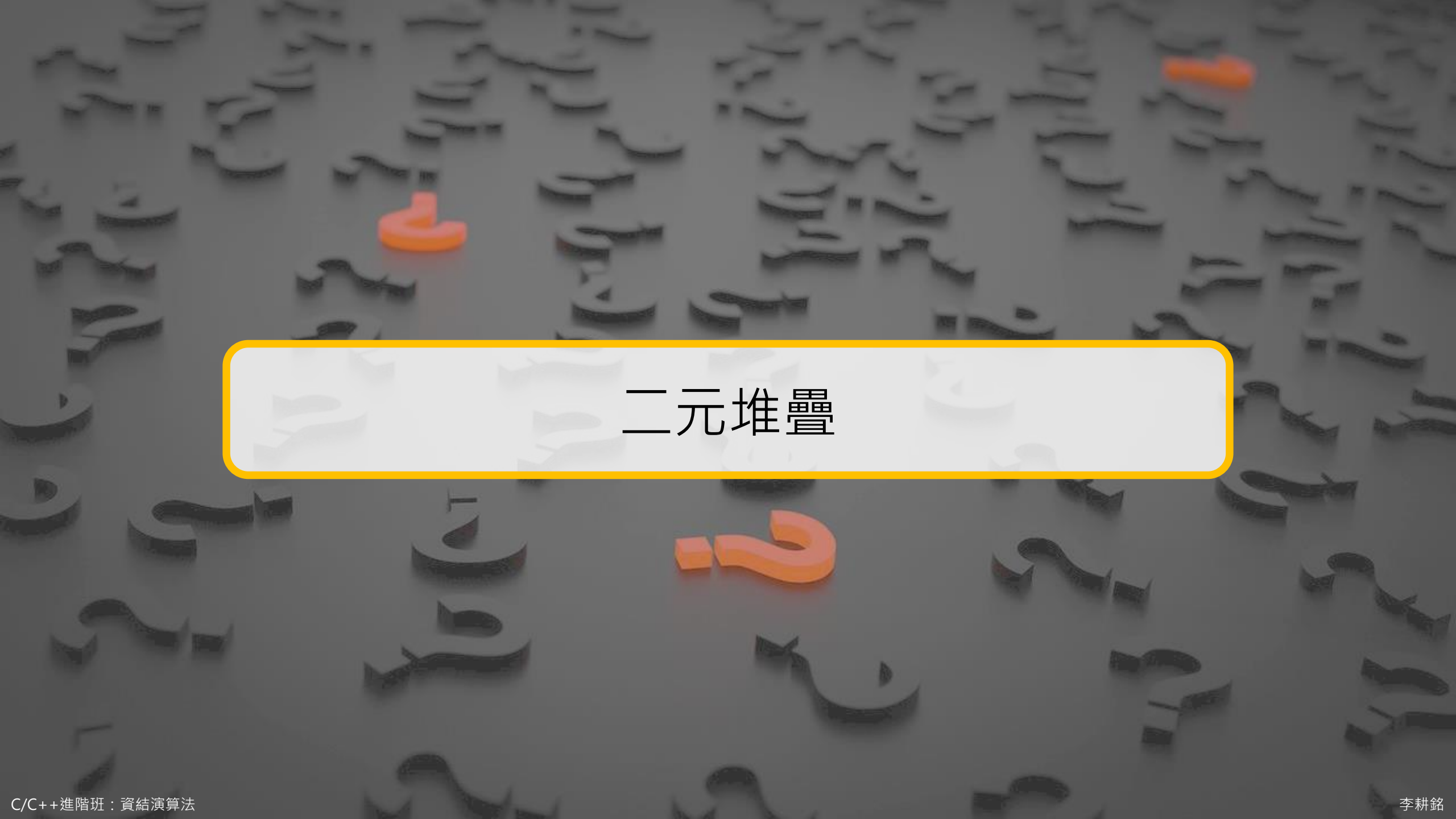
- 步驟
  1. 將兄弟節點們用平行線連接
  2. 只保留每個節點中最左邊的連結



一般樹化為二元樹

- 步驟
  1. 將兄弟節點們用平行線連接
  2. 只保留每個節點中最左邊的連結
  3. 右邊的兄弟節點順時針轉 45 度





二元堆疊

二元堆疊

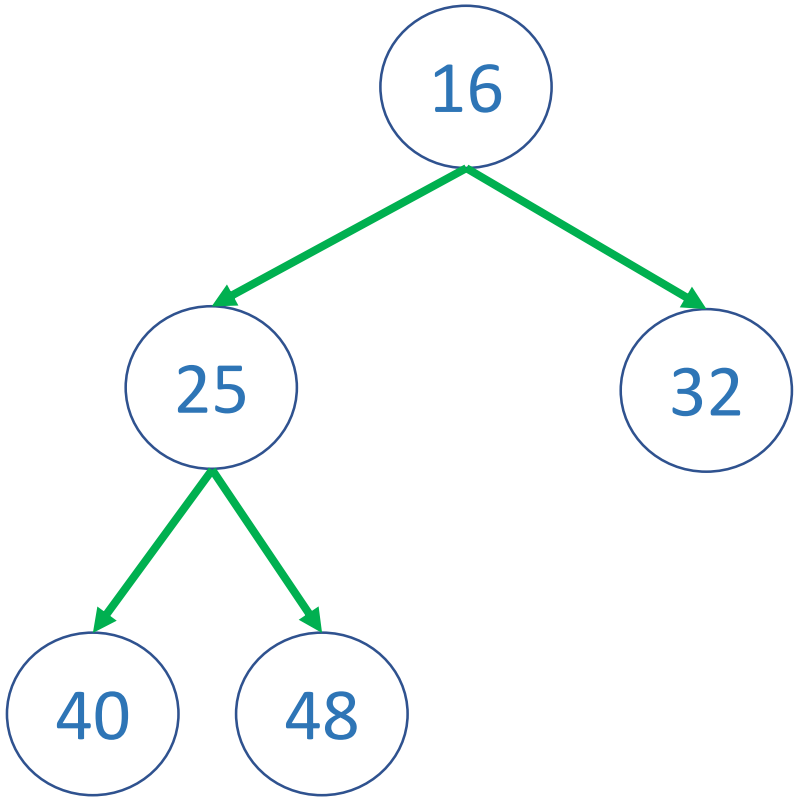
二元堆疊 (Binary Heap)

- 由完整二元樹組成
- 為了 heap sort 而發明出來
- 可分成 Min-Heap 與 Max-Heap
  1. Min-Heap

     根節點為所有子樹中的最小值
  2. Max-Heap

     根節點為所有子樹中的最大值

Min-Heap



二元堆疊

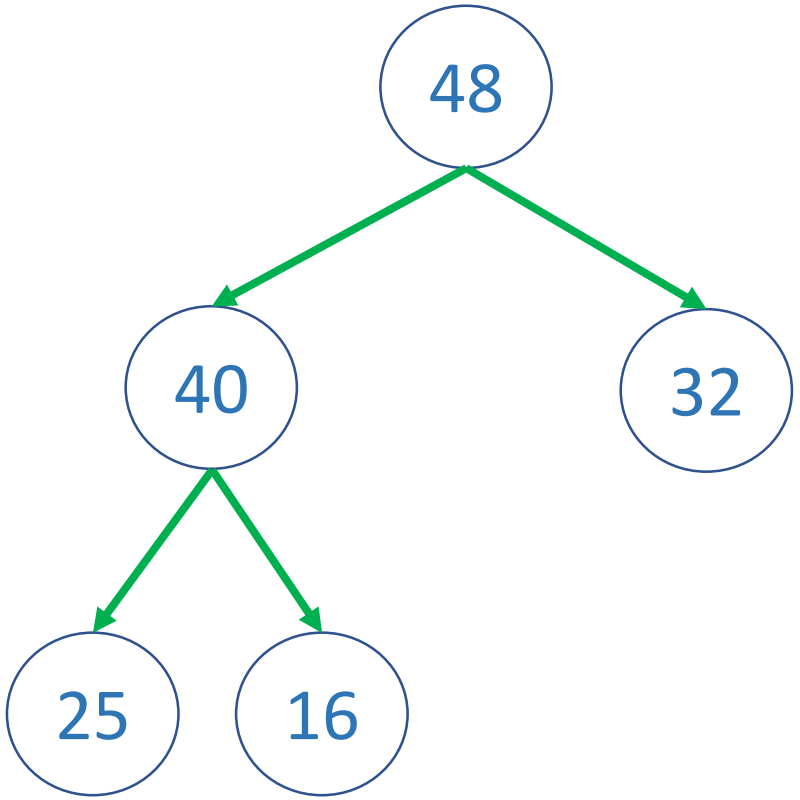
二元堆疊 (Binary Heap)

- 由完整二元樹組成
- 為了 heap sort 而發明出來
- 可分成 Min-Heap 與 Max-Heap
  1. Min-Heap

     根節點為所有子樹中的最小值
  2. Max-Heap

     根節點為所有子樹中的最大值

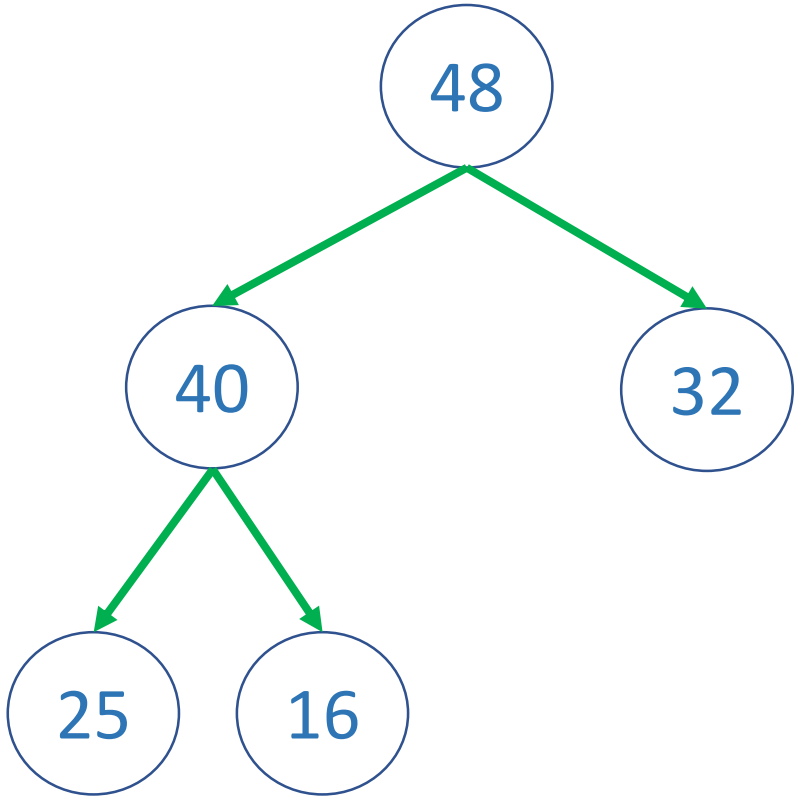
Max-Heap



二元堆疊

二元堆疊 (Binary Heap)

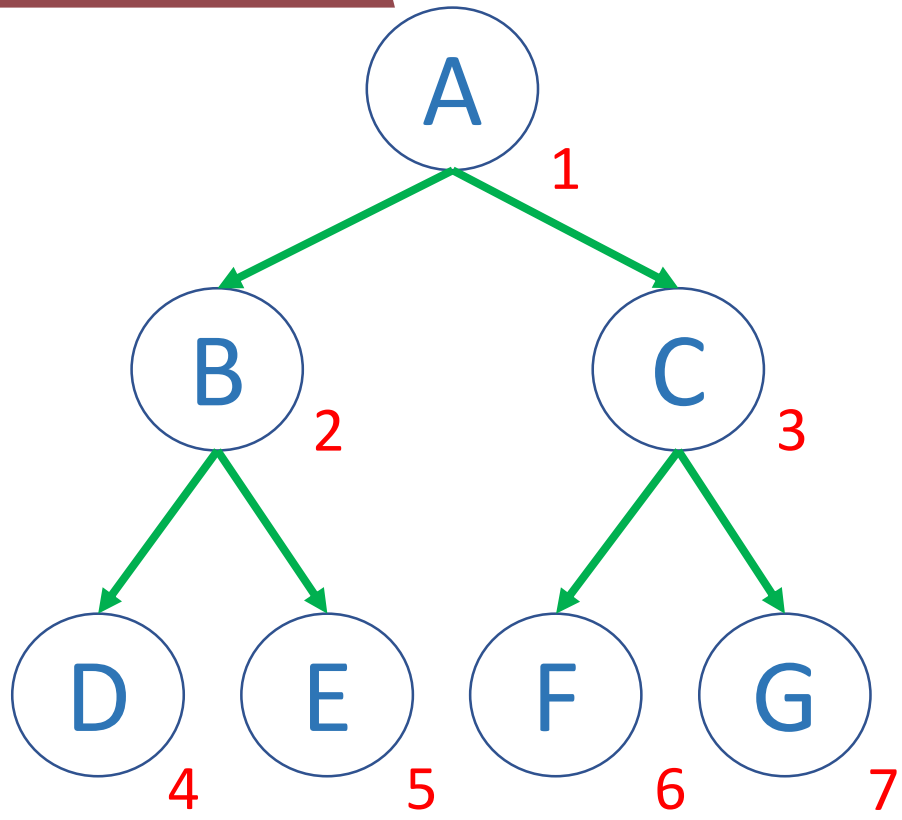
- 每個節點最多有兩個子節點
- 同一階層 (level) 內要從左到右排列
  - ➢ 完整二元樹可以直接用陣列表達
- Min-Heap 與 Max-Heap
  1. Min-Heap：根節點為最小值
  2. Max-Heap：根節點為最大值



48	40	32	25	16

陣列表示法

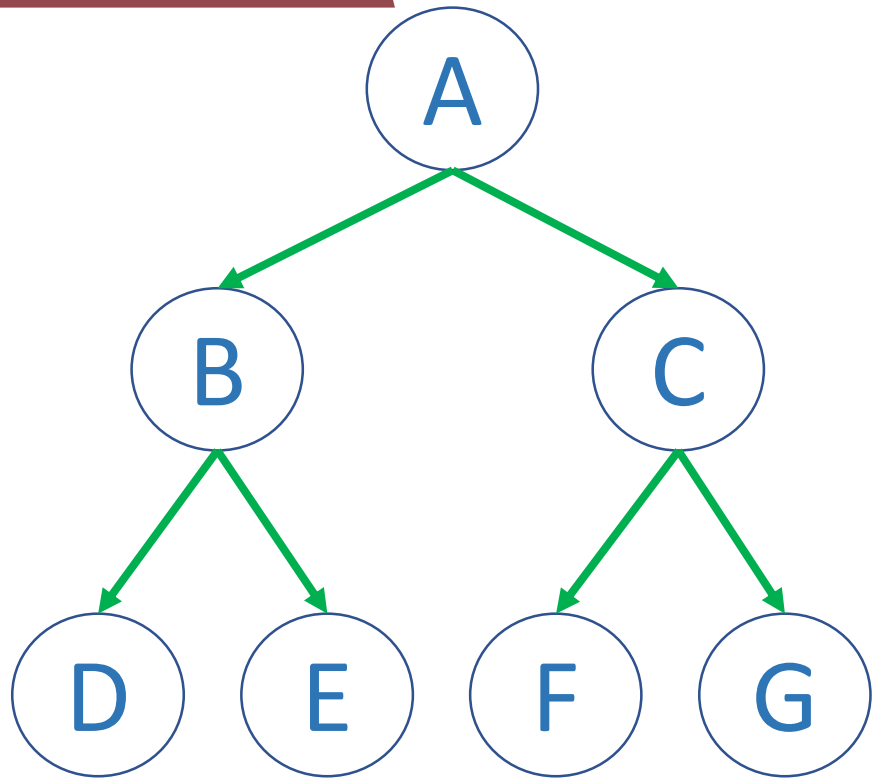
- 第 k 個節點 (k 從 1 開始)
  - ✓ Left child：索引值 $2k-1$
  - ✓ Right child：索引值 $2k$
  - ✓ Parent：索引值 $\left\lfloor \frac{k-2}{2} \right\rfloor$



A	B	C	D	E	F	G

陣列表示法

- 陣列中第 i 個索引值
  - ✓ Left child：索引值 $2i+1$
  - ✓ Right child：索引值 $2i+2$
  - ✓ Parent：索引值 $\left\lfloor \frac{i-1}{2} \right\rfloor$
- 範例，C 的索引值 2：
  - ✓ Left child：索引值 $2\times 2+1=5$
  - ✓ Right child：索引值 $2\times 2+2=6$
  - ✓ Parent：索引值 $\left\lfloor \frac{2-1}{2} \right\rfloor = 0$

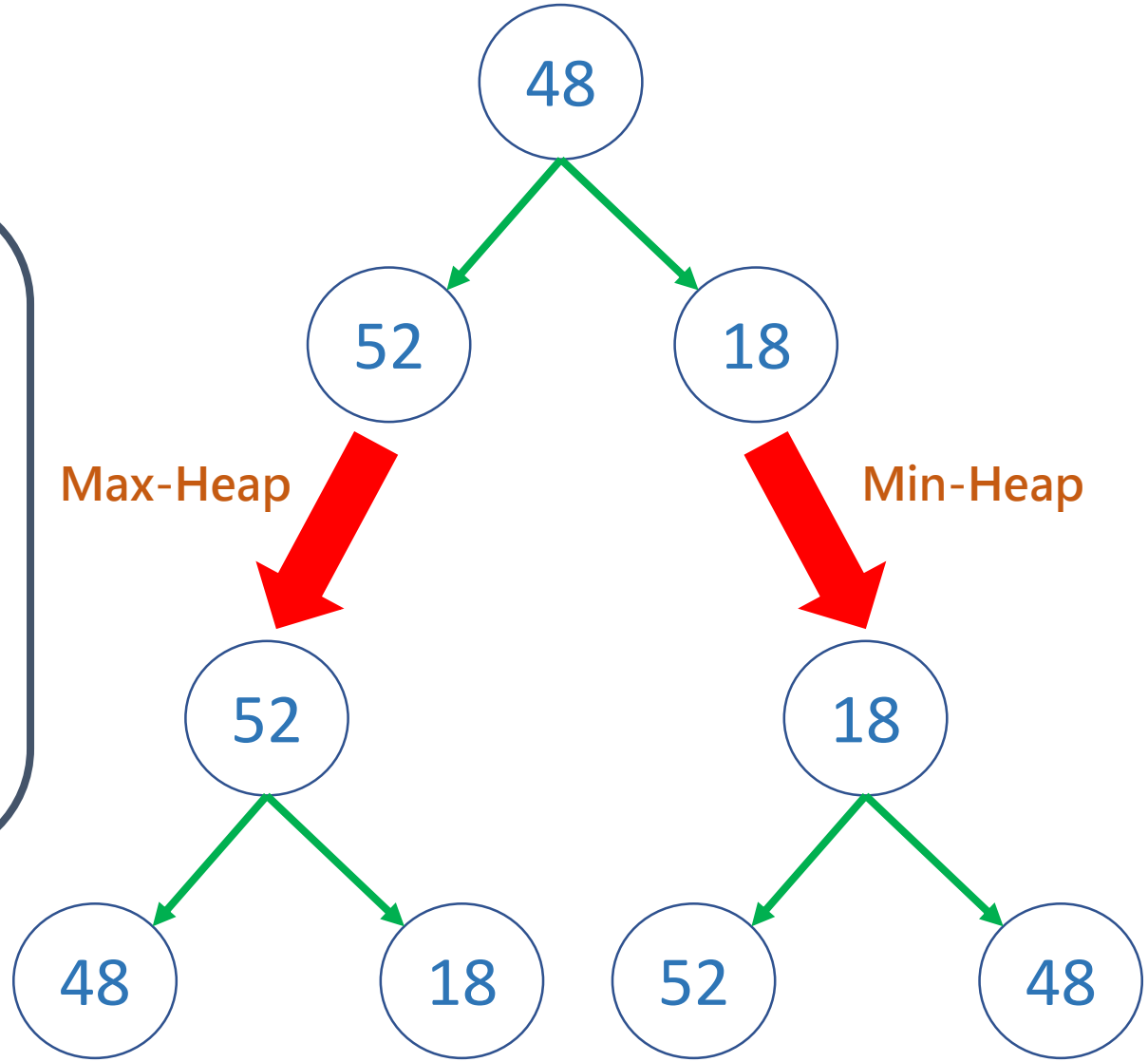


A	B	C	D	E	F	G
0	1	2	3	4	5	6

二元堆疊

Heapify

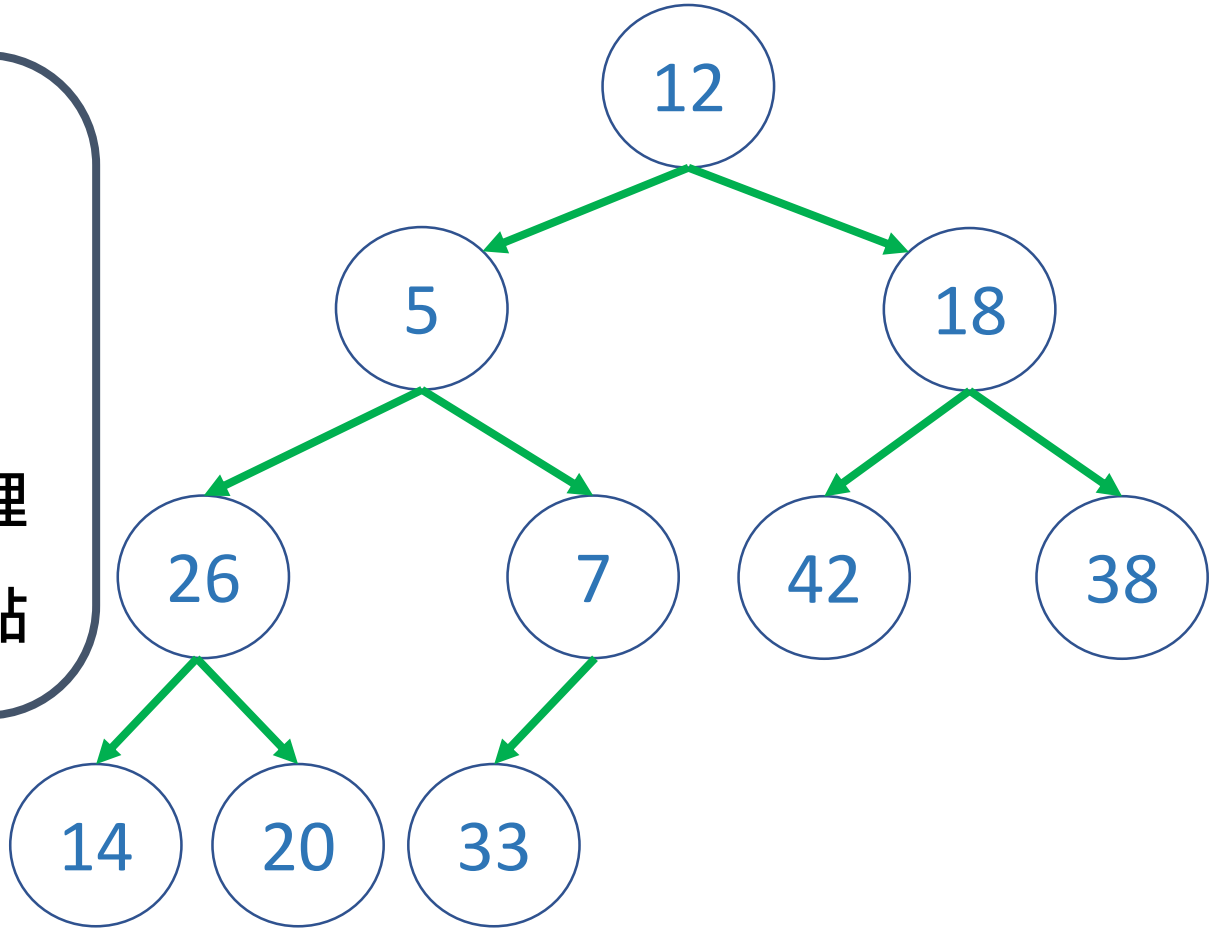
- Min-Heap：根節點為最小值
  - ➢ 取左、中、右最小的節點當根節點
- Max-Heap：根節點為最大值
  - ➢ 取左、中、右最小的節點當根節點



二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

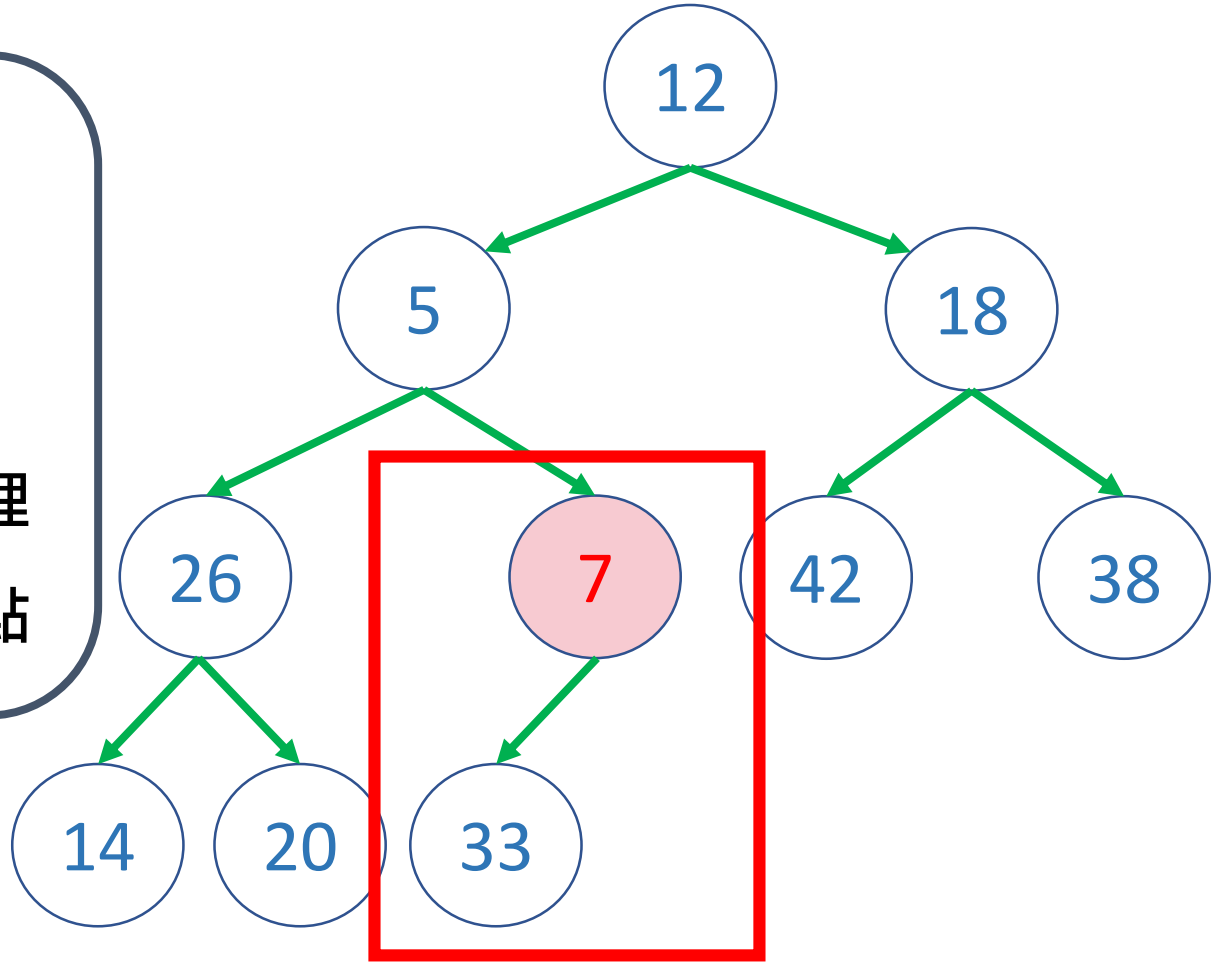


12	5	18	26	7	42	38	14	20	33

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

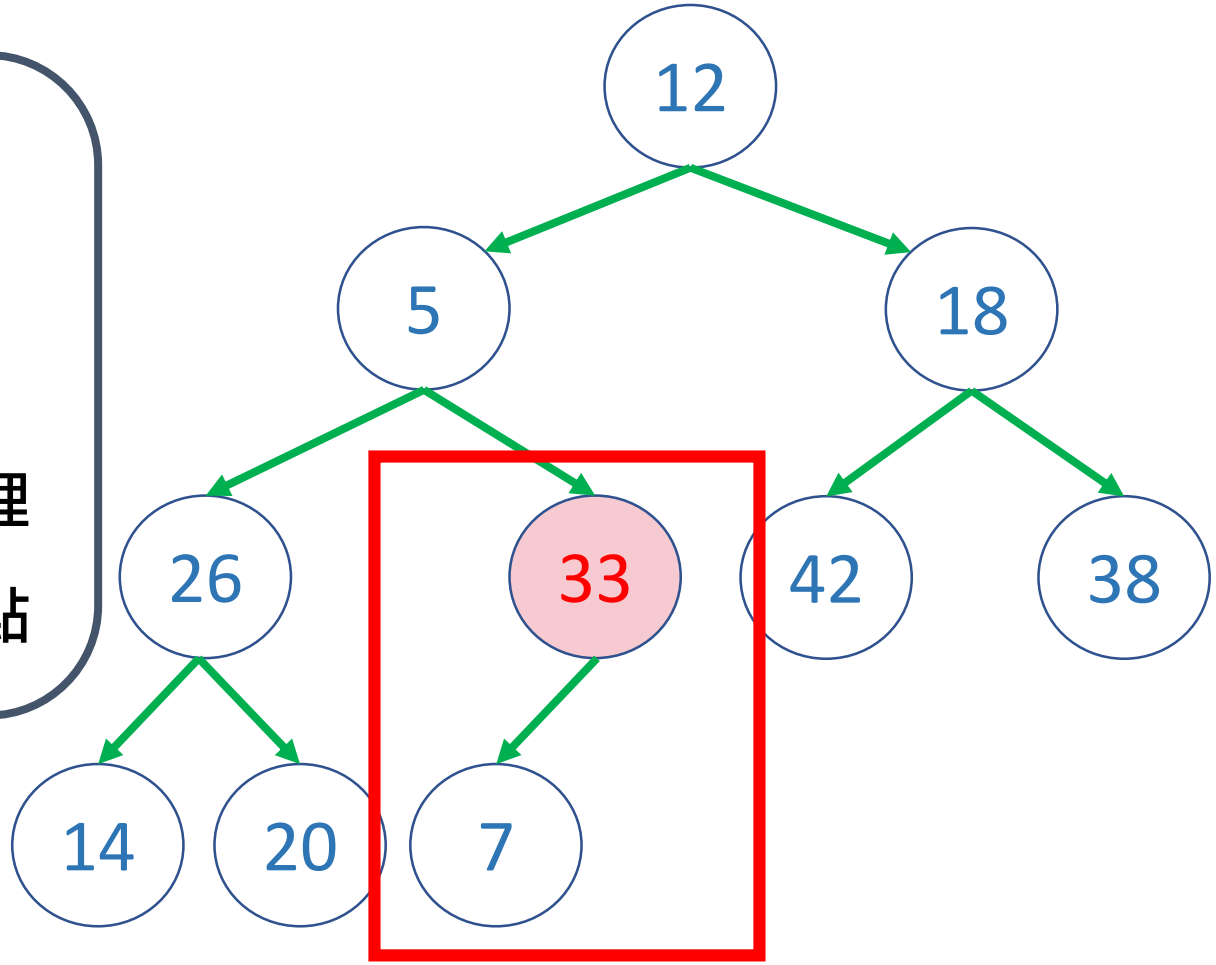


12	5	18	26	7	42	38	14	20	33

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

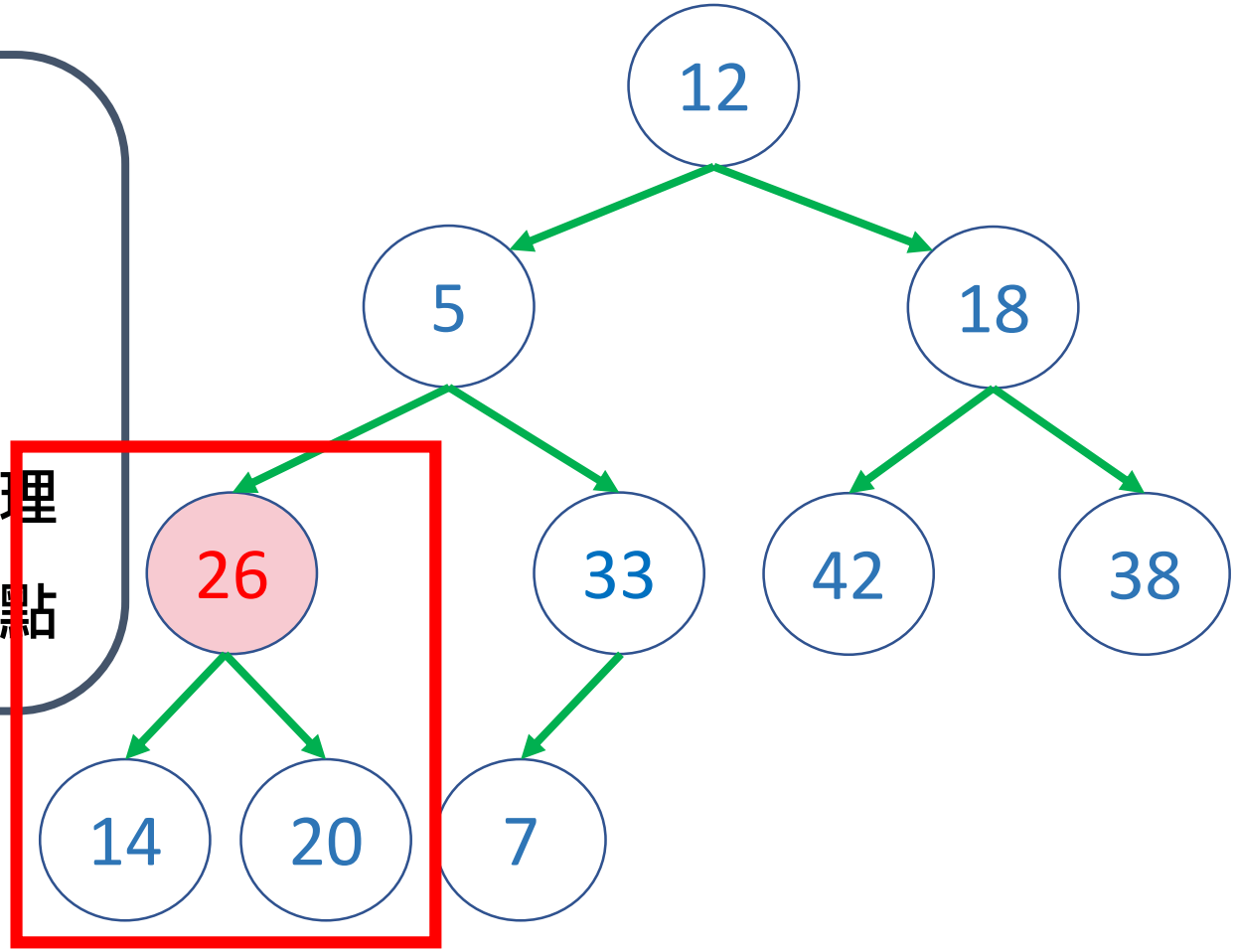


12	5	18	26	33	42	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

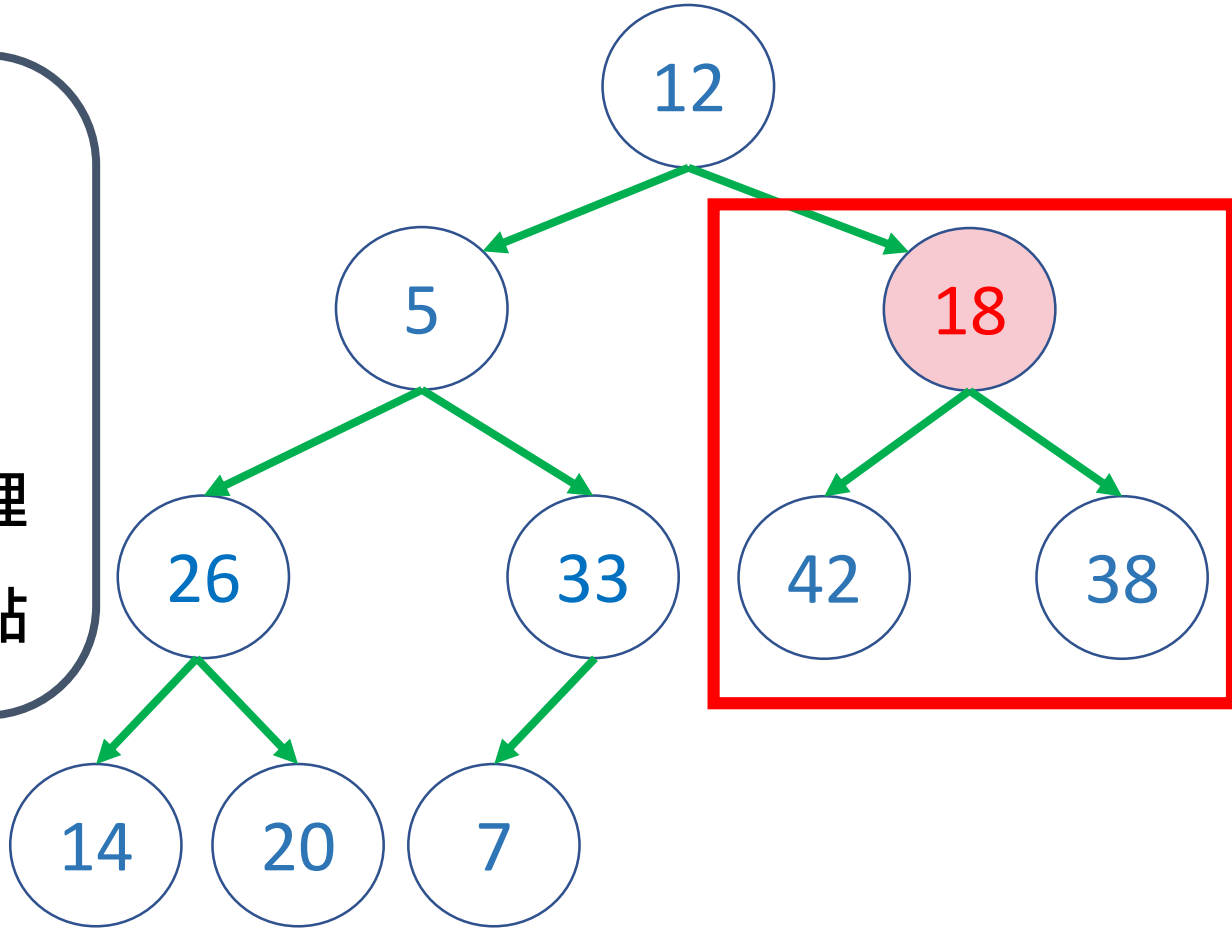


12	5	18	26	33	42	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

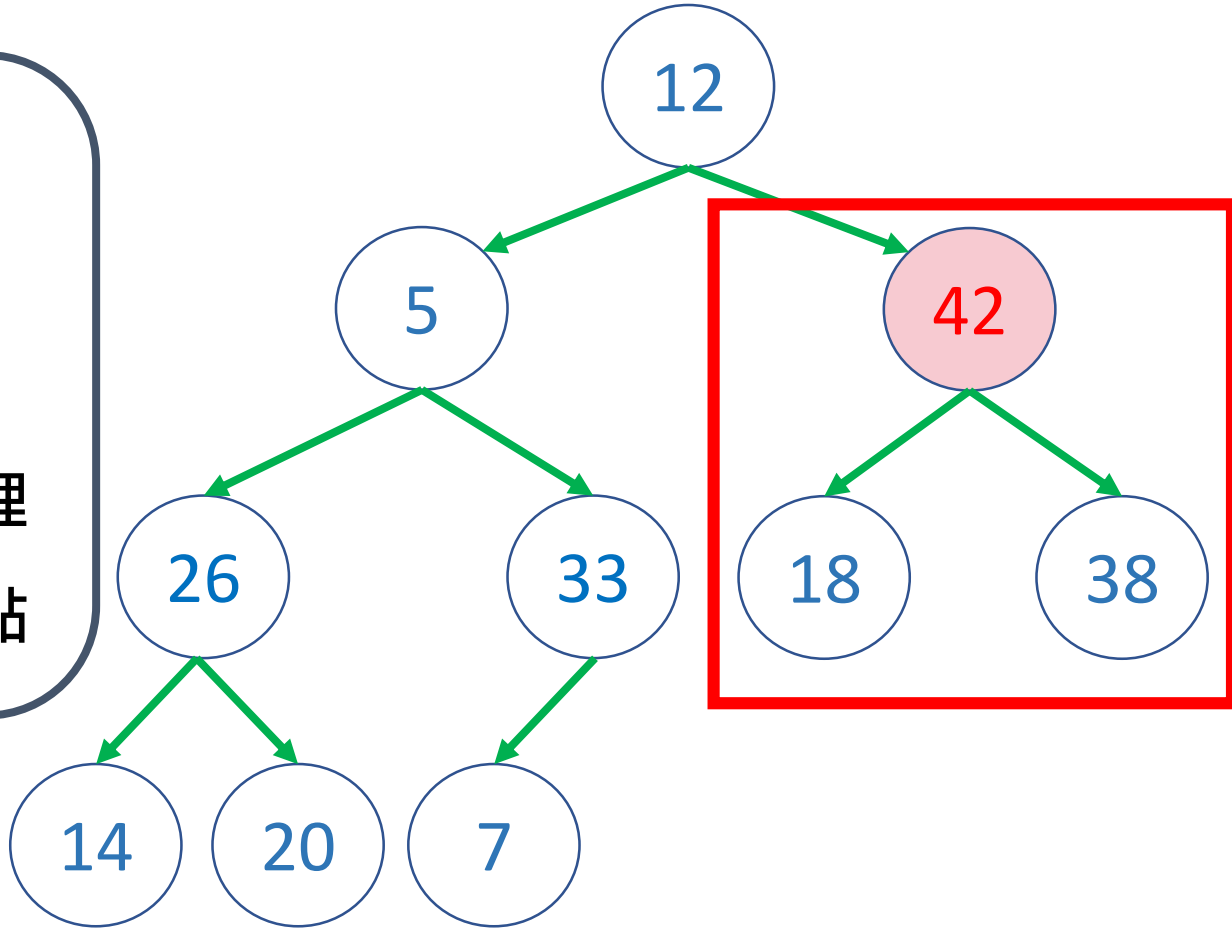


12	5	18	26	33	42	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

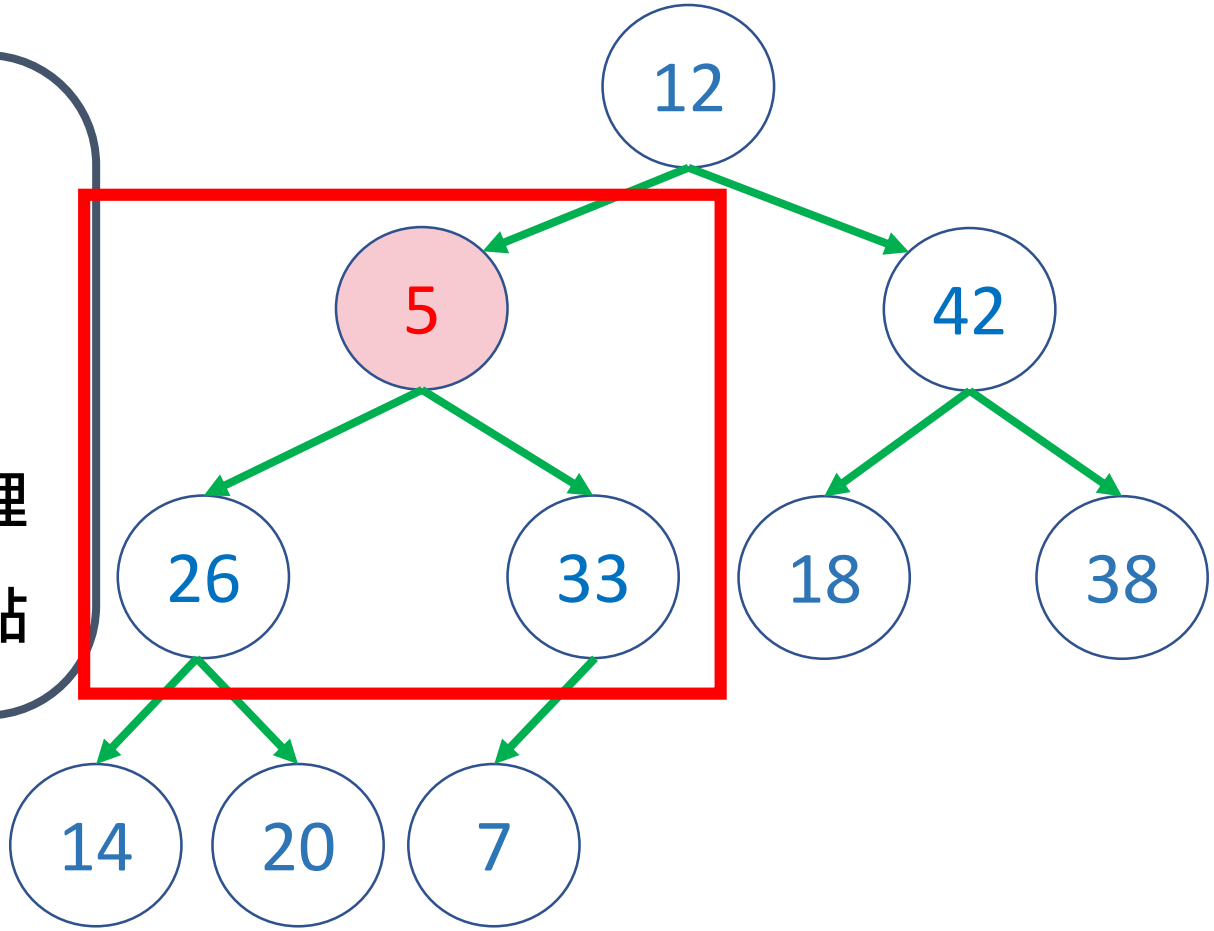


12	5	42	26	33	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

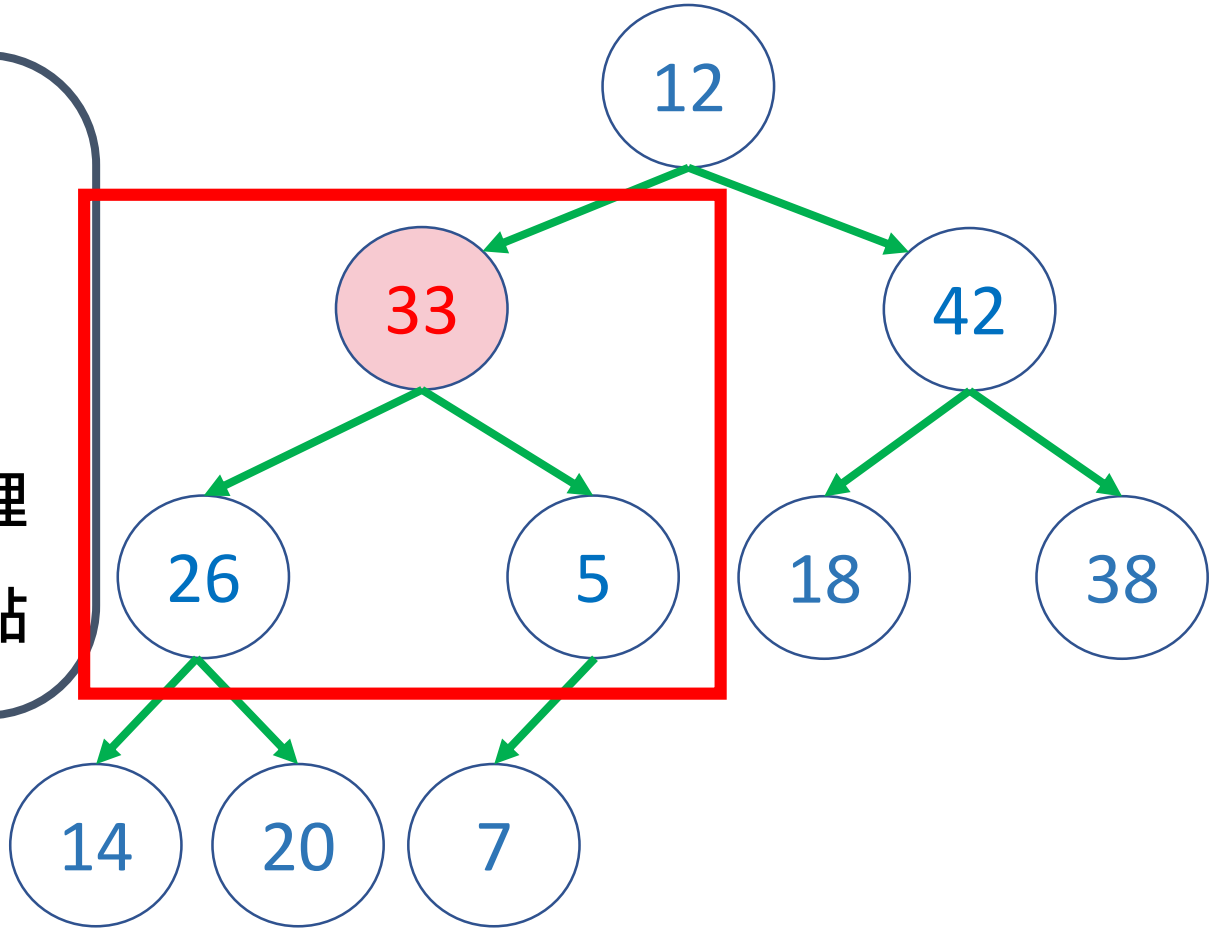


12	5	42	26	33	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

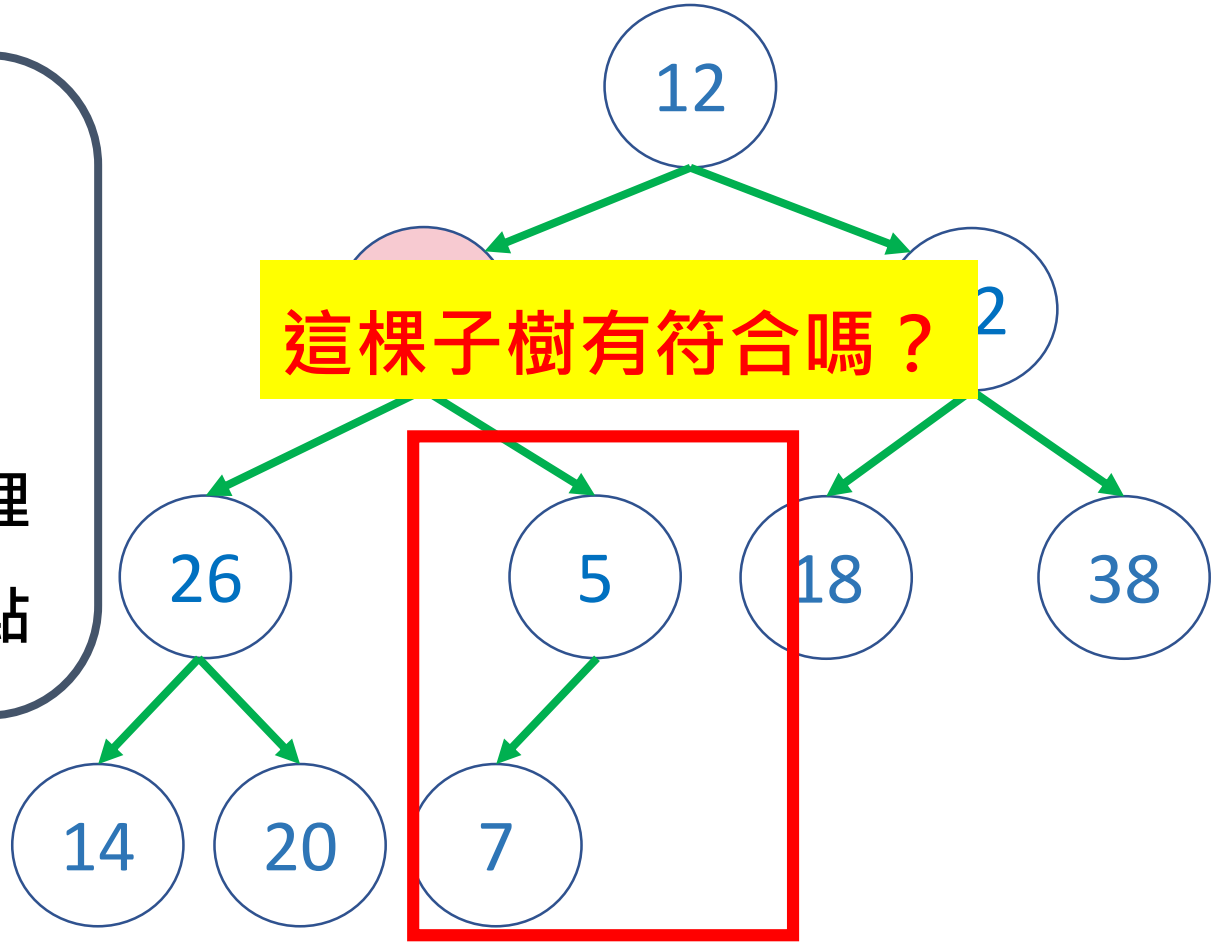


12	33	42	26	5	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

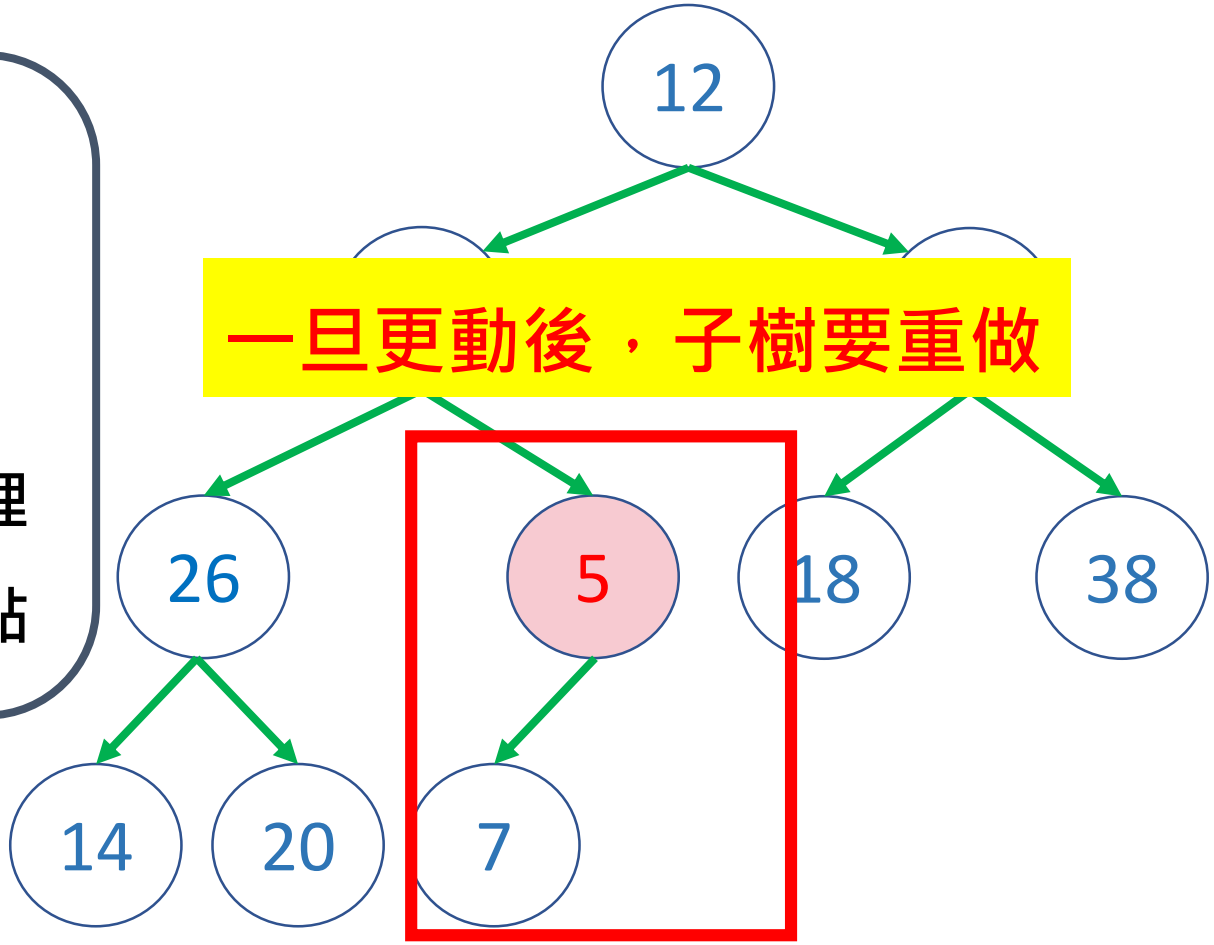


12	33	42	26	5	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

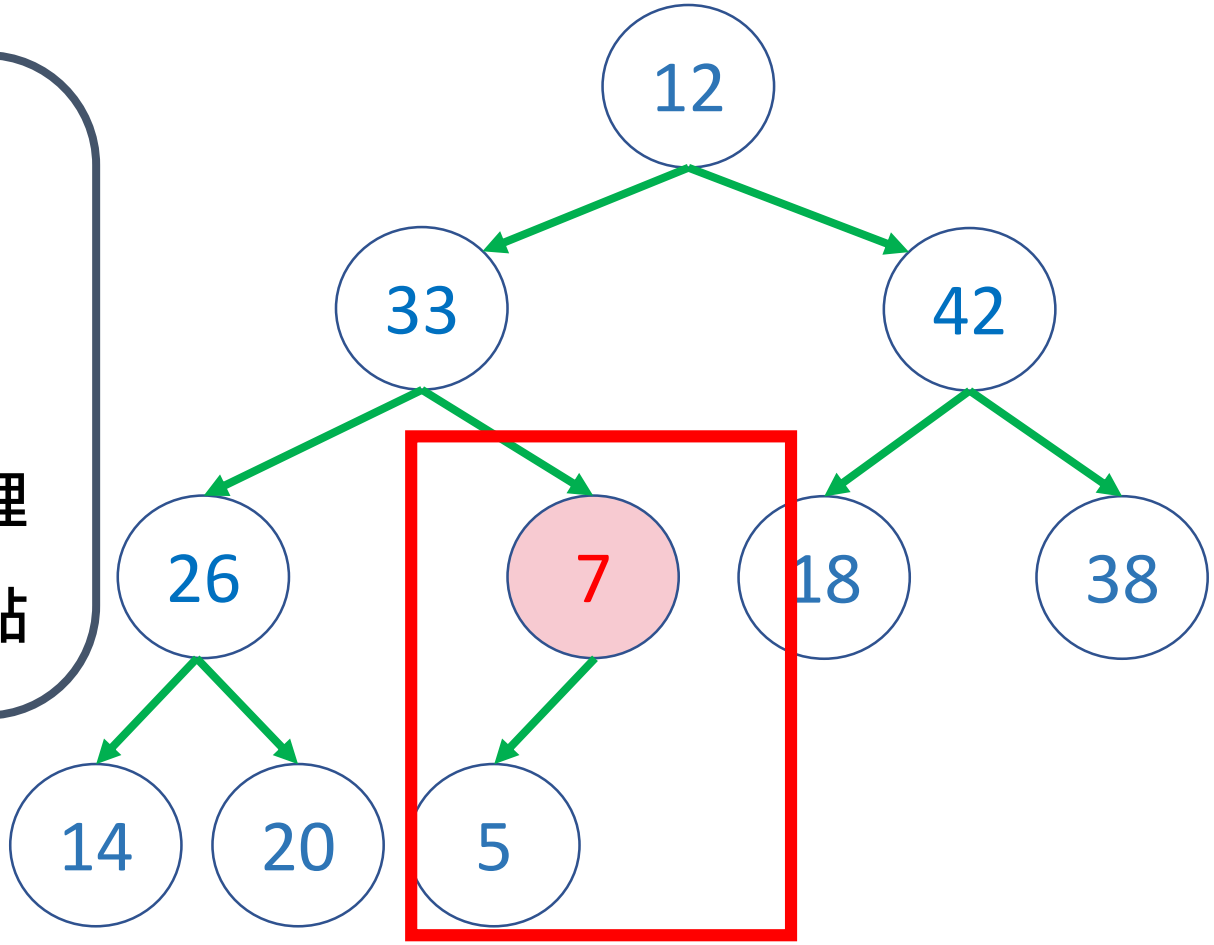


12	33	42	26	5	18	38	14	20	7

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

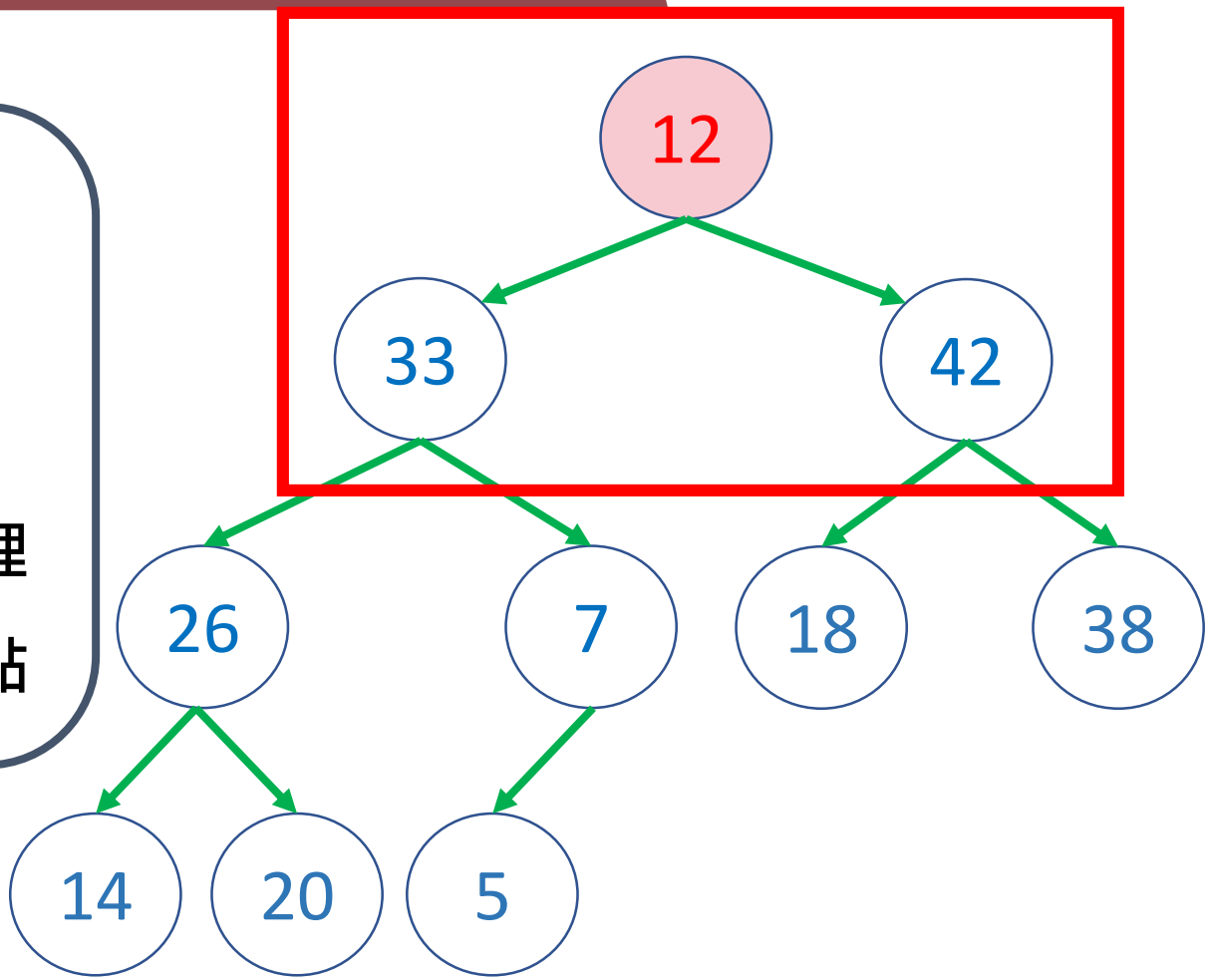


12	33	42	26	7	18	38	14	20	5

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

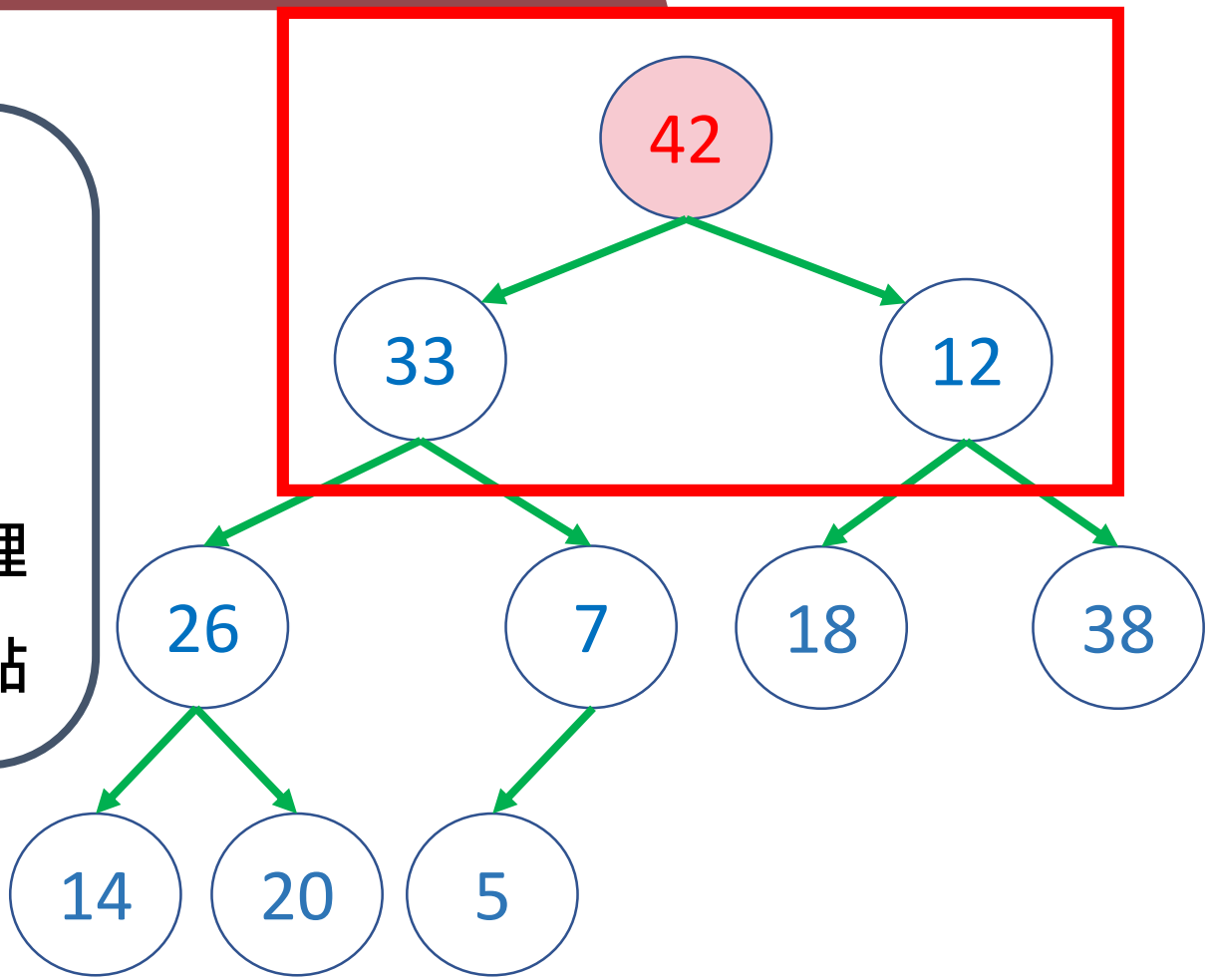


12	33	42	26	7	18	38	14	20	5

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

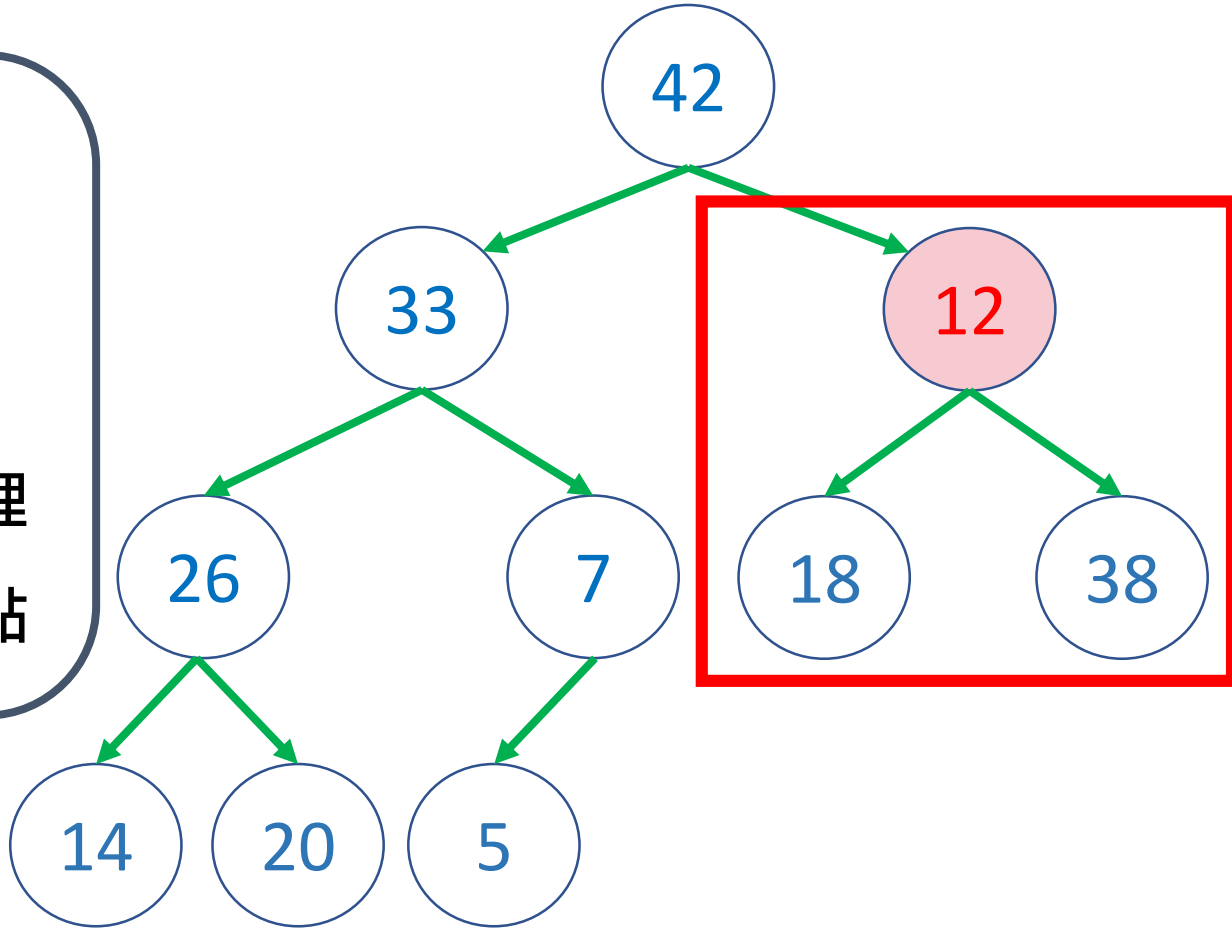


42	33	12	26	7	18	38	14	20	5

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點

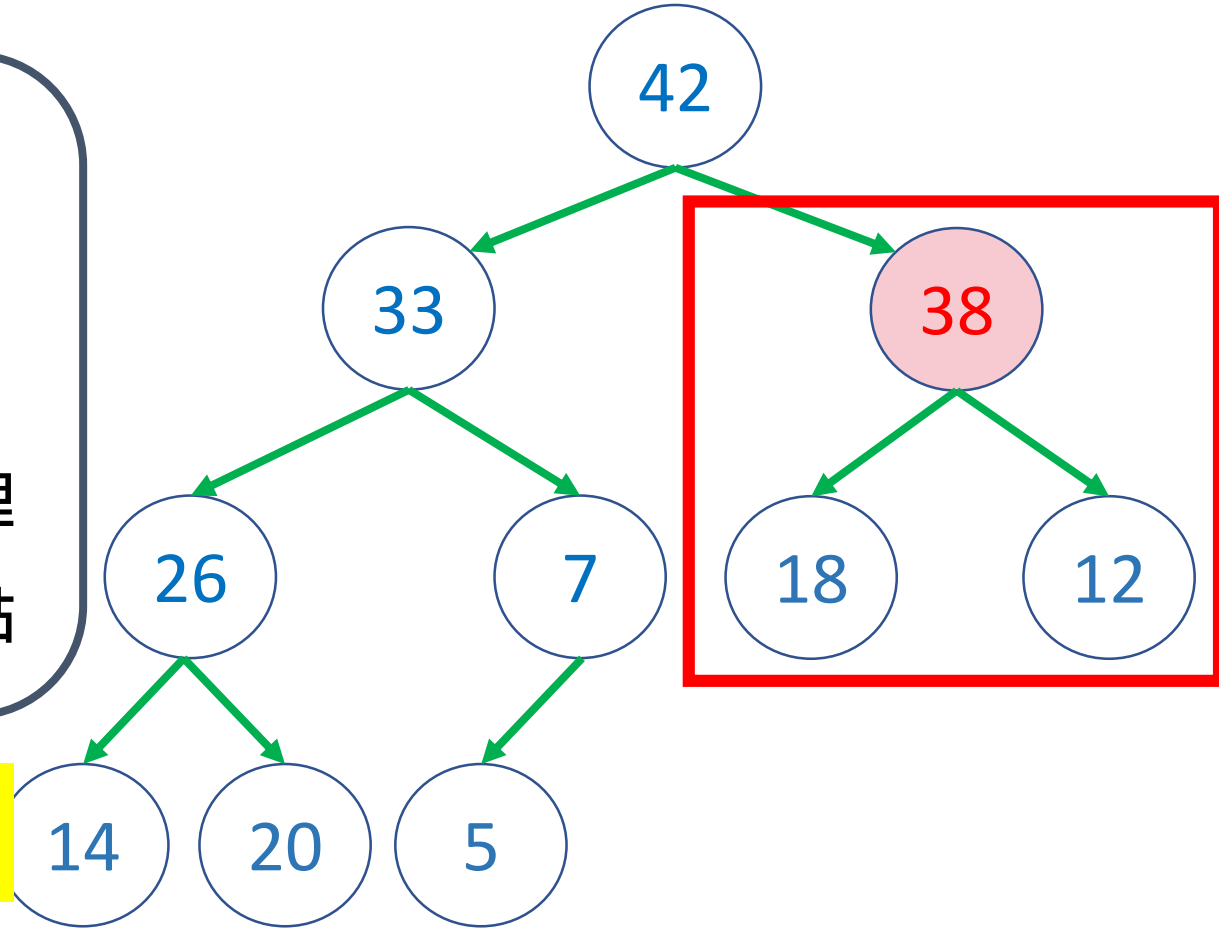


42	33	12	26	7	18	38	14	20	5

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點



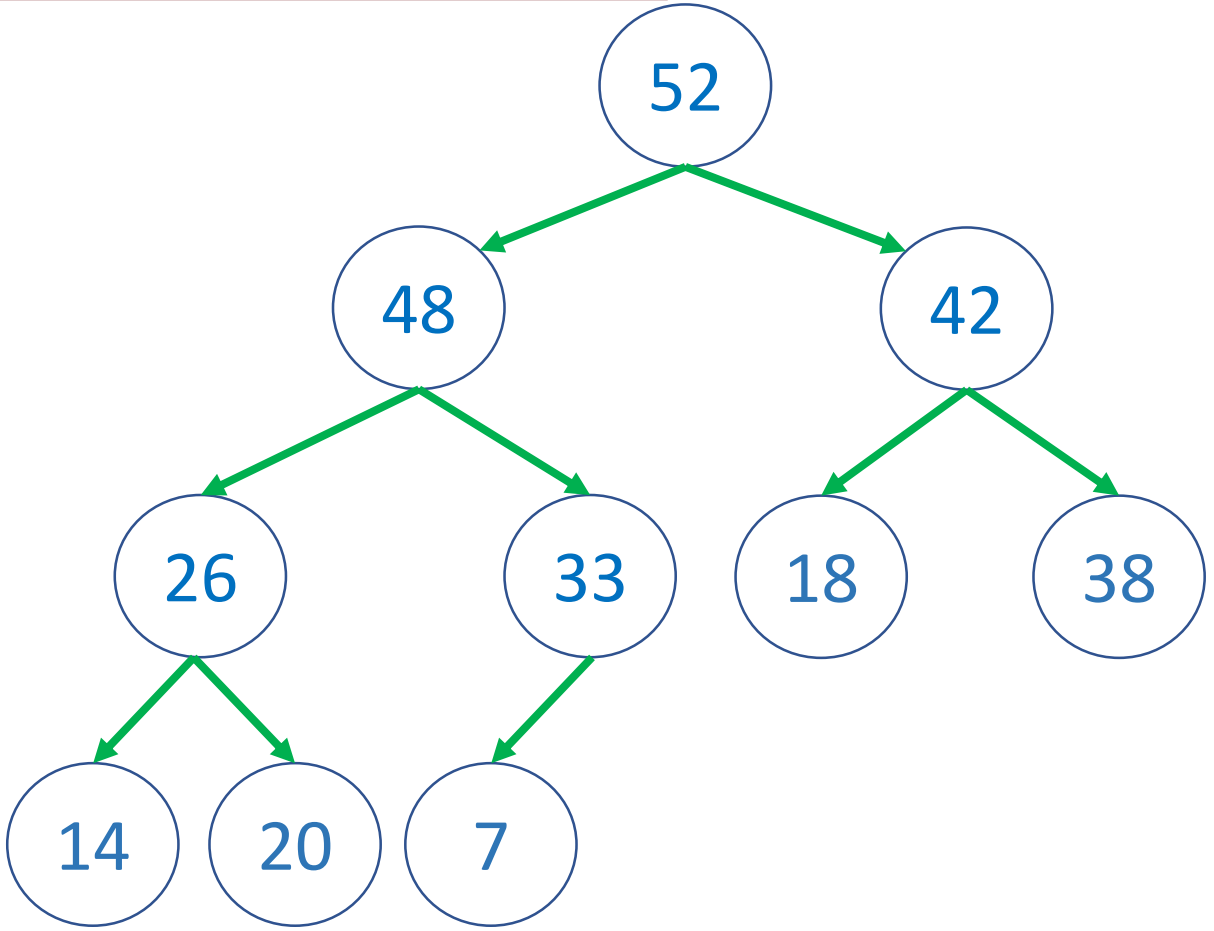
根節點就是整棵樹/陣列中的最大值！

42	33	38	26	7	18	12	14	20	5

二元堆疊

建立二元堆疊 (Binary Heap)

- 給定一陣列長度為 n
  - 末端元素索引值：$n-1$
  - 該元素的父節點索引值：$\left\lfloor \frac{n-2}{2} \right\rfloor$
  - Heapify 複雜度
    - ✓ $O(log_2 n)$
  - 建立二元堆疊複雜度：
    - ✓ $O(nlog_2 n)$



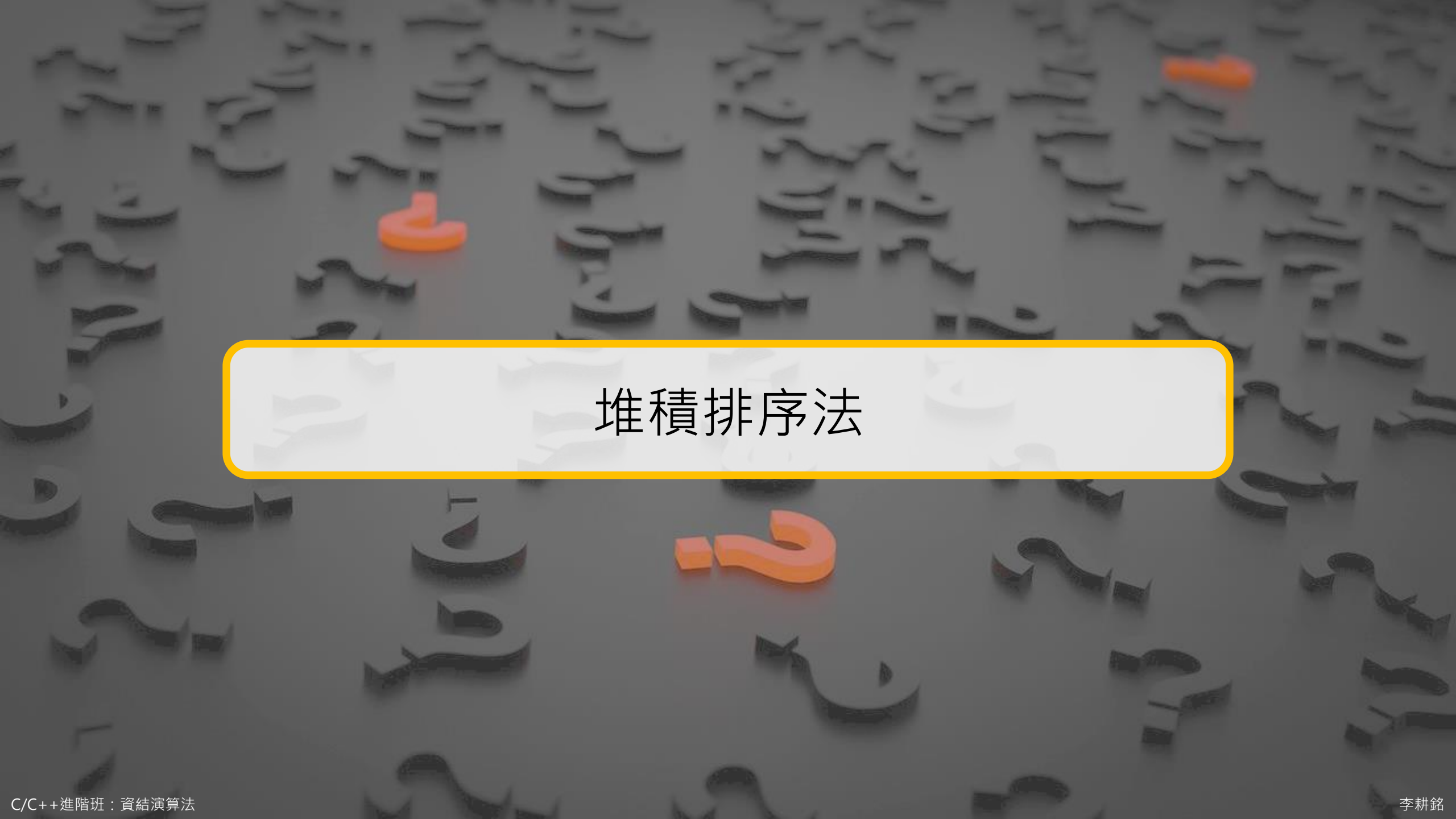

```
for(int i = len/2-1; i>=0 ; i--){
      Heapify(i);
}
```

Example Code

Mission

利用以下陣列實作 heapify，包含 min-heap 與 max-heap

15	18	6	25	8	11	34	20	2	38

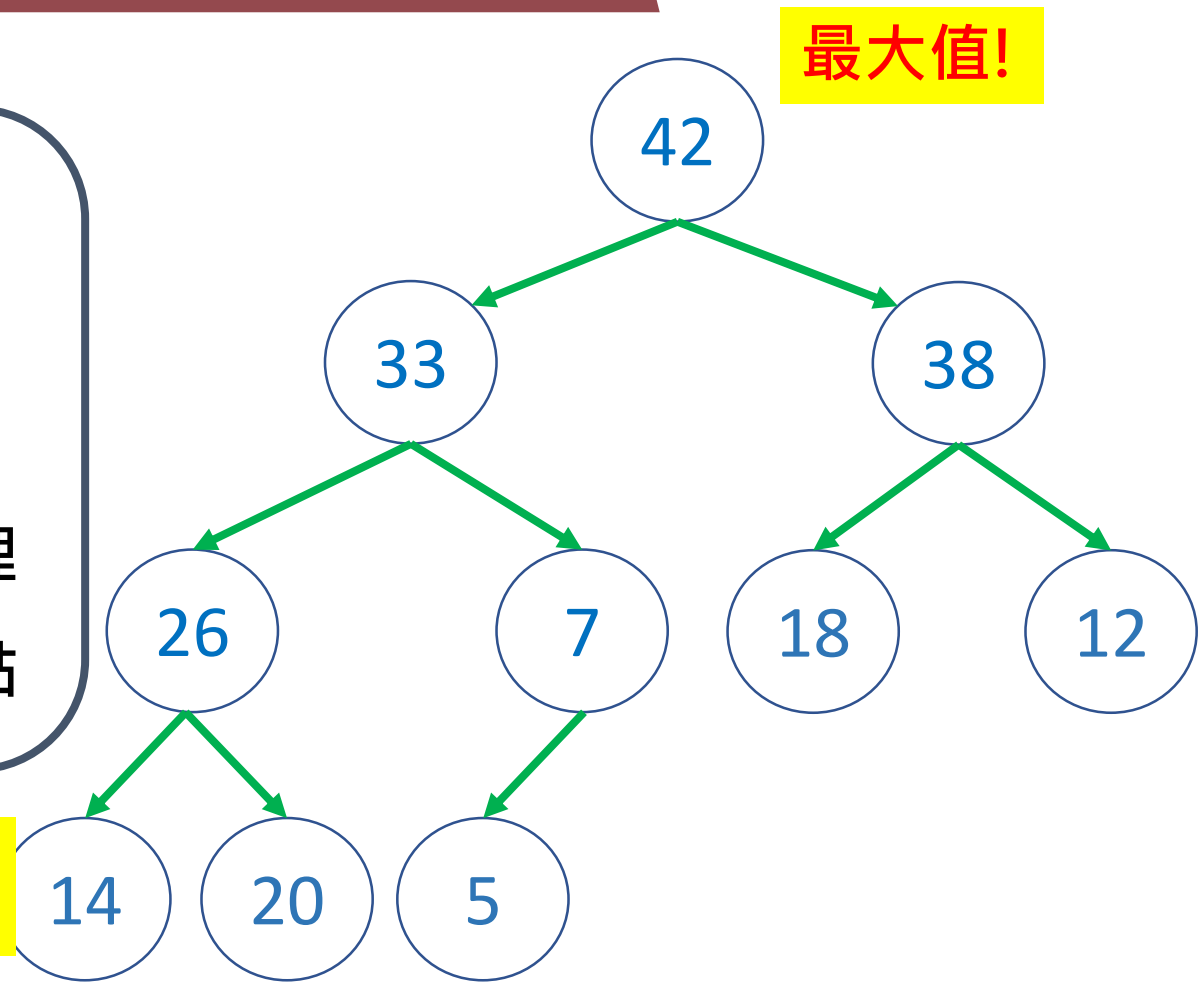


堆積排序法

二元堆疊

建立最大二元堆疊 (Max-Heap)

- 對具有子節點的節點進行
  - 沒有子節點的話本身就會符合規則
  - 從最下面且具有子節點的節點向上處理
  - 目前節點、左右節點中最大的為根節點



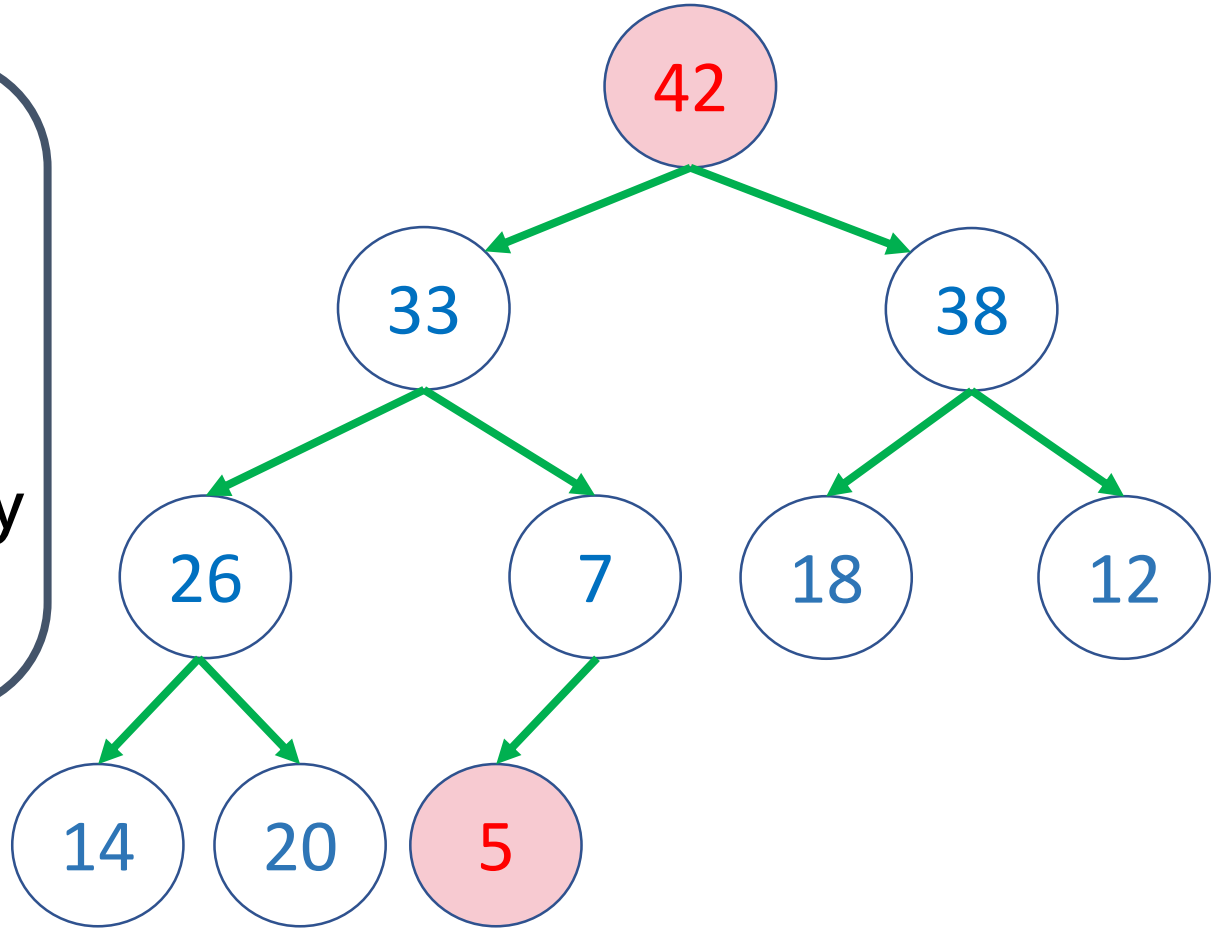
根節點就是整棵樹/陣列中的最大值！

42	33	38	26	7	18	12	14	20	5

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

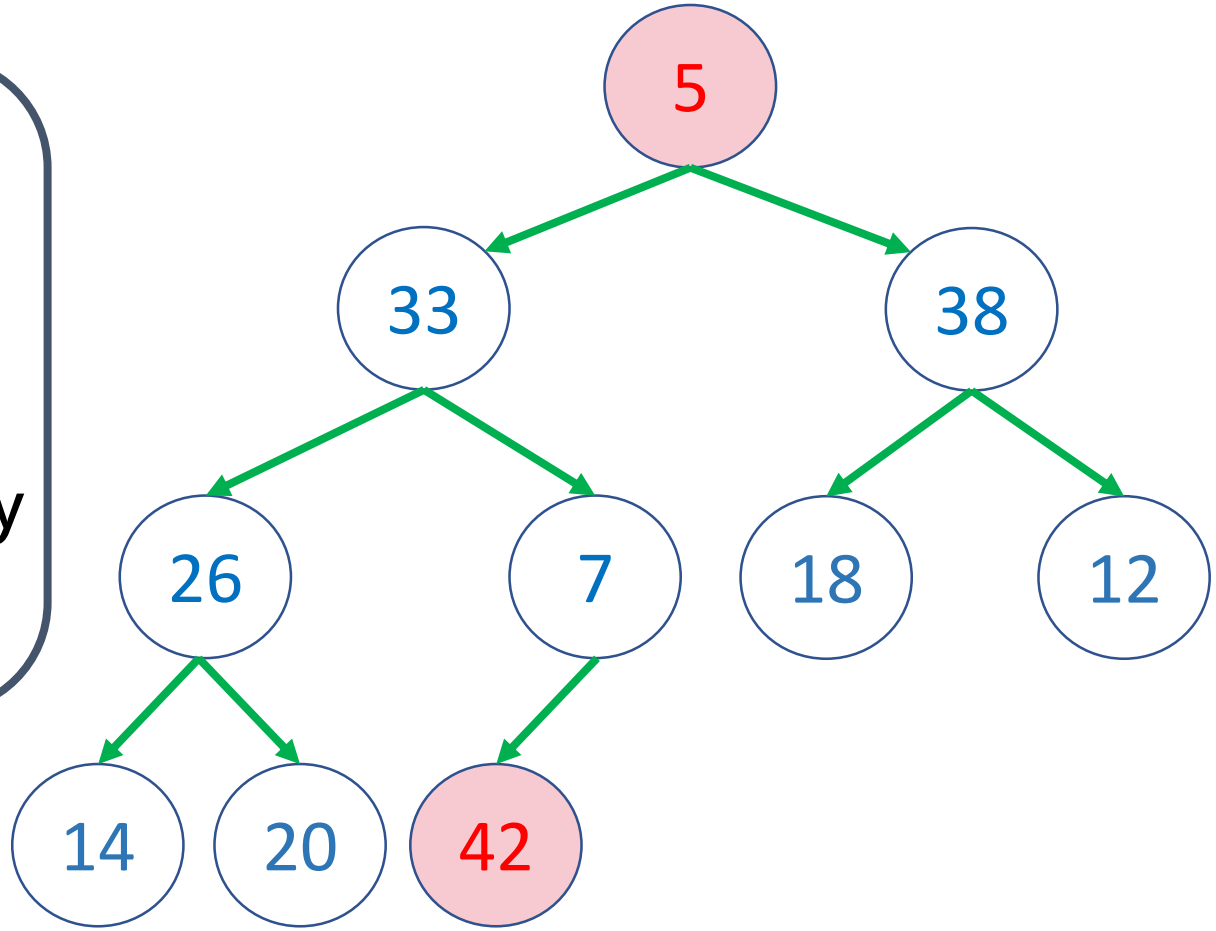


42	33	38	26	7	18	12	14	20	5

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

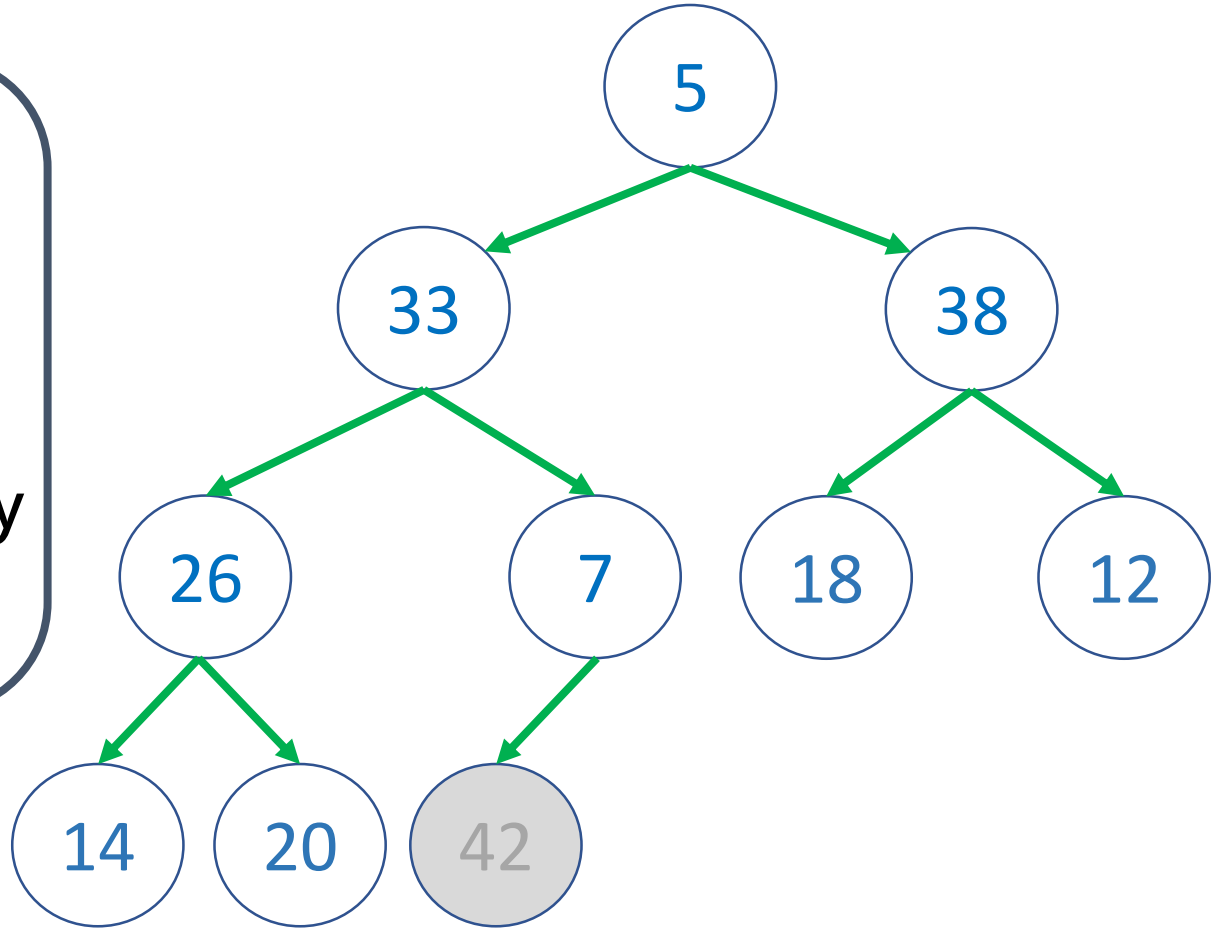


5	33	38	26	7	18	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

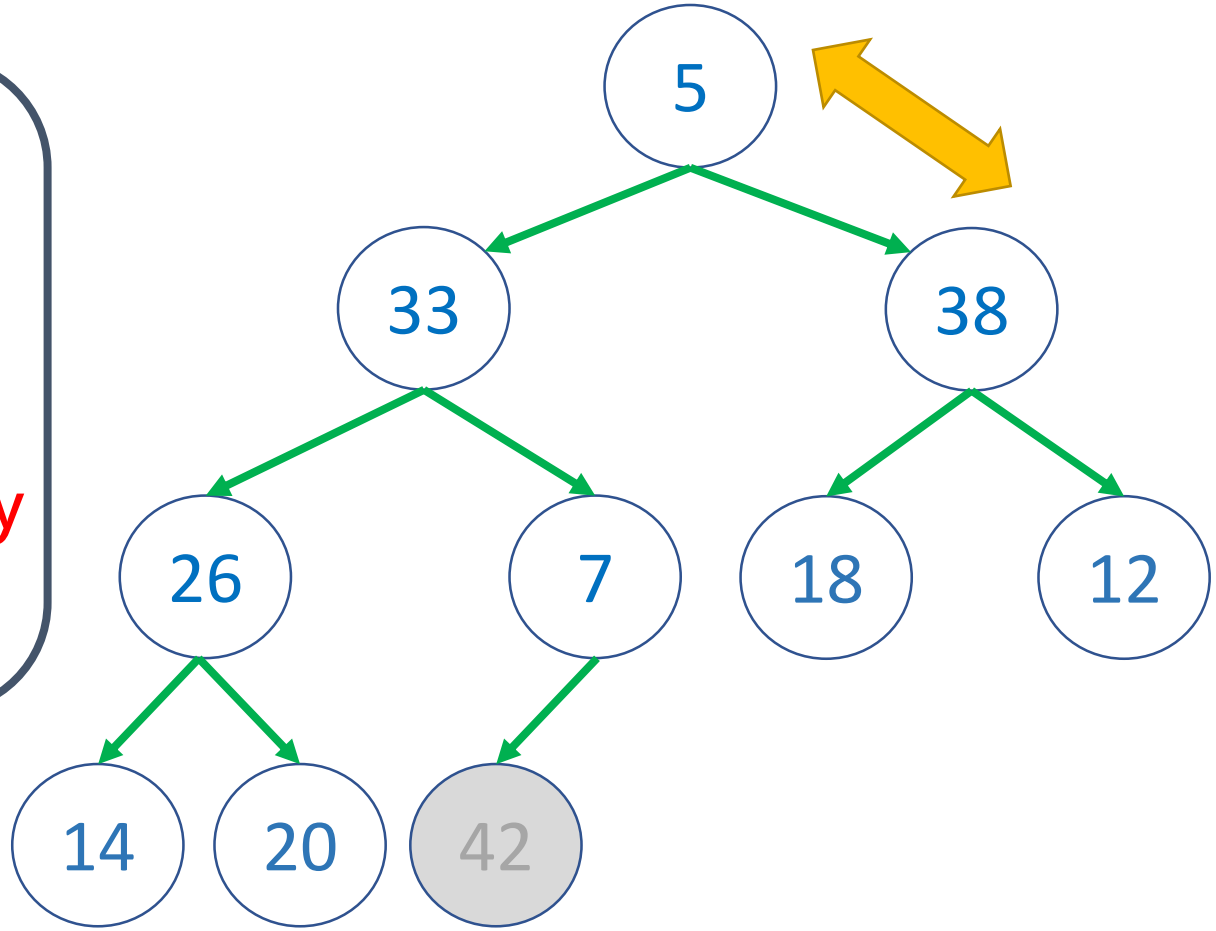


5	33	38	26	7	18	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

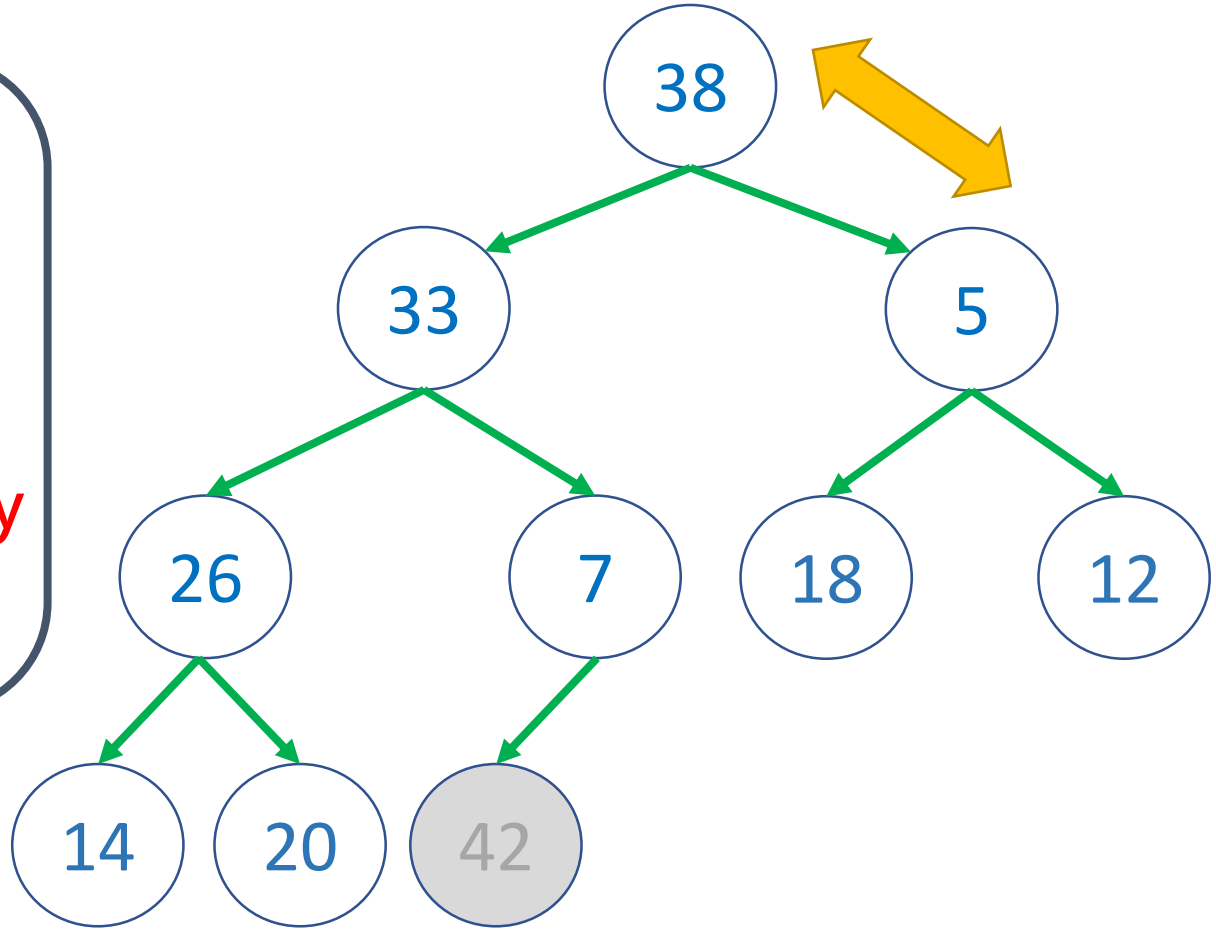


5	33	38	26	7	18	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

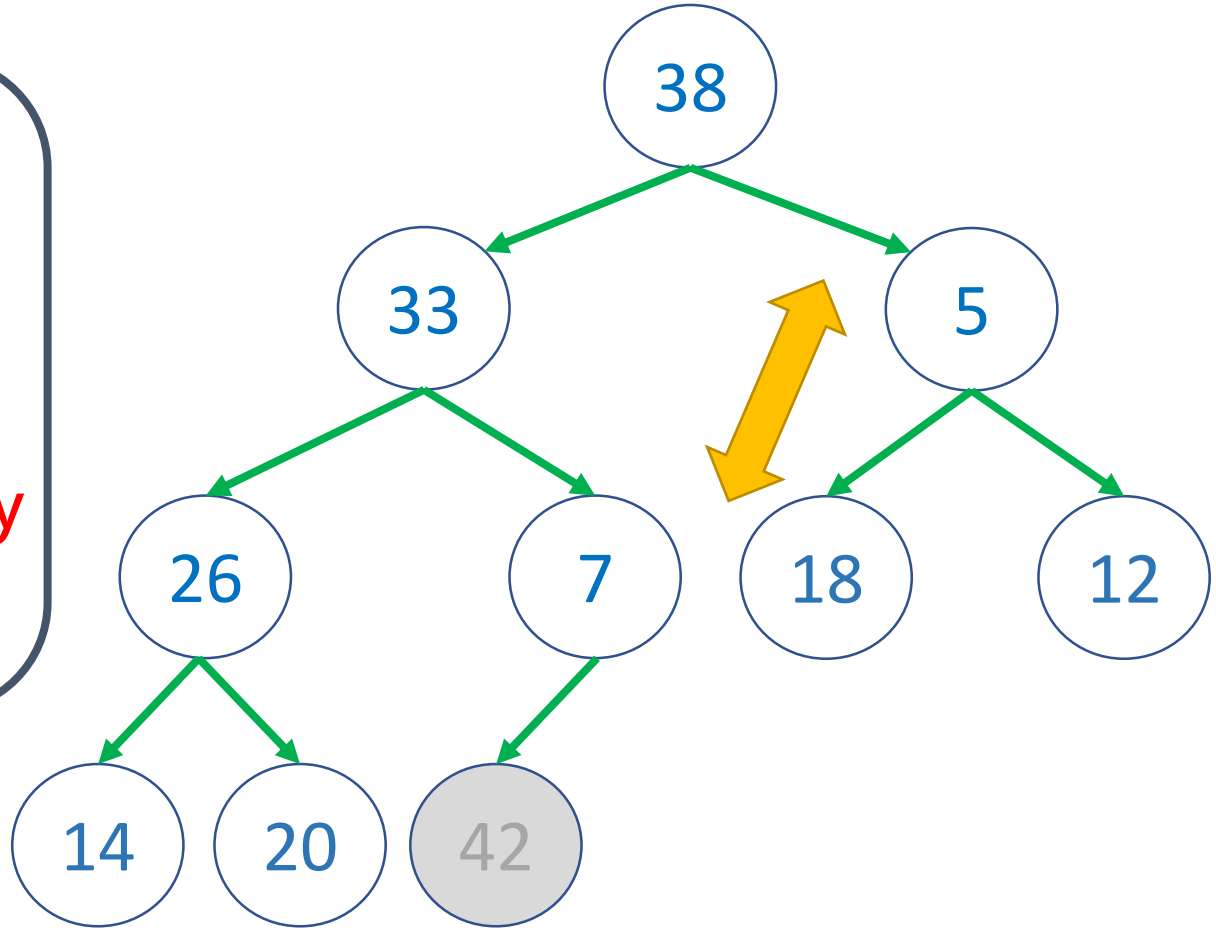


38	33	5	26	7	18	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

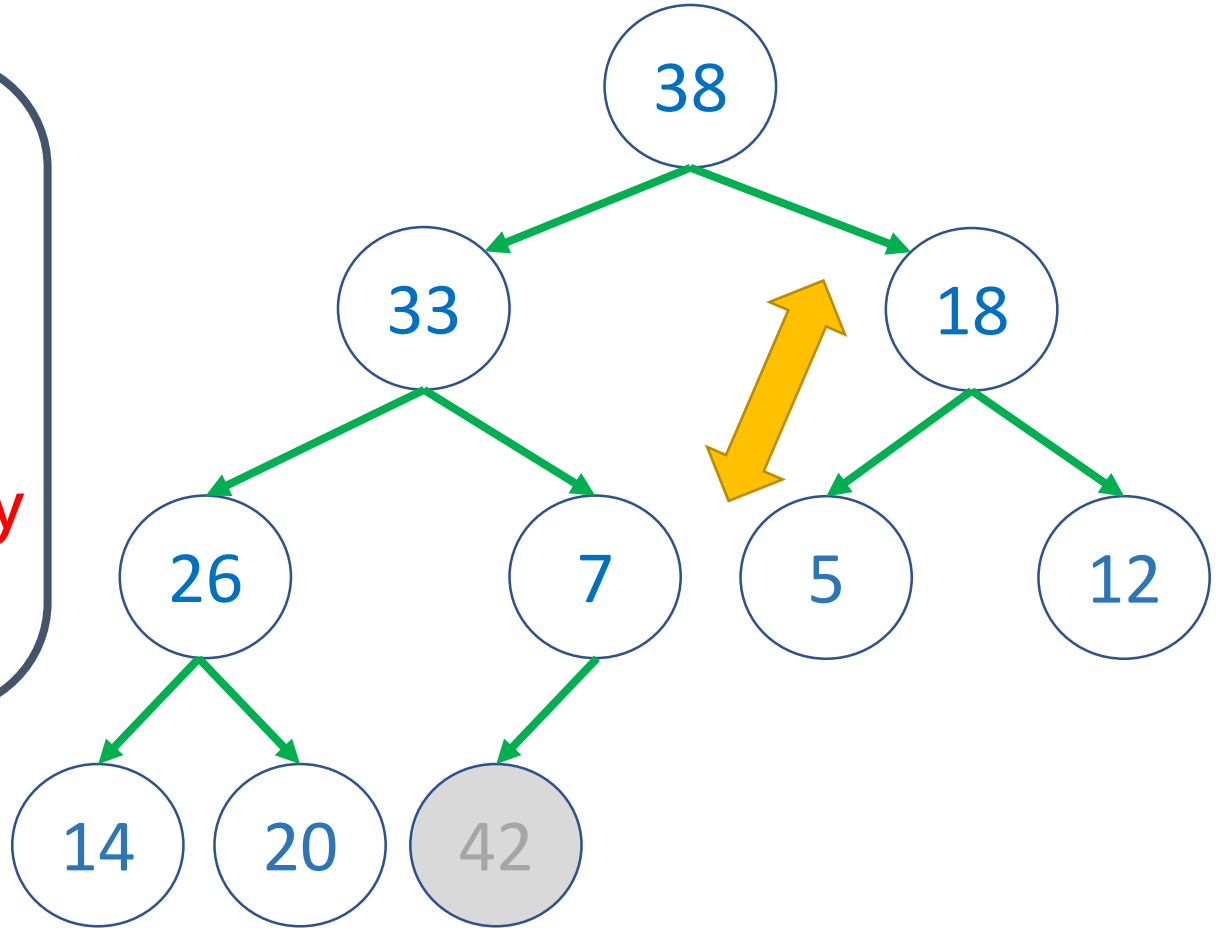


38	33	5	26	7	18	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

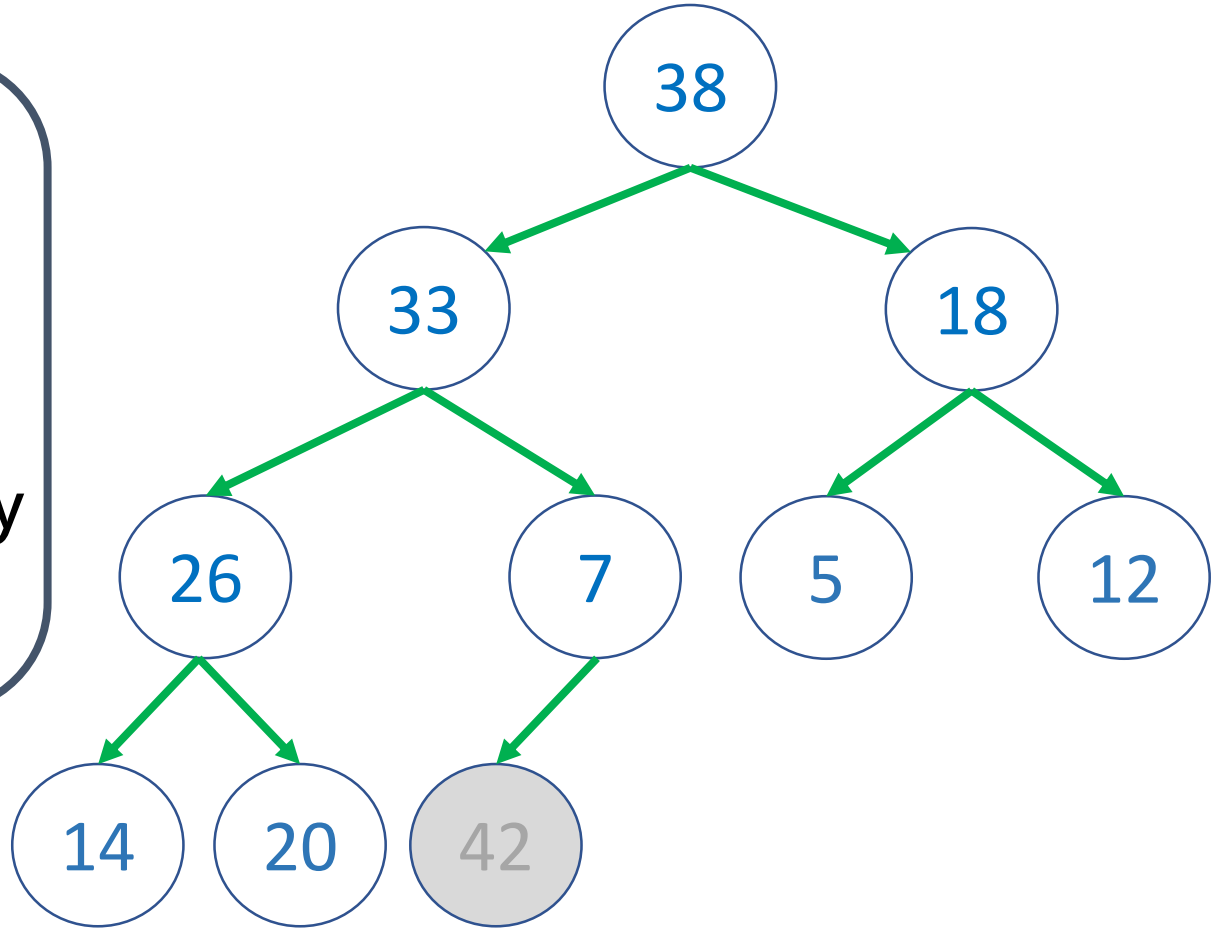


38	33	18	26	7	5	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

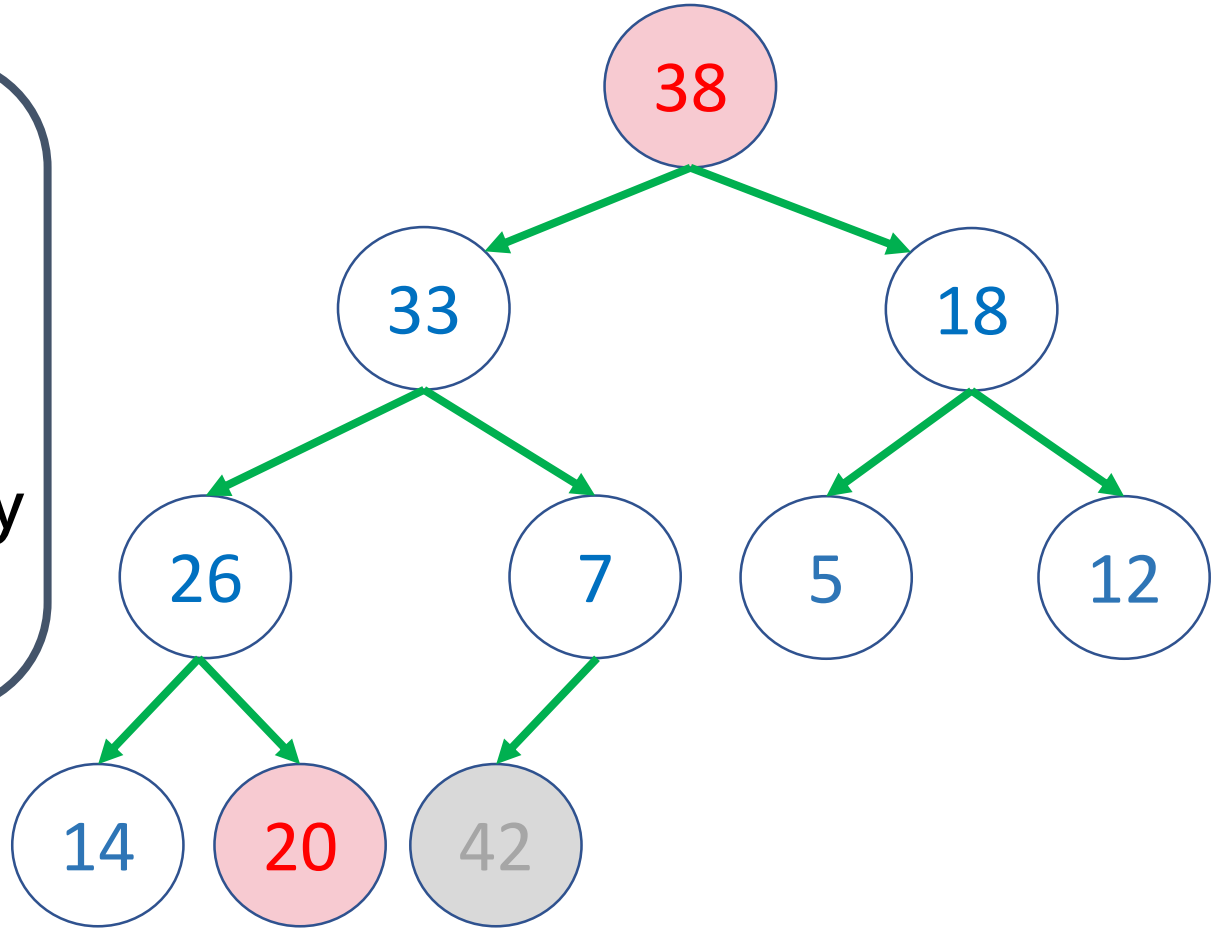


38	33	18	26	7	5	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

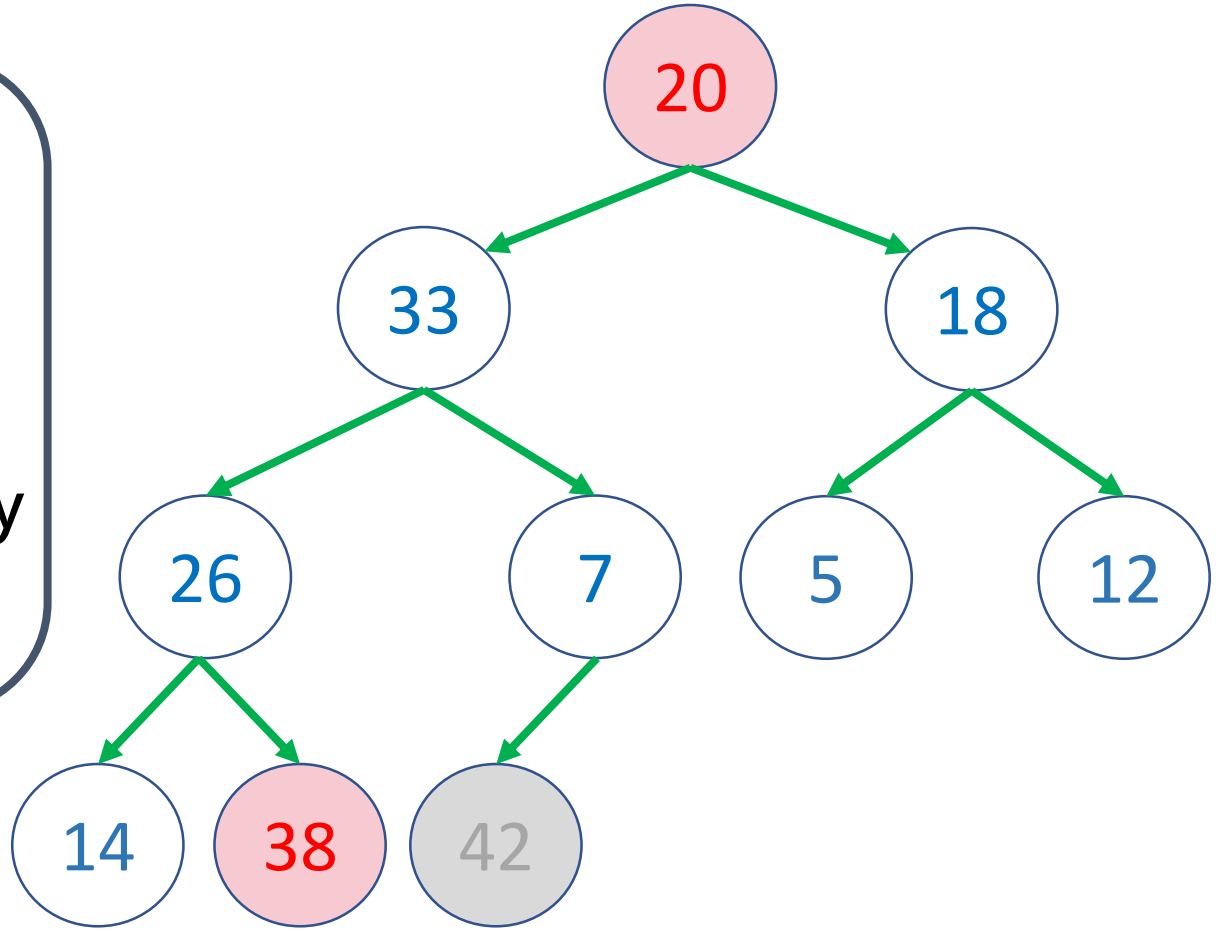


38	33	18	26	7	5	12	14	20	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

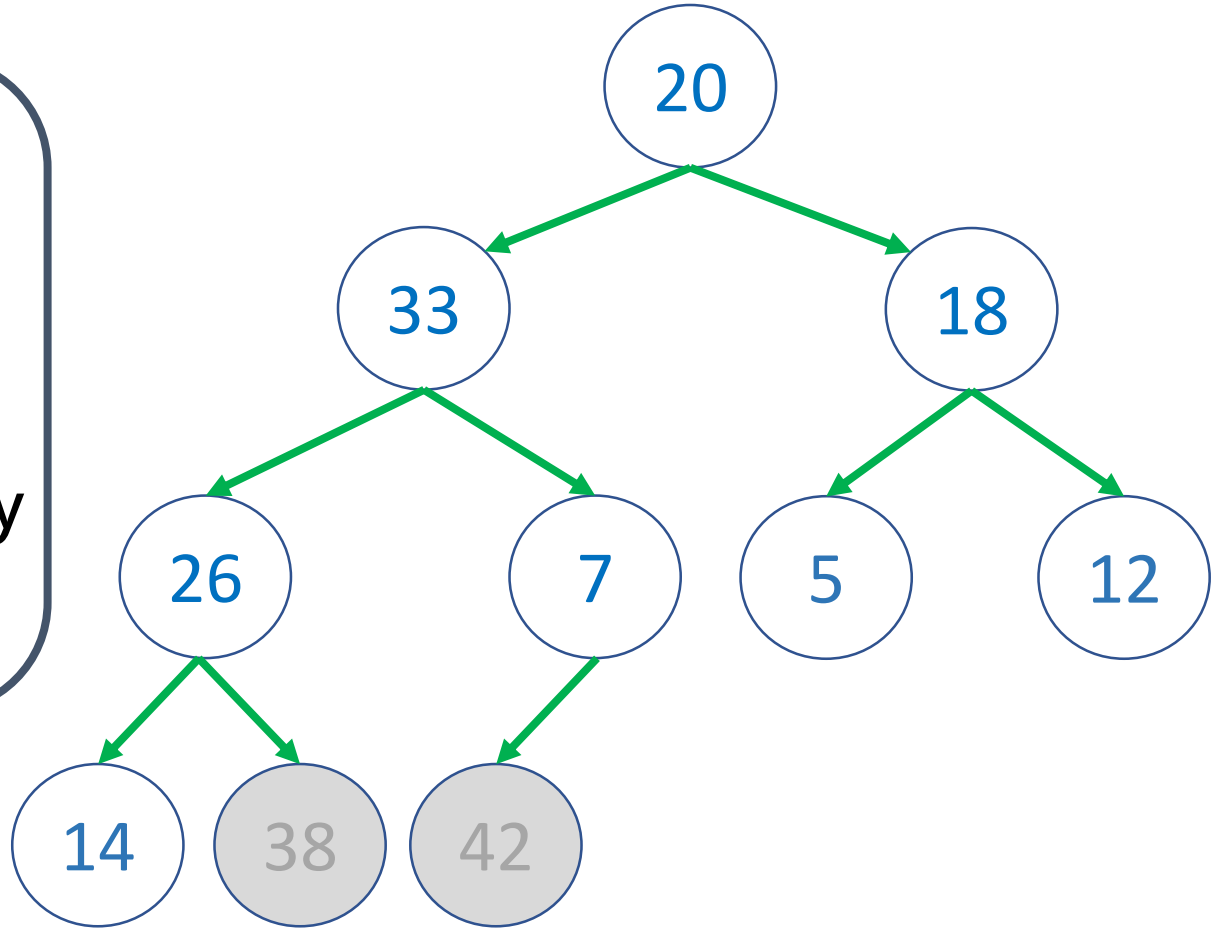


20	33	18	26	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

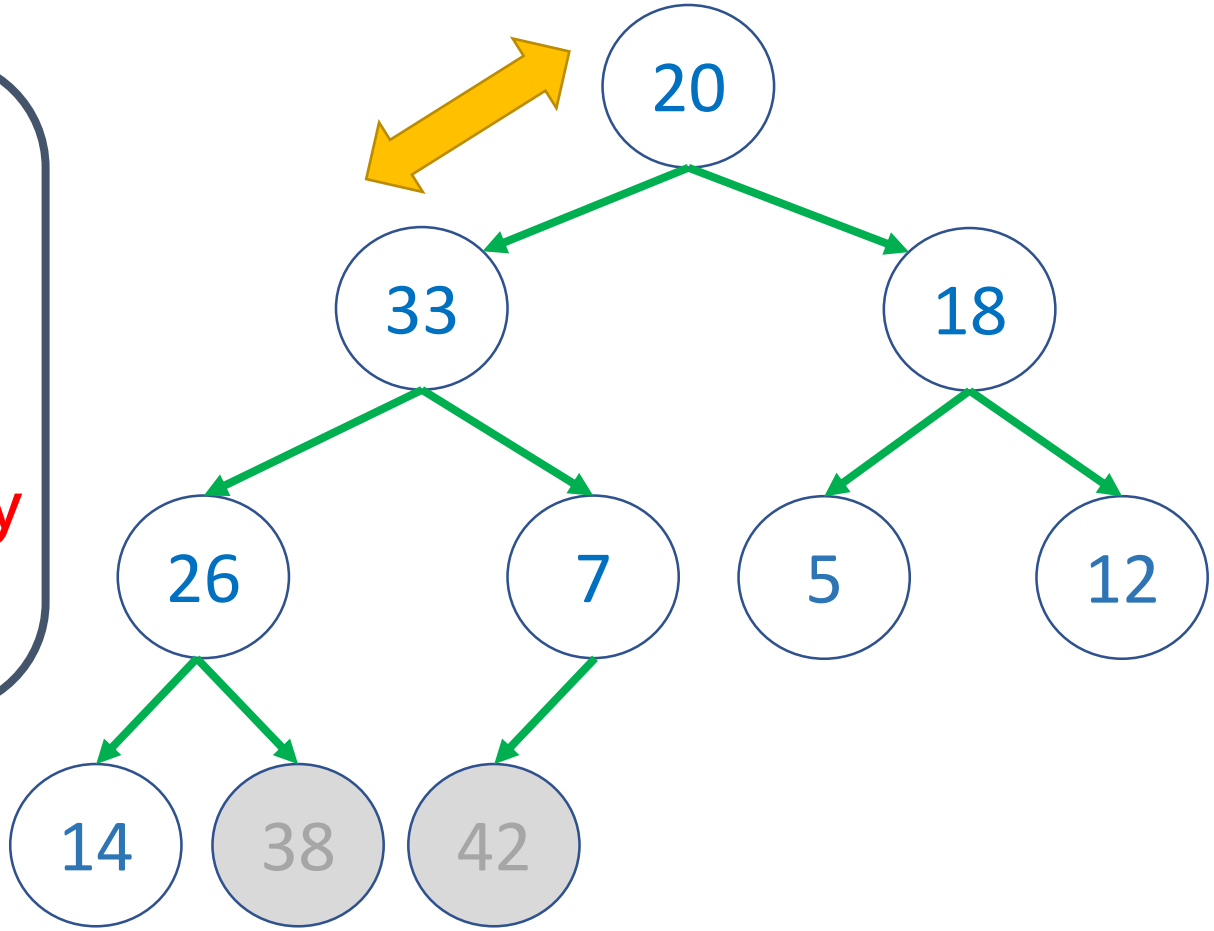


20	33	18	26	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

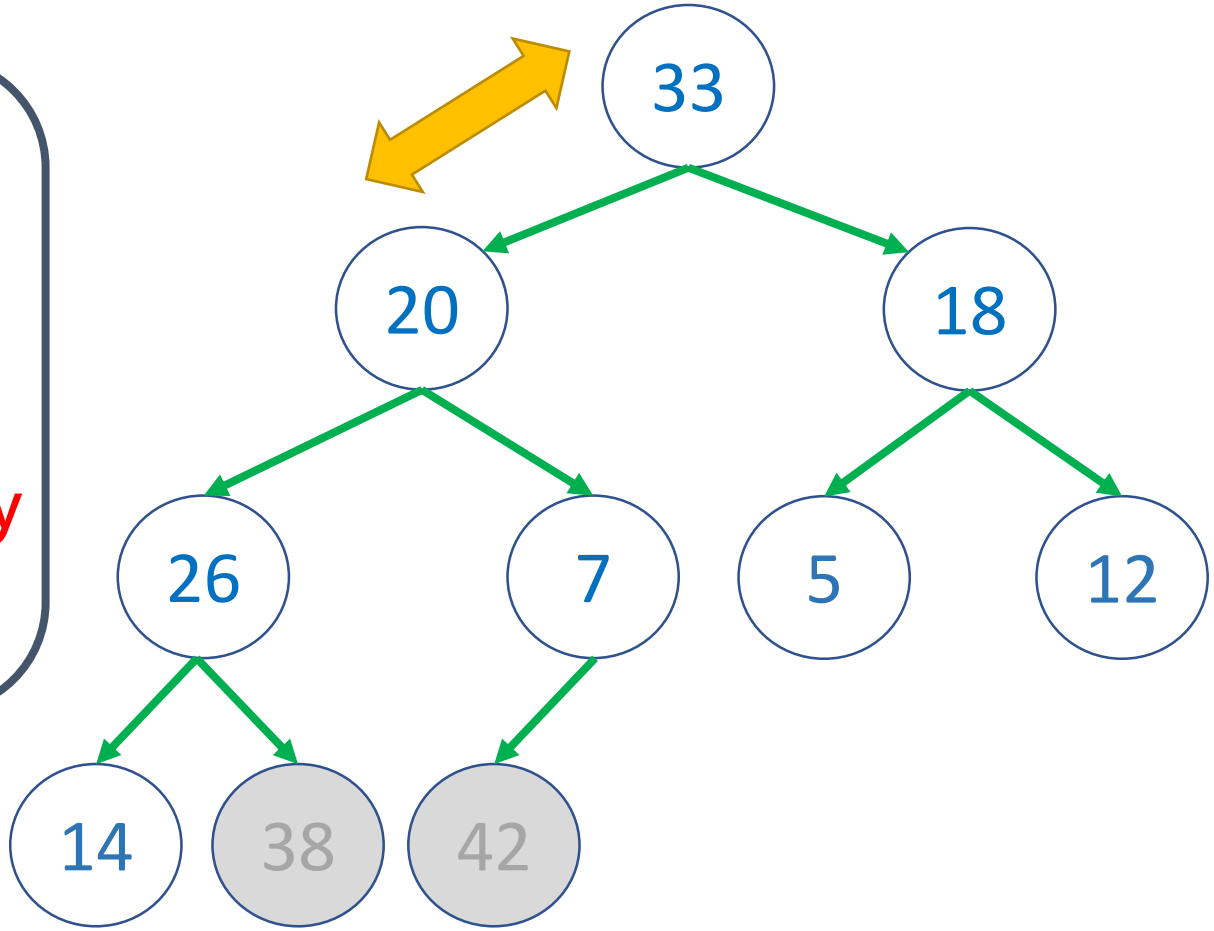


20	33	18	26	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

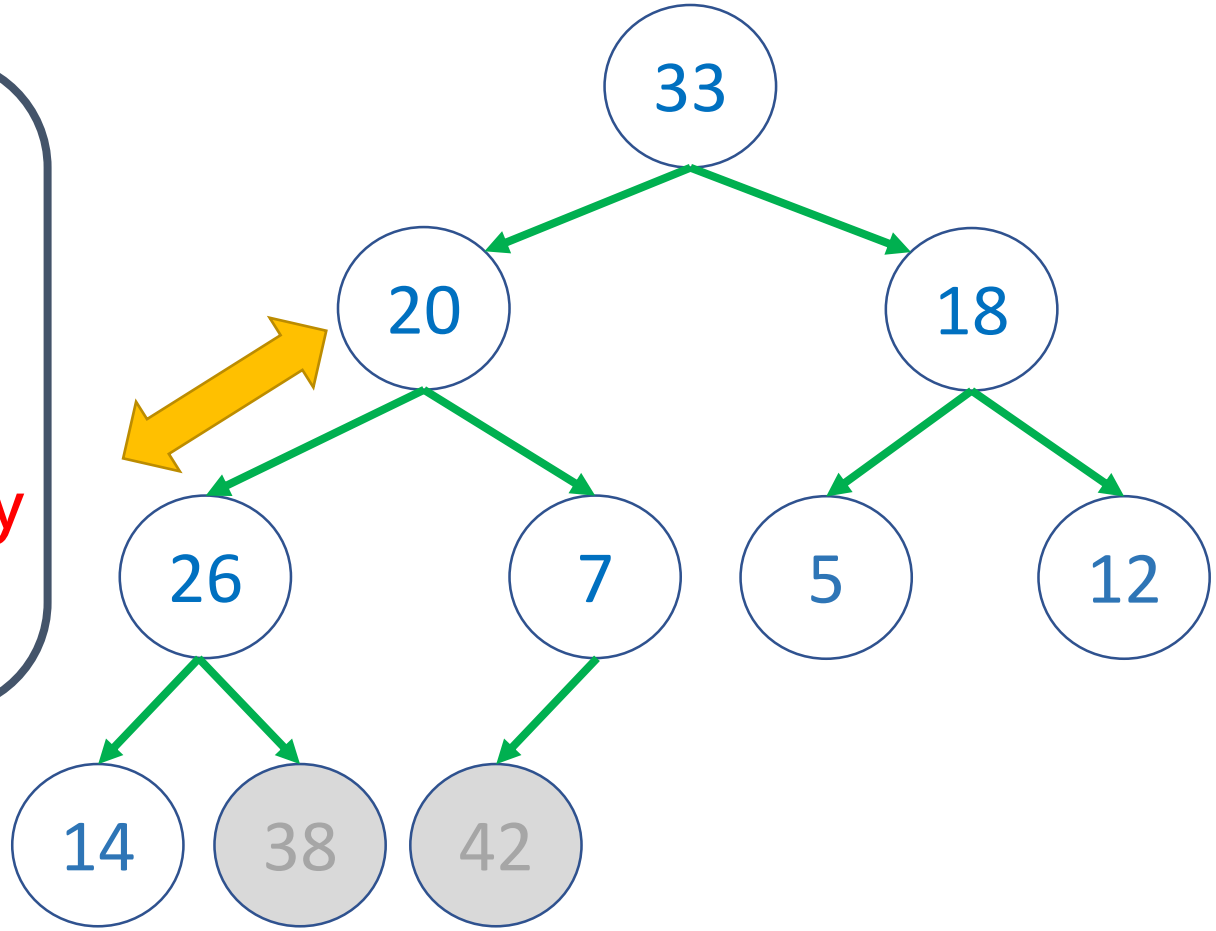


33	20	18	26	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

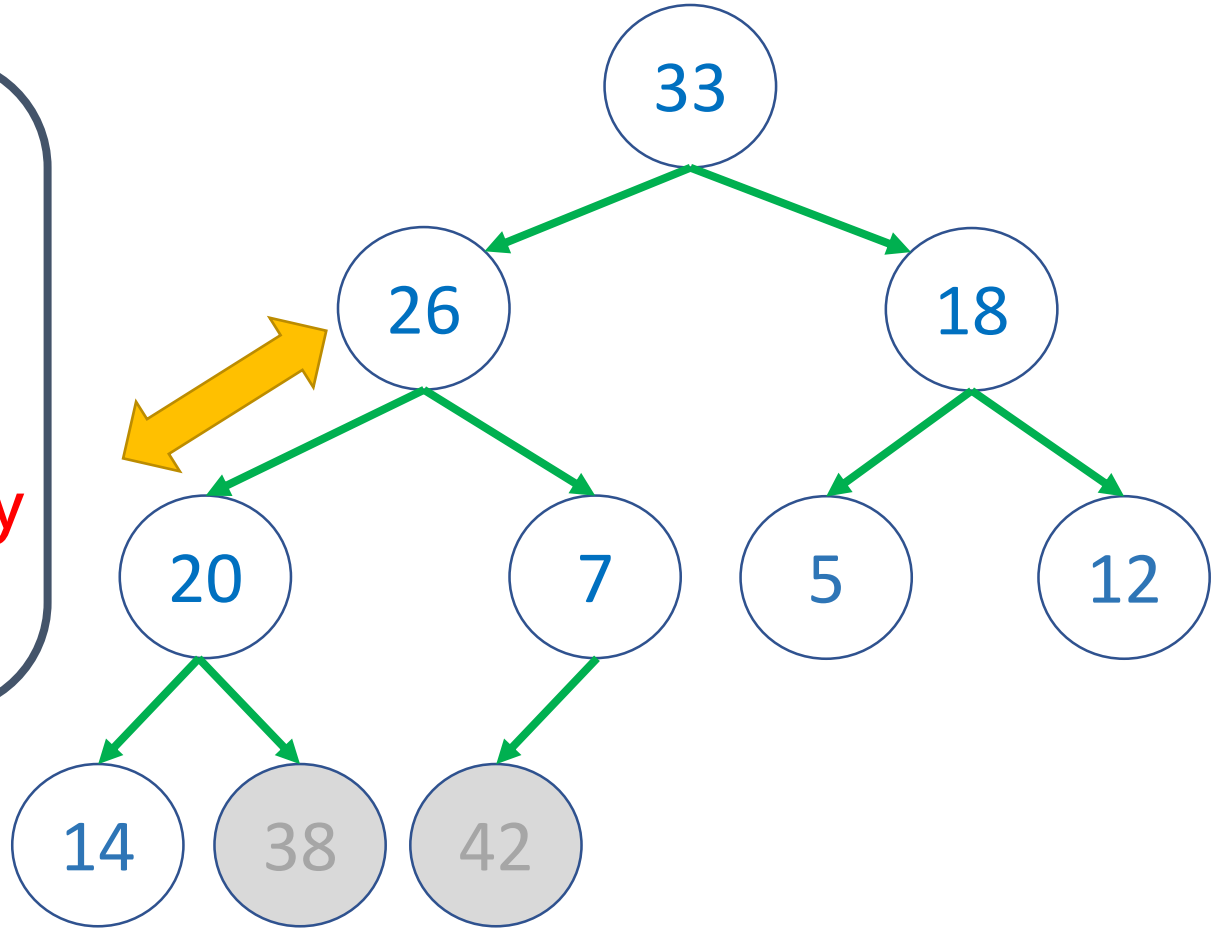


33	20	18	26	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

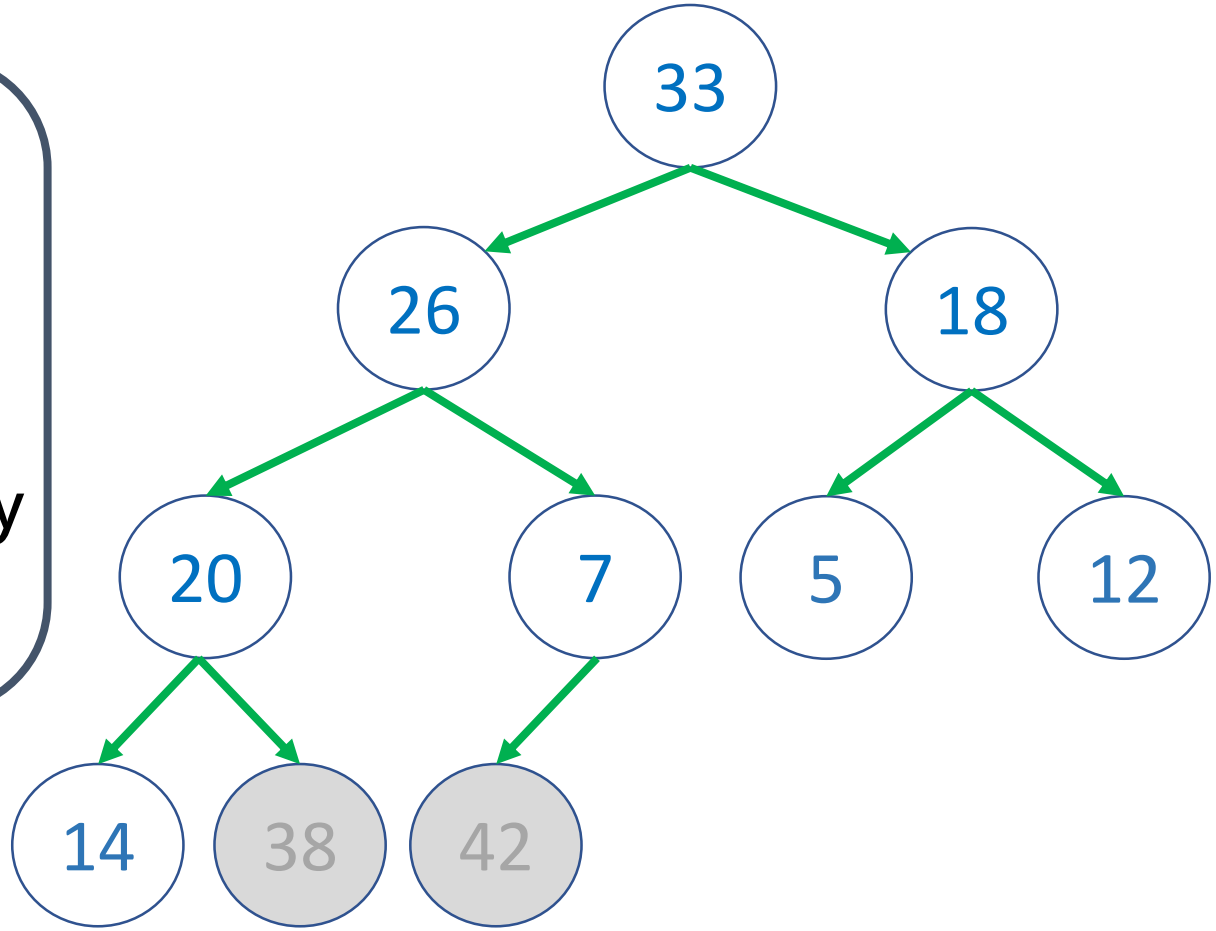


33	26	18	20	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

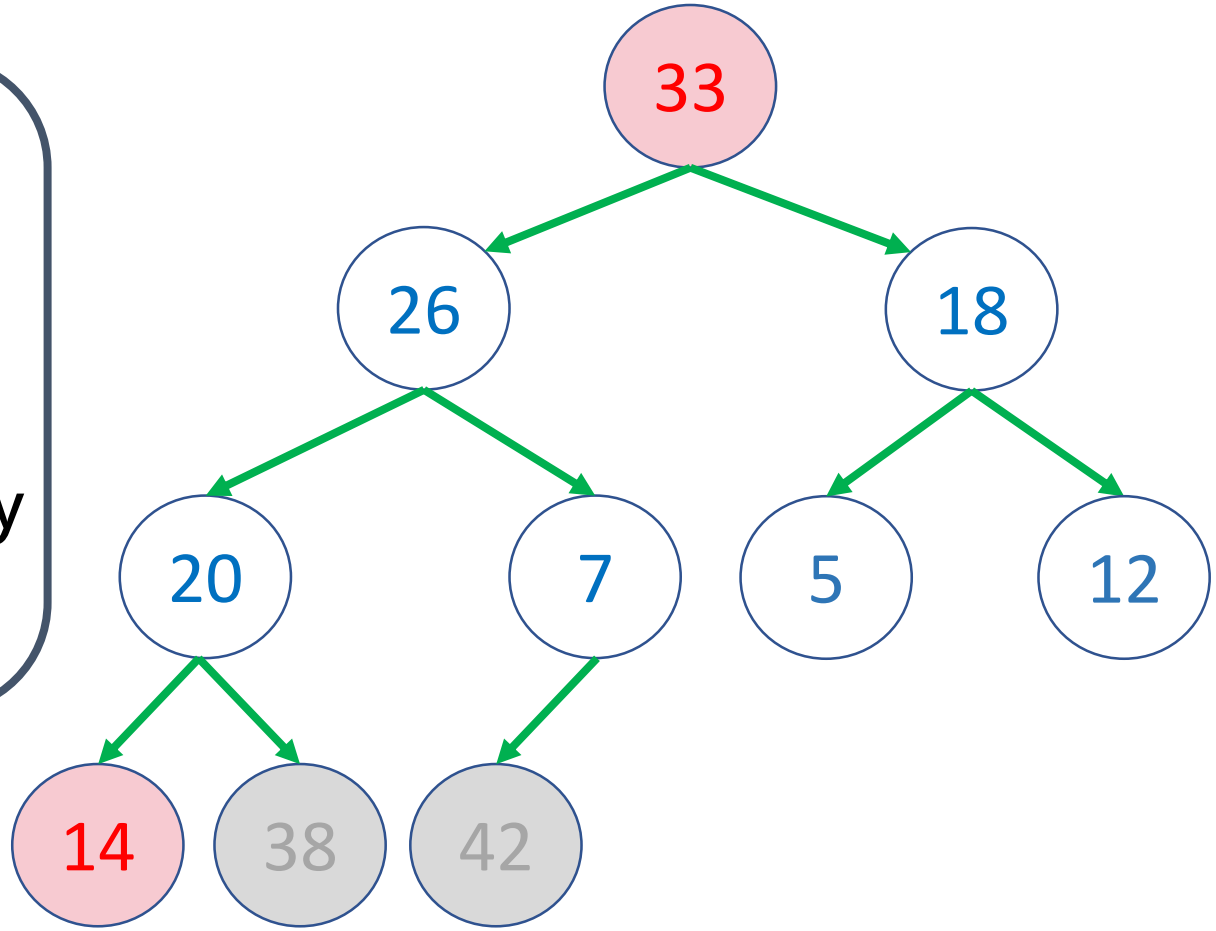


33	26	18	20	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

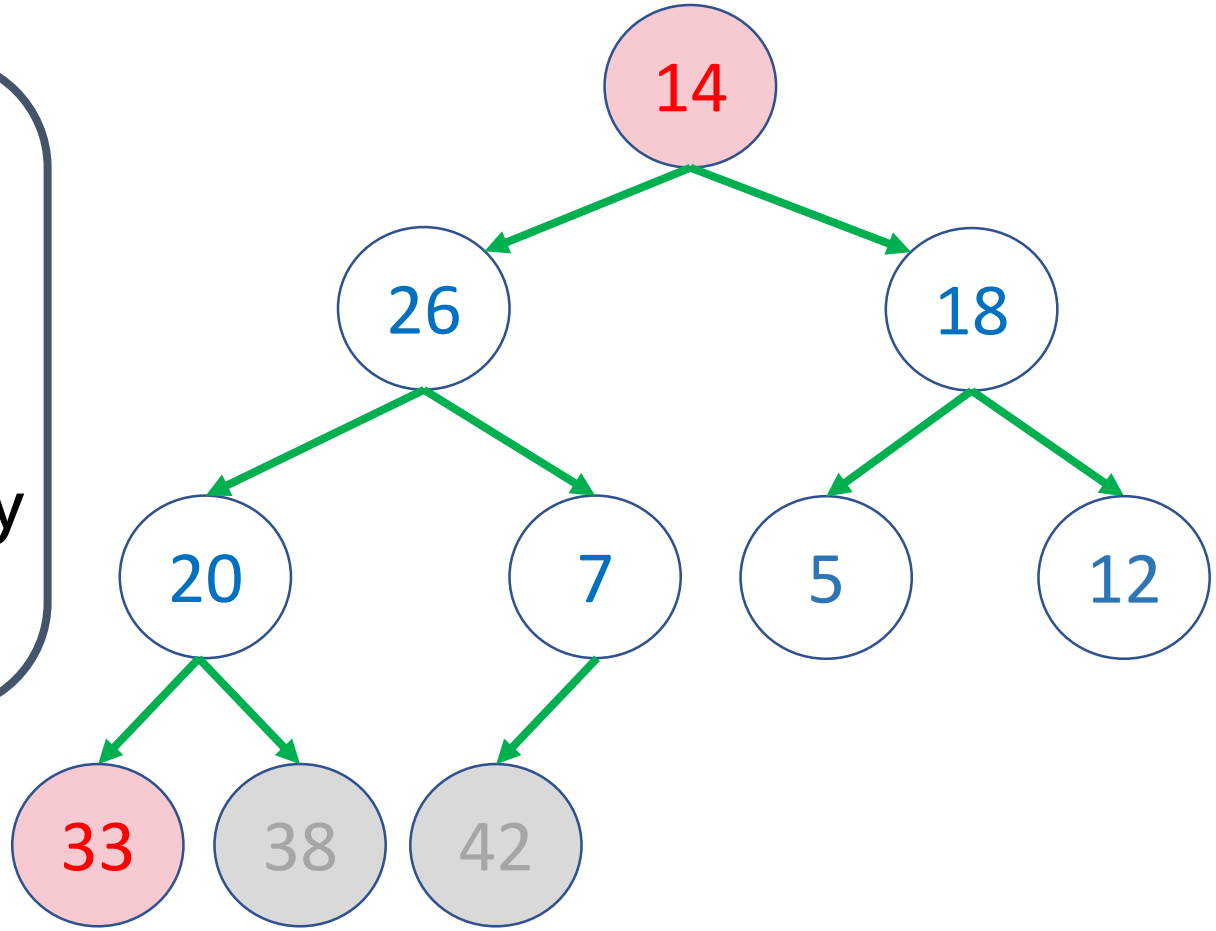


33	26	18	20	7	5	12	14	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

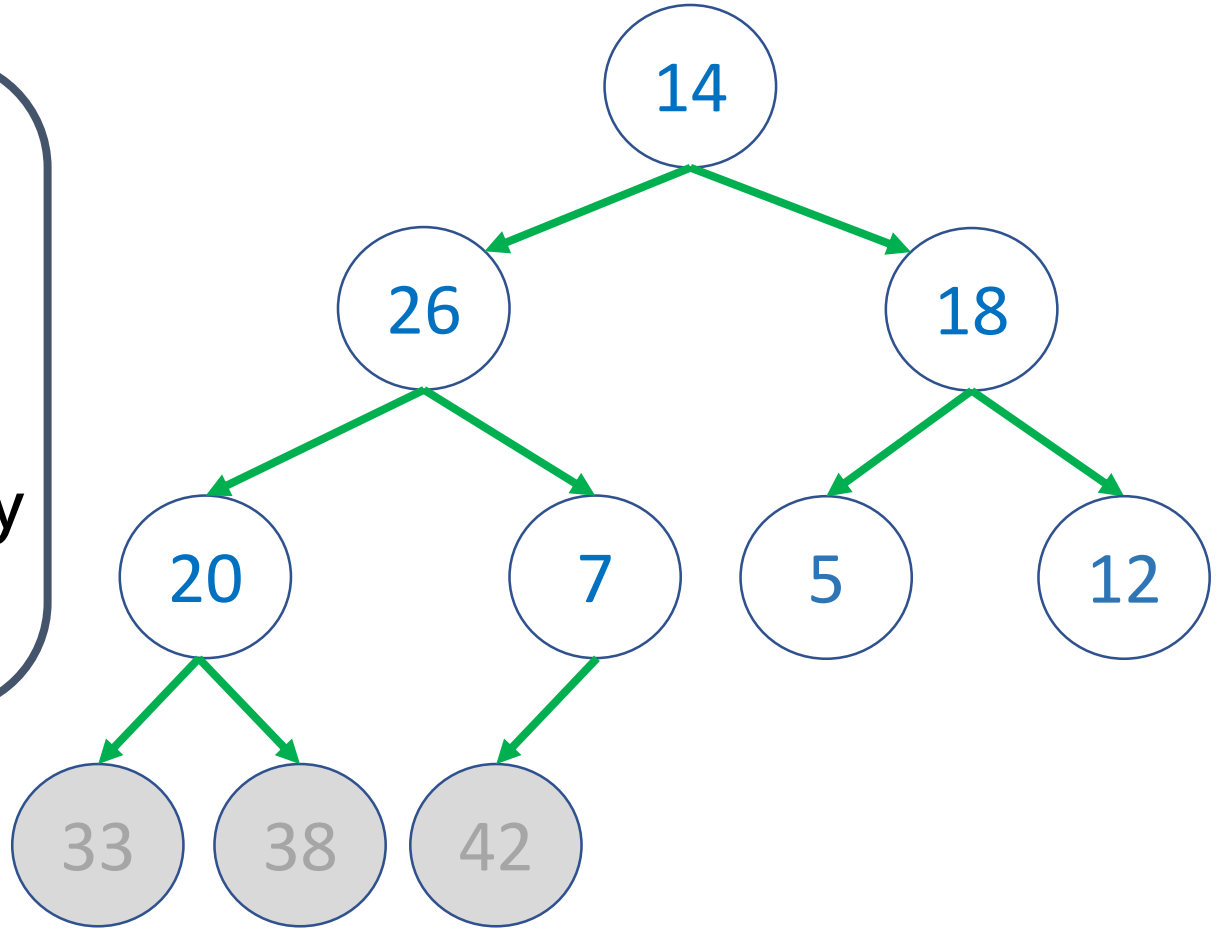


14	26	18	20	7	5	12	33	38	42

堆積排序法

Heap Sort

1. 把最末端跟第一個末端元素交換位置
2. 交換後最後一個元素消失 (已排序)
3. 交換後對根節點及其子節點進行 Heapify
4. 重複步驟 1~3，直到陣列為空

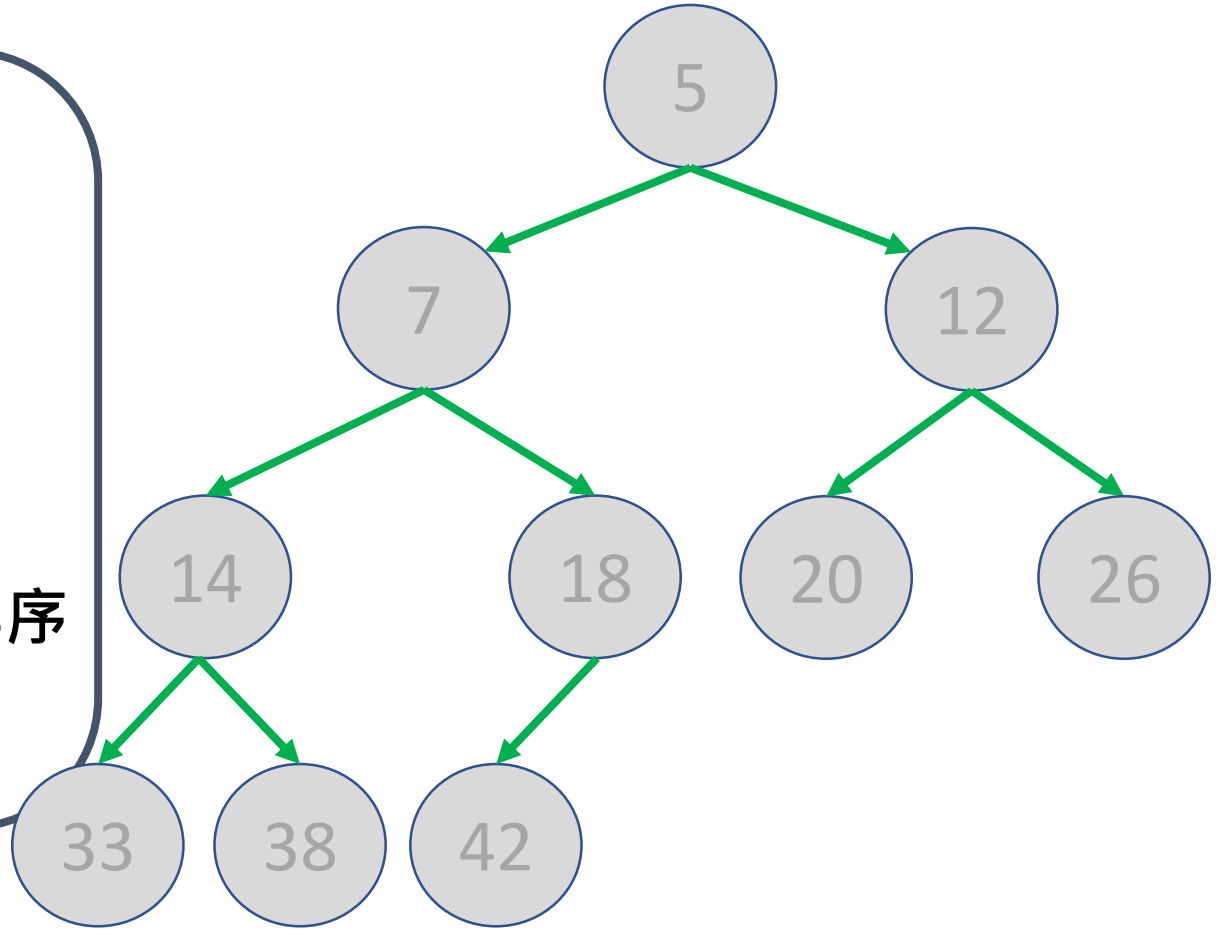


14	26	18	20	7	5	12	33	38	42

堆積排序法

Heap sort

1. 每次形成 max/min-heap 時
   - ➢ 根節點就是該二元樹的最大/小值
2. 首項與末項對調後最大/小值便在最後
3. 依序取出剩餘數列中最大/最小即完成排序
4. 升冪用 max-heap、降冪用 min-heap



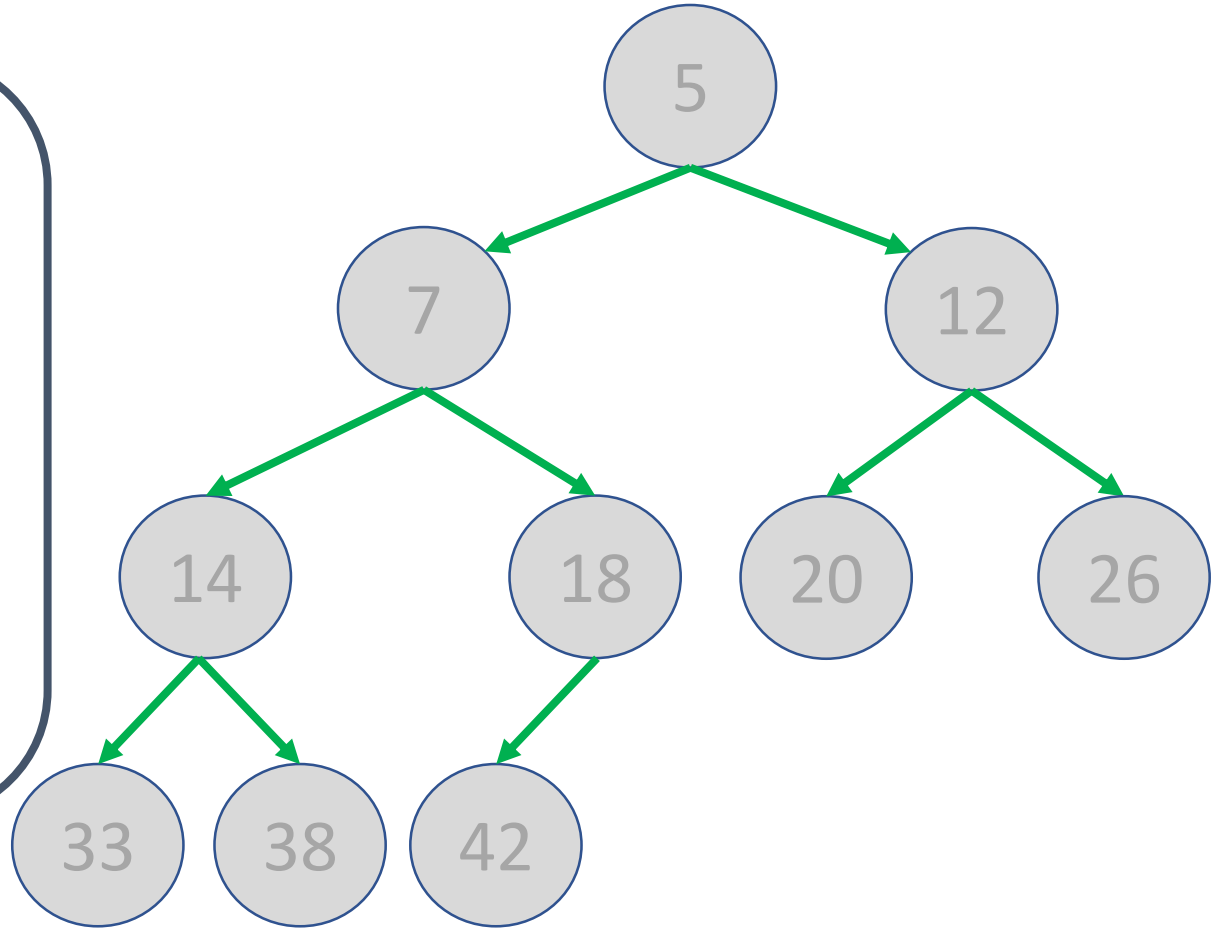
5	7	12	14	18	20	26	33	38	42

堆積排序法

Heap sort 複雜度

1. 建立最大/最小二元堆疊：$O(n\log_2 n)$
2. 每輪 heapify： $O(\log_2 n)$
3. 幾輪 heapify：$O(n)$
4. 總複雜度：

   $O(n\log_2 n)+O(n\log_2 n)=O(n\log_2 n)$



5	7	12	14	18	20	26	33	38	42

Example Code

Mission

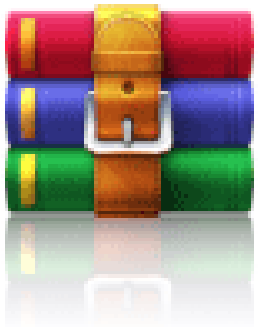
試利用以下數列實作 Heap sort

15	18	6	25	8	11	34	20	2	38

霍夫曼編碼 Huffman Coding

壓縮

- 壓縮
  - ✓ 將資料經過重新計算、編排或編碼
  - ✓ 達到濃縮資料、儲存空間減少
  - ✓ 可以轉換回原本的檔案
    1. mp3
    2. jpg
    3. WinRAR

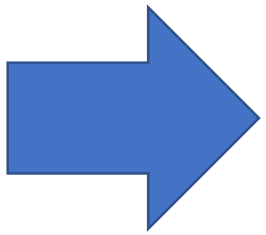


WinRAR

霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼 (Huffman Coding)
  - ✓ 把資料重新編碼減少使用空間
  - ✓ 頻率最高的資料用最小空間編碼
  - ✓ 但編碼表也需要空間 : (
  - ✓ JPG、PDF、MP3都有用！

字元	出現頻率	編碼	總長度 (bits)
A	1	00	2
B	5	01	10
C	2	10	4
D	4	11	8
Total : 24 bits			



字元	出現頻率	編碼	總長度 (bits)
A	1	111	3
B	5	0	5
C	2	110	6
D	4	10	8
Total : 22 bits			

霍夫曼編碼 (Huffman Coding)

- 原始資料如下 (14 個字元)
  - ✓ ABCADBDDDADBDA
- 任意編碼成下面格式
  - ✓ A : 00、B、01、C : 10、D : 11
  - ✓ 共需 28 bits 的空間

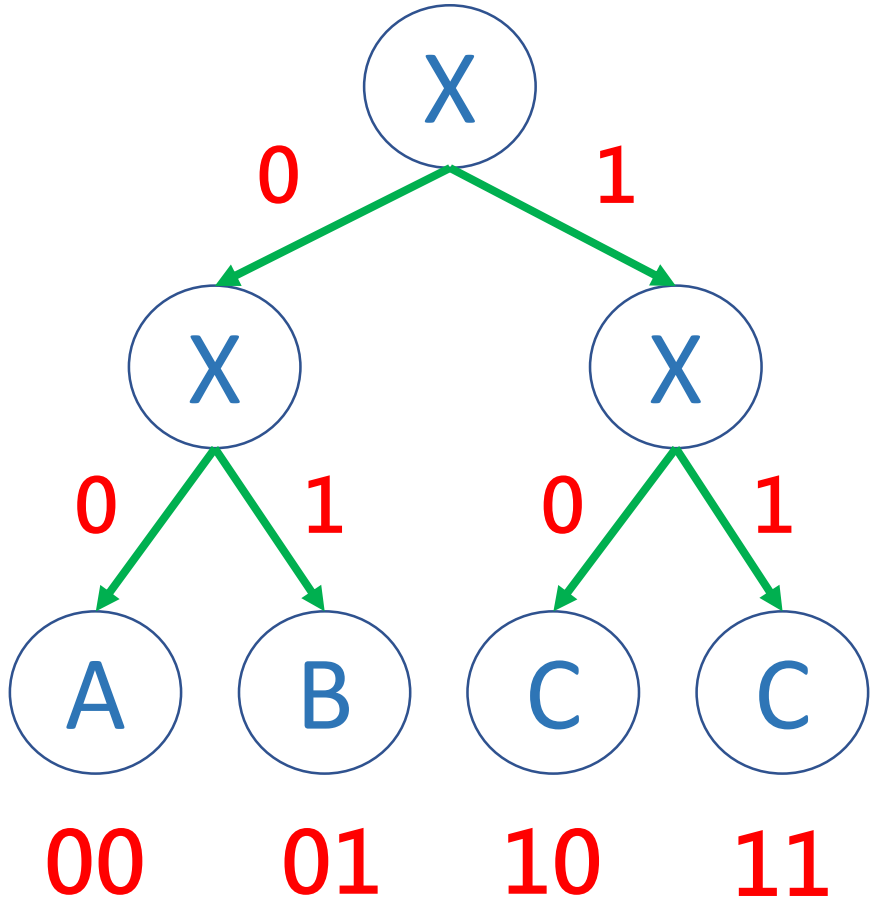
ABCADBDDDADBDA

↓

00 01 10 00 11 01 11 11 11 00 11 01 11 00

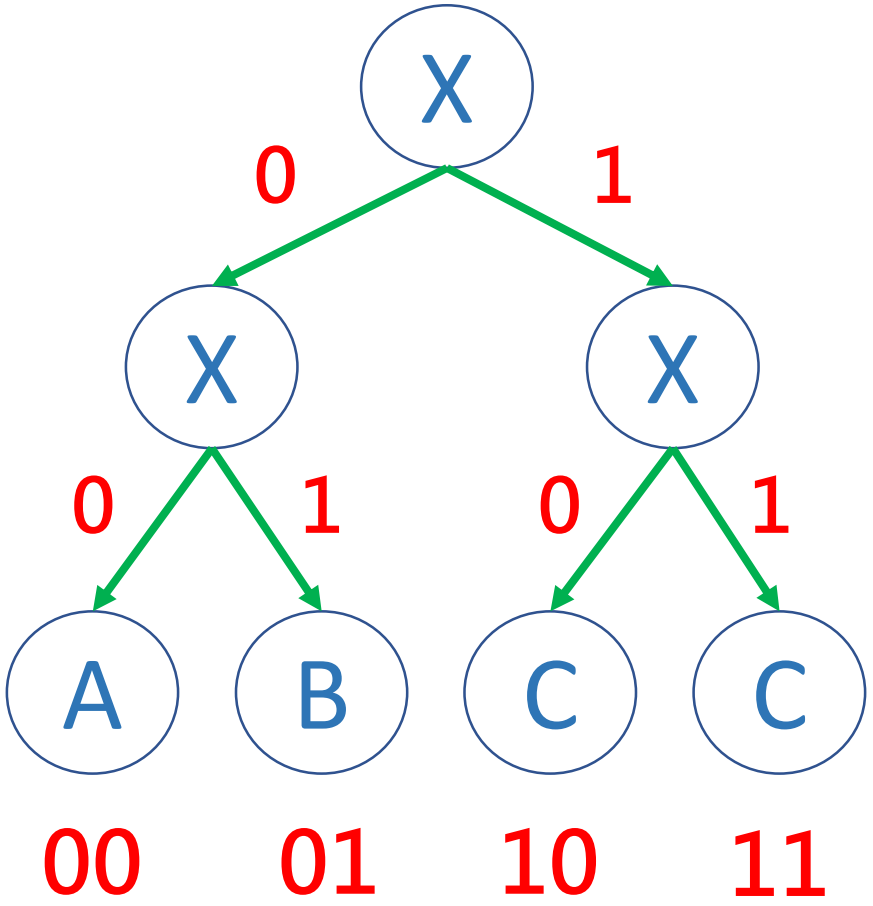
霍夫曼編碼 (Huffman Coding)

- 為什麼不用一個位元去編碼？
  - ✓ A：0、B、1、C：00、D：01



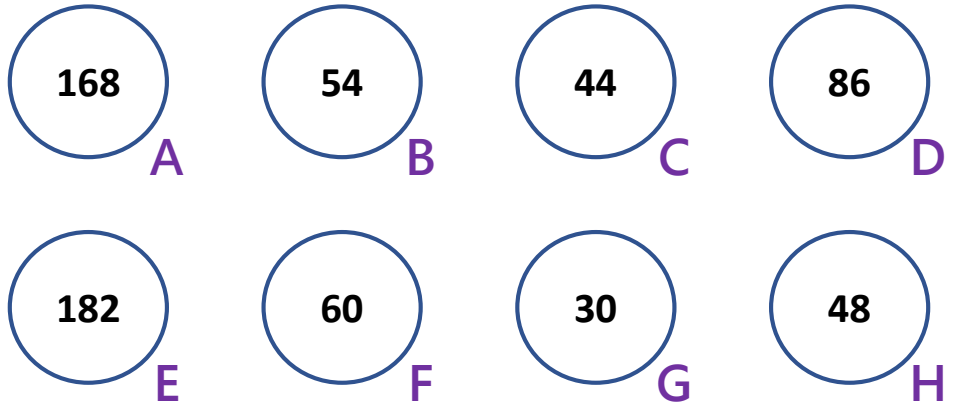
霍夫曼編碼 (Huffman Coding)

- 為什麼不用一個位元去編碼？
  - ✓ A：0、B、1、C：00、D：01
- 如果收到 01，解碼成：
  - ✓ A(0)B(1)？
  - ✓ D(01) ？
- 編碼時須確保解碼的結果是唯一解！
- 以二元樹來看，編碼結果在 Leaf node 上



霍夫曼編碼 (Huffman Coding)

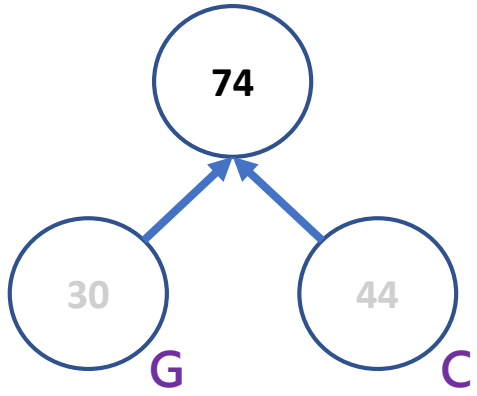
- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點

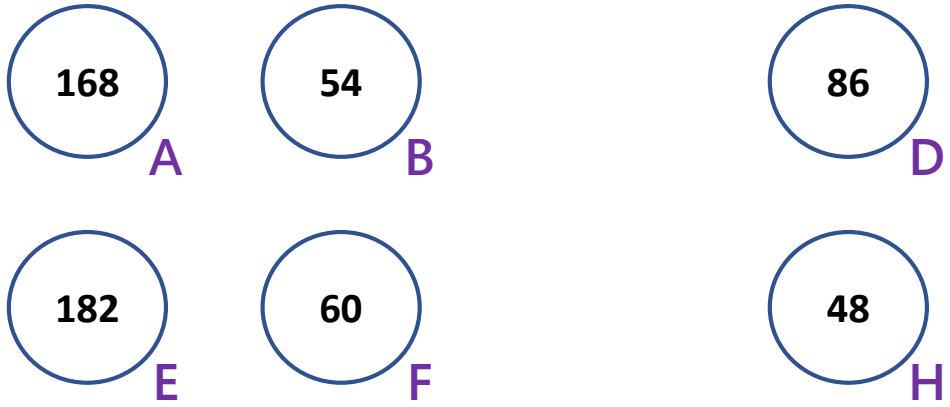


字元	頻率
A	168
B	54
C	44
D	86
E	182
F	60
G	30
H	48
Total : 672	

霍夫曼編碼 (Huffman Coding)

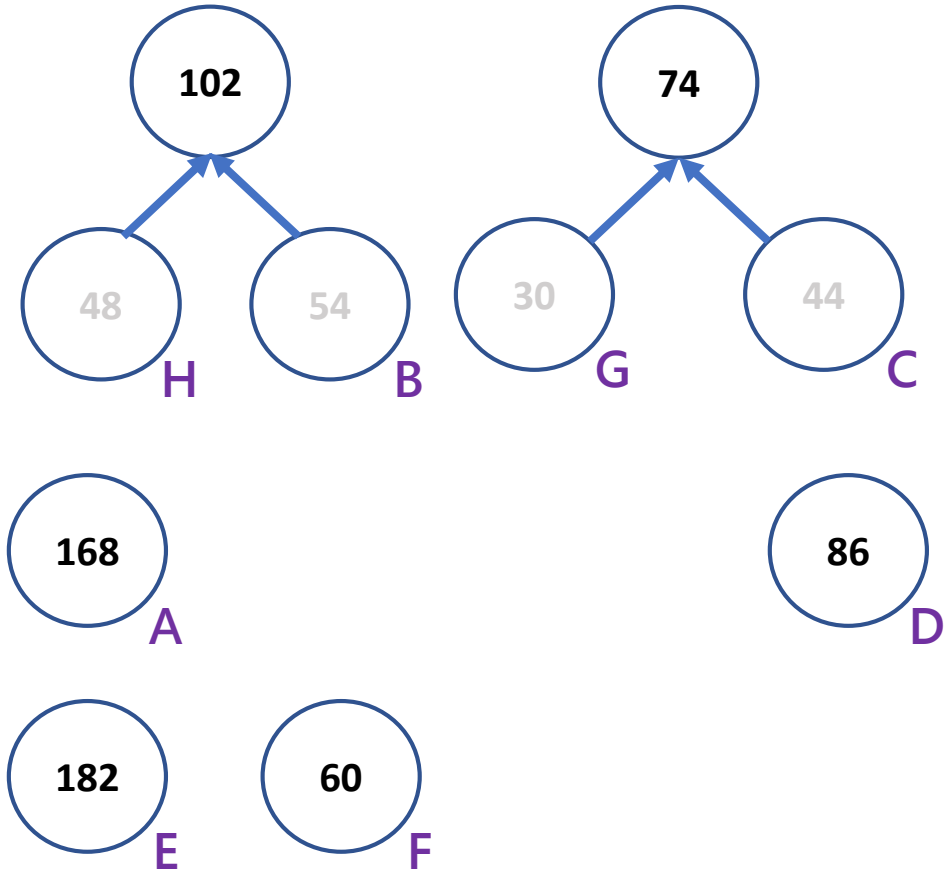
- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和





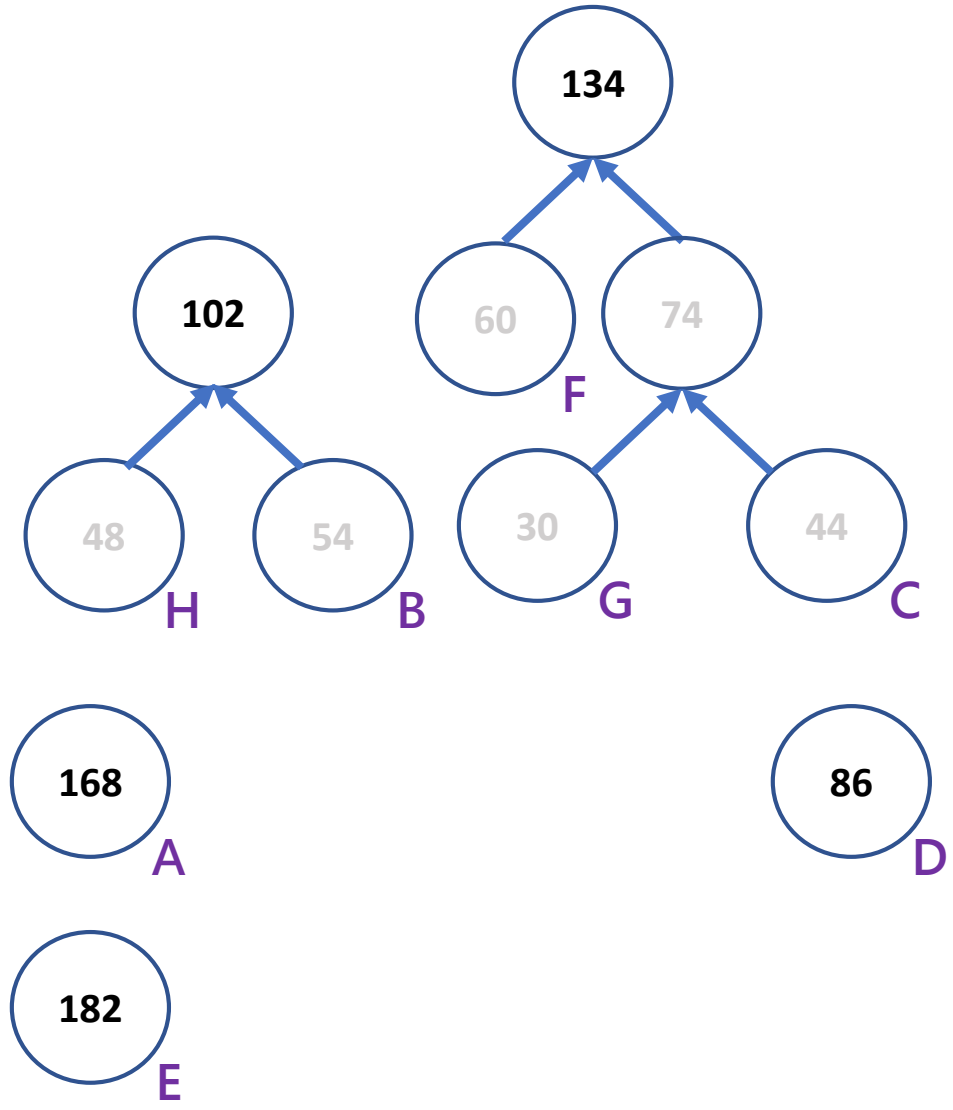
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹



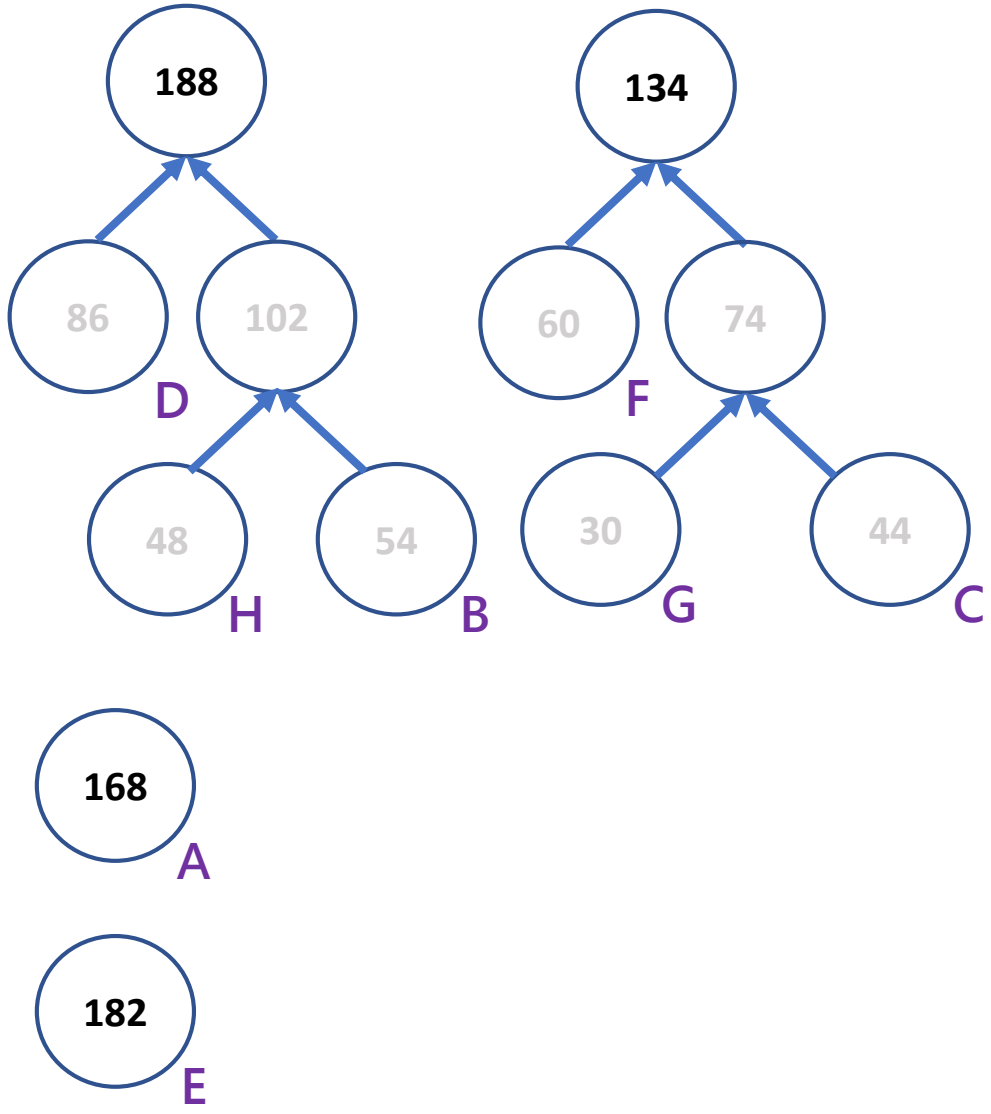
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹



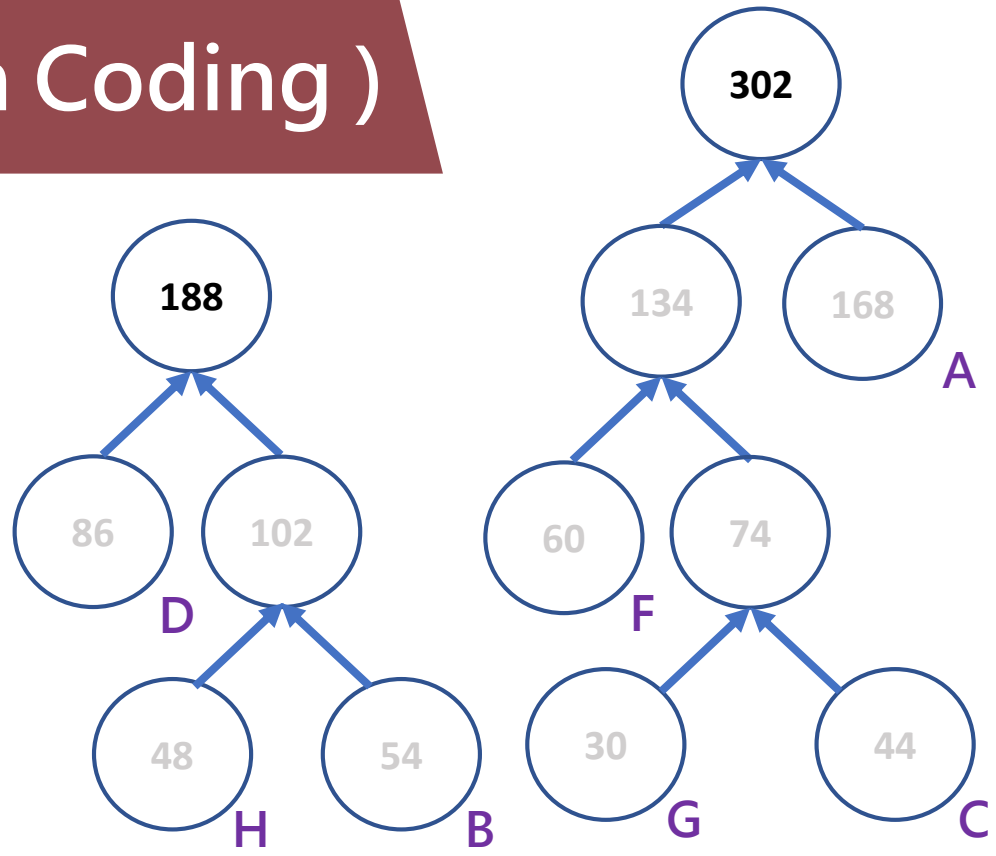
霍夫曼編碼 (Huffman Coding)

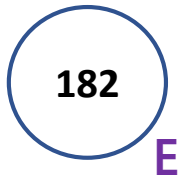
- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹



霍夫曼編碼 (Huffman Coding)

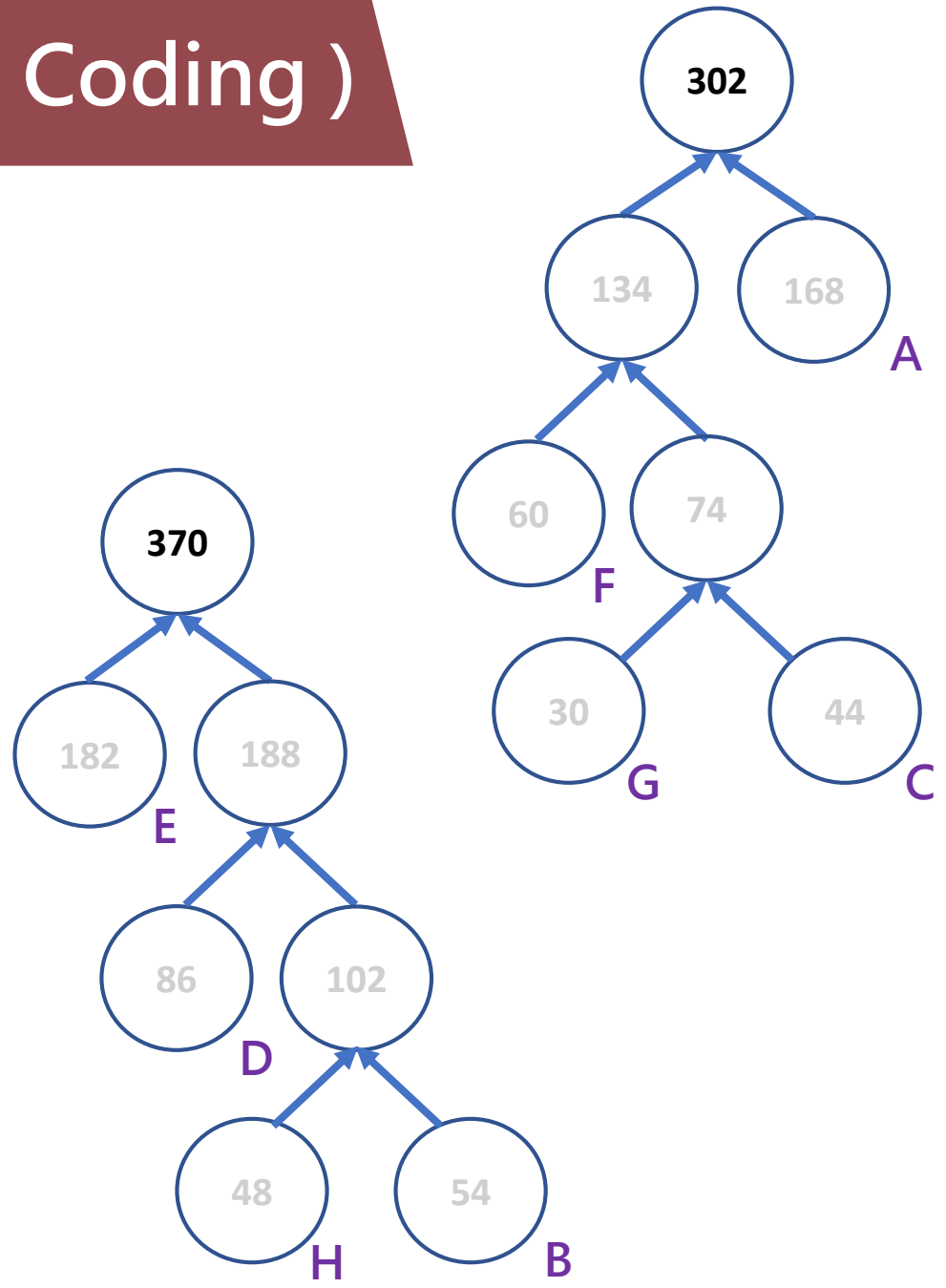
- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹





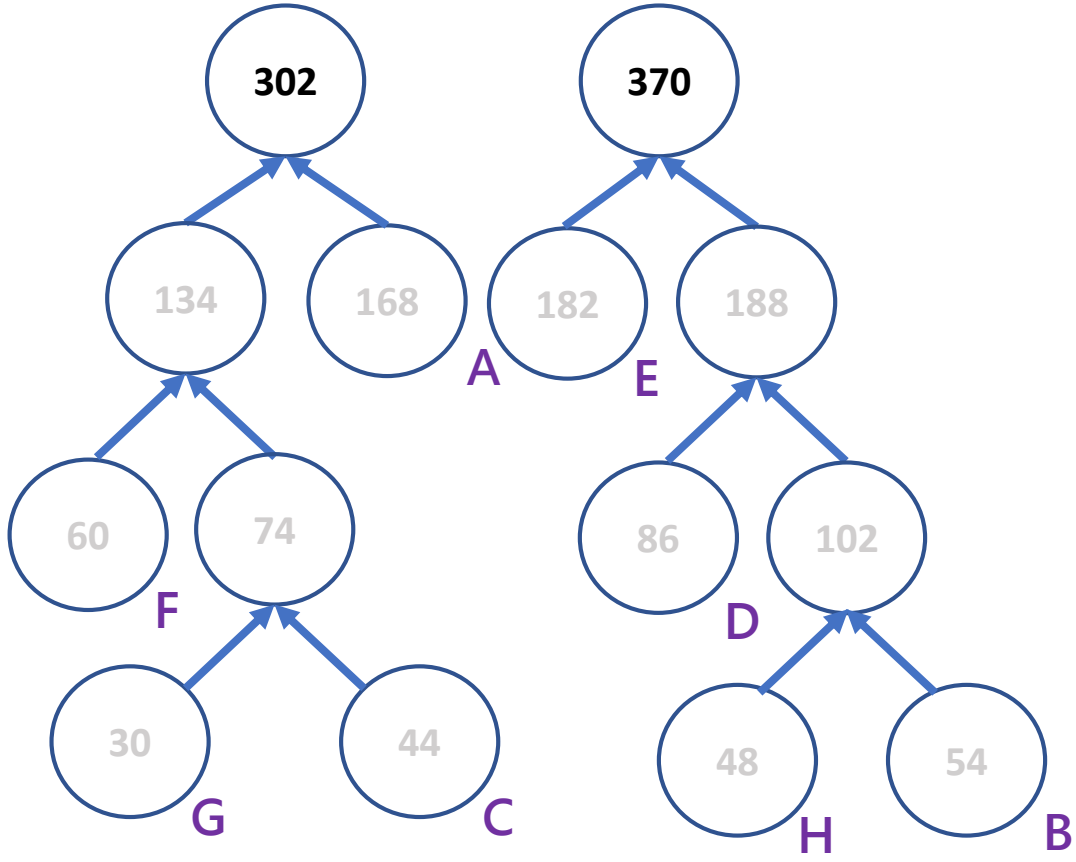
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹



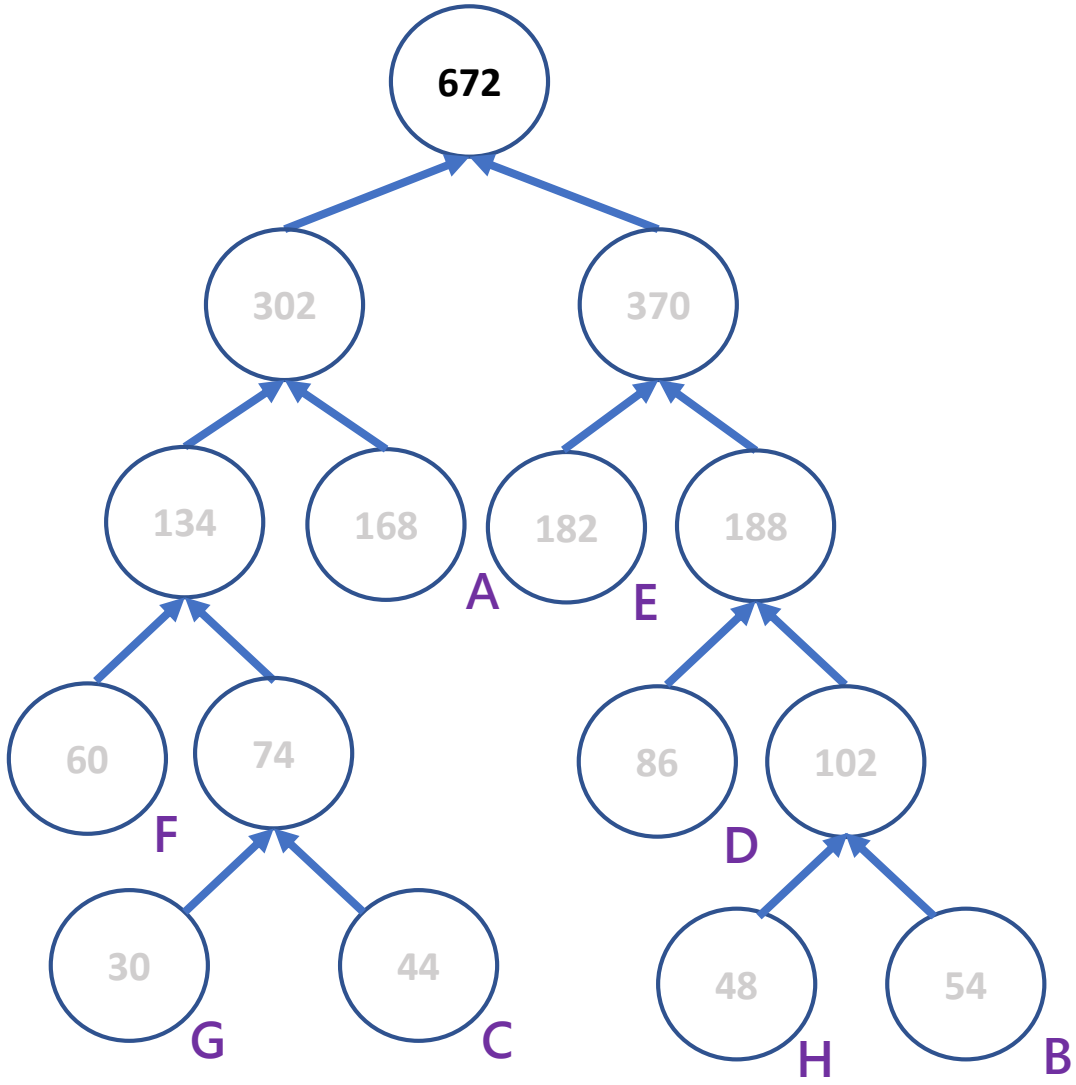
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹



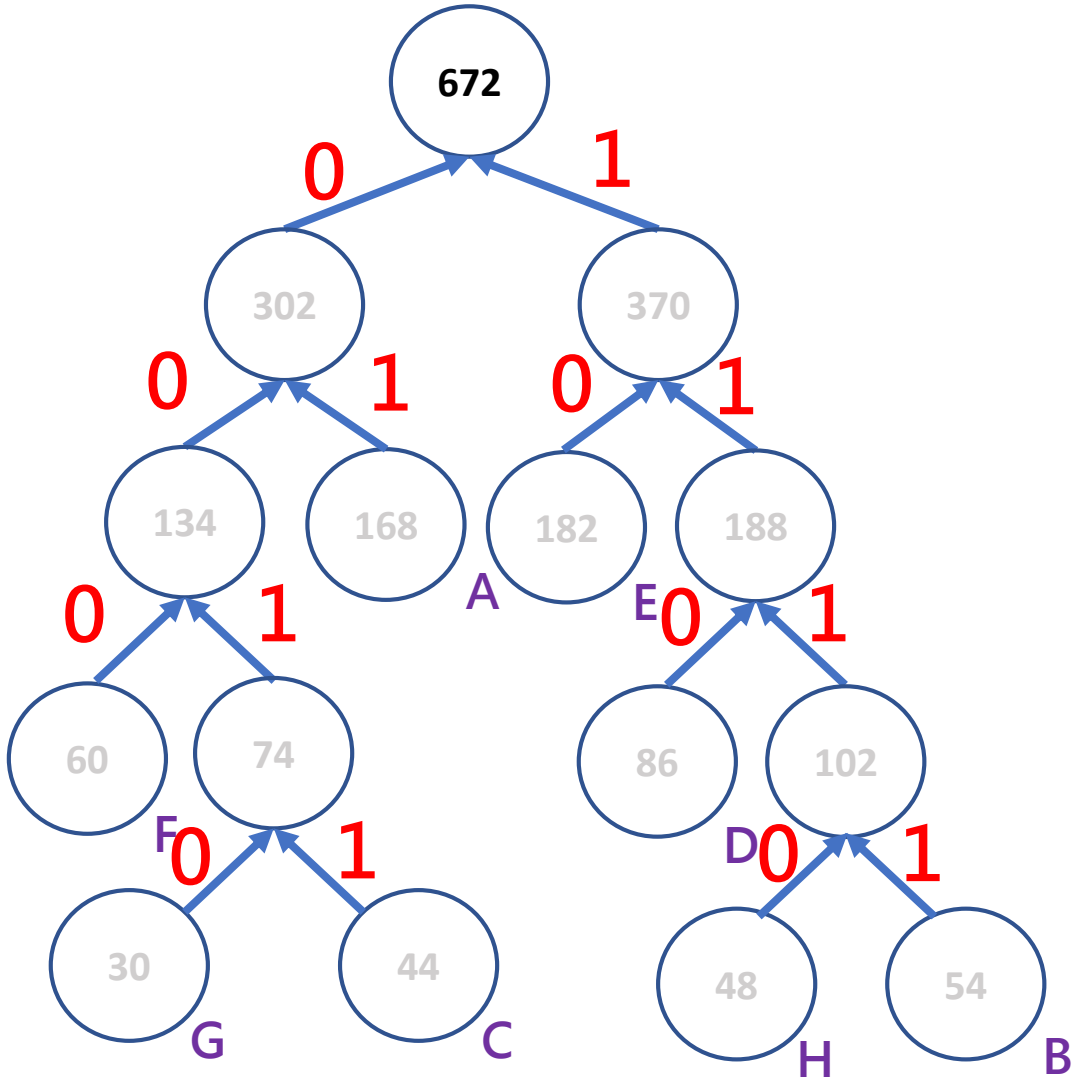
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹



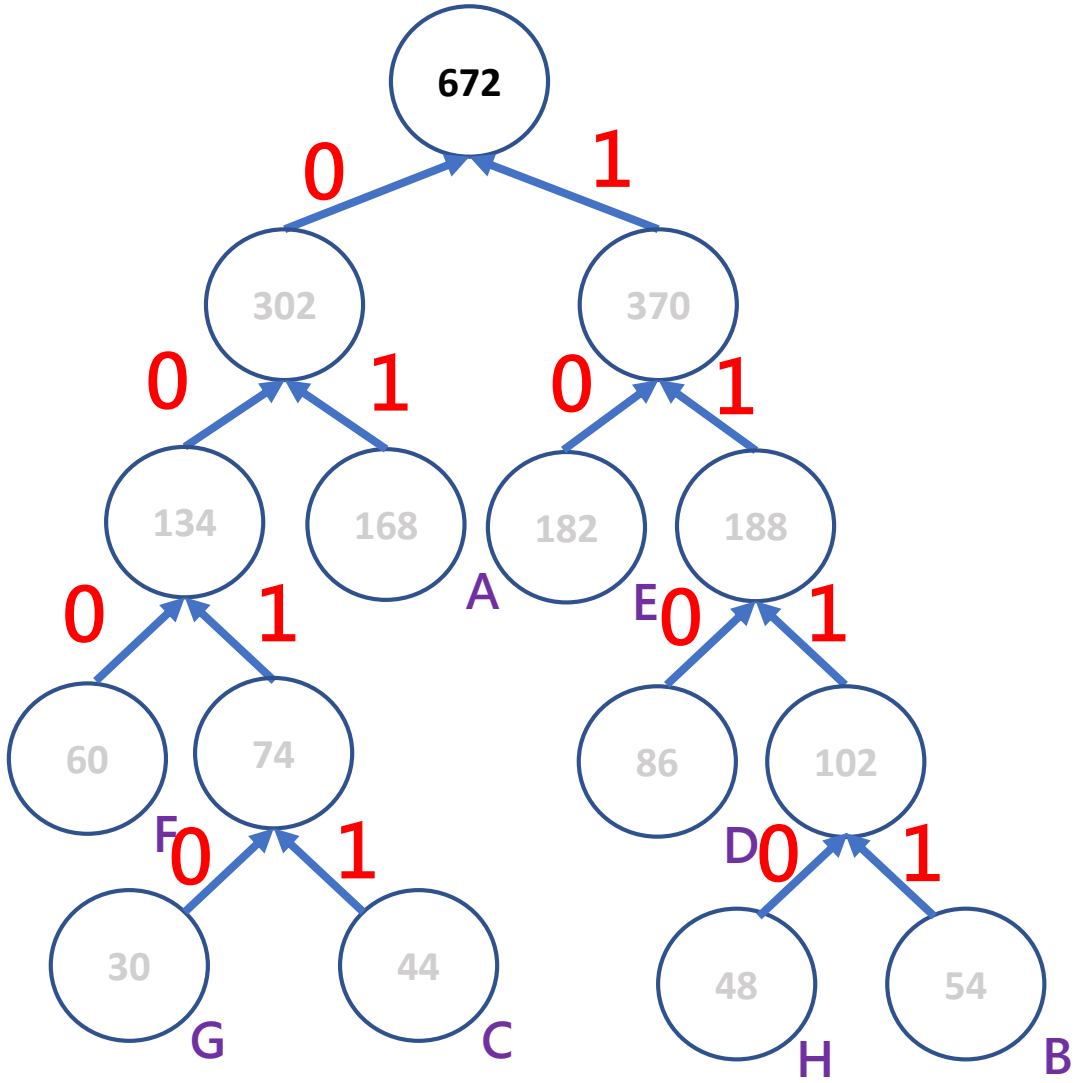
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹
  5. 依序往左、右分別填 0、1



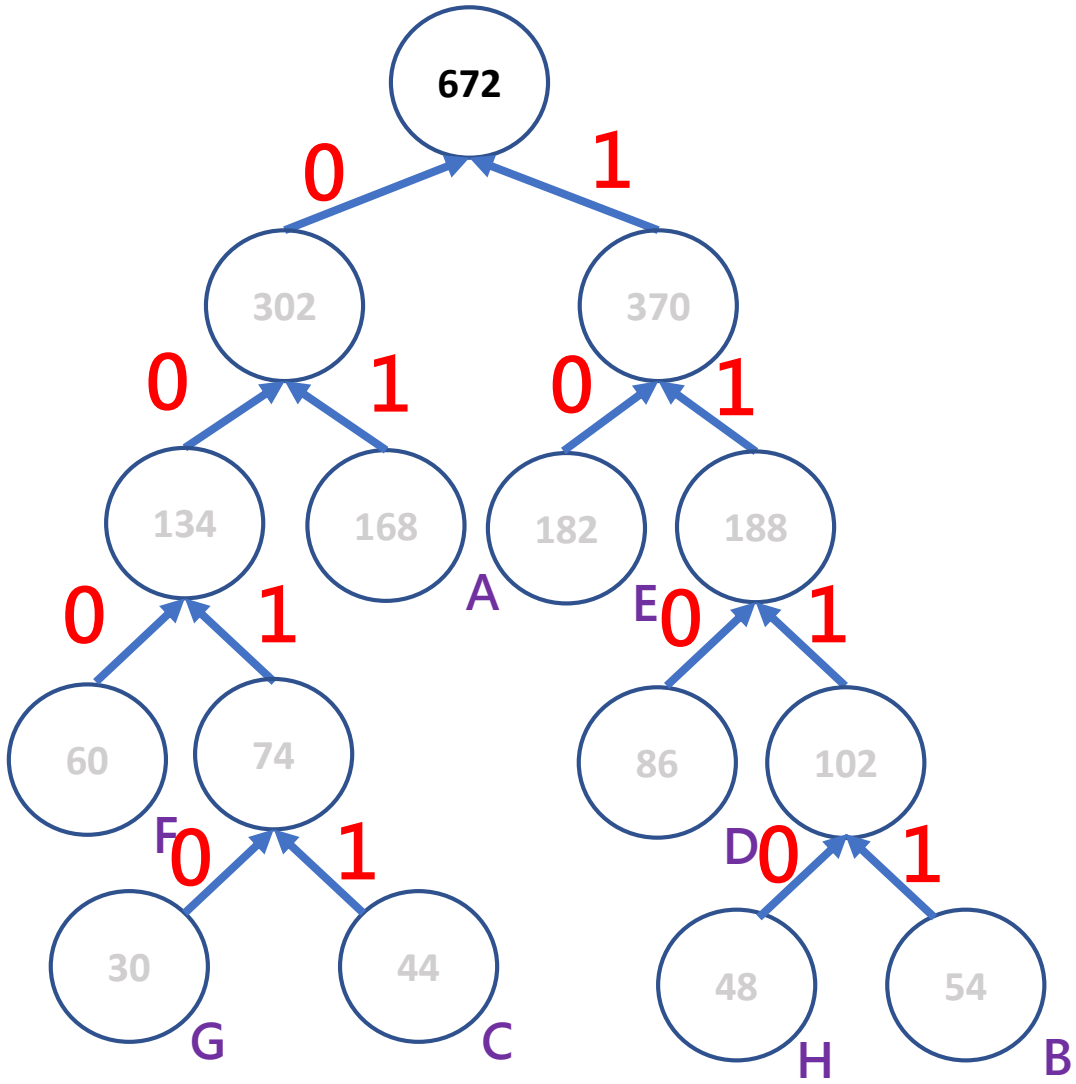
霍夫曼編碼 (Huffman Coding)

- 霍夫曼編碼步驟
  1. 建立字頻表
  2. 每個字元與頻率視為一個節點
  3. 最小的兩節點依序聚合成子樹
     - 父節點為子節點的頻率和
  4. 重複步驟 3 ，直到形成嚴格二元樹
  5. 依序往左、右分別填 0、1
  6. 建立編碼表



霍夫曼編碼 (Huffman Coding)

字元	頻率	編碼
A	168	01
B	54	1111
C	44	0011
D	86	110
E	182	10
F	60	000
G	30	0011
H	48	1110
Total : 672		



霍夫曼編碼 (Huffman Coding)

一般編碼			
字元	頻率	編碼	總空間
A	168	000	504
B	54	001	162
C	44	010	132
D	86	011	258
E	182	100	546
F	60	101	180
G	30	110	90
H	48	111	144
Total : 672		Total : 2016	

霍夫曼編碼			
字元	頻率	編碼	總空間
A	168	01	336
B	54	1111	216
C	44	0011	176
D	86	110	258
E	182	10	364
F	60	000	180
G	30	0010	120
H	48	1110	192
Total : 672		Total : 1842	

壓縮率=1842/2016~91.4%

Priority Queue

- Priority Queue 優先權佇列
  - ➢ Priority
    - ✓ 額外賦予資料權重
  - ➢ Queue
    - ✓ 依照優先權依序排列後再依序輸出
- 沒有 front() 只有 top()
- 輸出次序
  - ➢ 依照資料間的權重大小

Priority Queue

Priority Queue

- #include <queue>
  - ✓ priority_queue<datatype, container, compared_method>
    - ☐ datatype：要比較的資料型態
    - ☐ container：組成 queue 的容器 (vector 或 deque)
      - 預設為 vector
    - ☐ compared_method：比較方式
  - ✓ priority_queue<datatype>
- 預設權重越大越接近 top

Priority Queue

- compared_method
  - ✓ greater<datatype>：由小到大
  - ✓ less<datatype>：由大到小
  - ✓ 預設由大到小 (less)、運算子<
- 自定義函式
  - ✓ 重載運算子 <
  - ✓ 寫一個結構或類別，內含()運算子重載

Practice

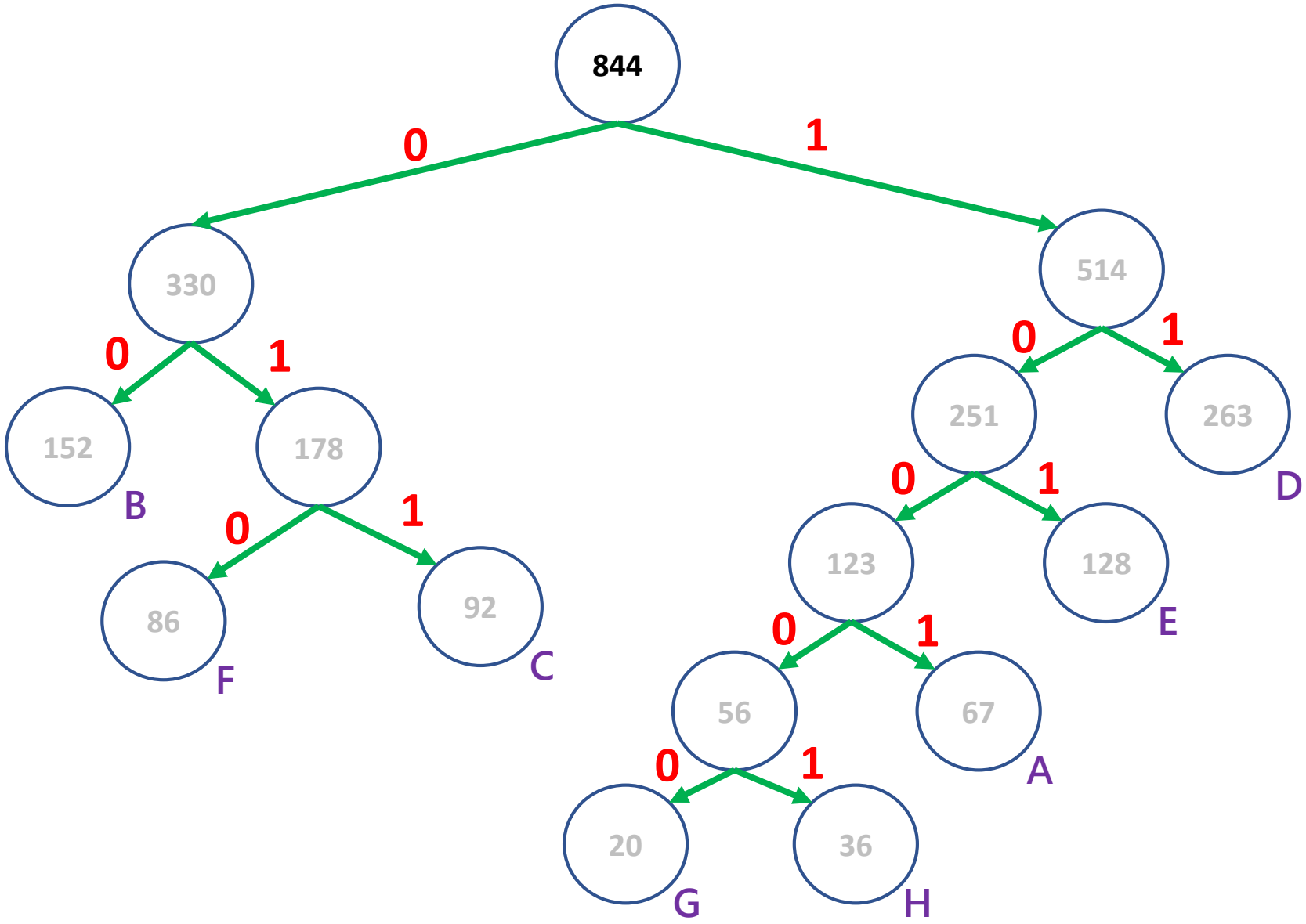
Mission

給定一字頻表如下，請利用霍夫曼編碼重新編成二進位制

字元	頻率	編碼
A	67	
B	152	
C	92	
D	263	
E	128	
F	86	
G	20	
H	36	

Practice

字元	頻率	編碼
A	67	1001
B	152	00
C	92	011
D	263	11
E	128	101
F	86	010
G	20	10000
H	36	10001



Example Code

Mission

利用優先權佇列 (Priority Queue) 新增或刪除節點，使其權重小的先輸出

```
struct Node;
struct Node{
    char character;
    int counts;
    Node* left;
    Node* right;
};
typedef struct Node Node;
```

```
class Compare{
    public:
        bool operator()(Node* n1,Node* n2){
            return n1->counts > n2->counts;
        }
};
```

Example Code

```
class Huffman_Tree{
    private:
        priority_queue<Node*, vector<Node*>, Compare> all_nodes;
        Node* Pop_Node();
    public:
        Node* root_node;
        Huffman_Tree();
        void Clear_Queue();
        void Insert(char,int);
        void Build_Huffman_Tree();
        void Print_Code_Table();
        void Traversal(Node*,queue<char>);
};
```

Example Code

Mission

給定字元與其出現頻率，試著完成霍夫曼編碼的以下兩功能：

1. 建立霍夫曼編碼
2. 向下圖一樣印出每個字母的編碼形式

```
Char: F, Counts: 60, Code: 000
Char: G, Counts: 30, Code: 0010
Char: C, Counts: 44, Code: 0011
Char: A, Counts: 168, Code: 01
Char: E, Counts: 182, Code: 10
Char: D, Counts: 86, Code: 110
Char: H, Counts: 48, Code: 1110
Char: B, Counts: 54, Code: 1111
```

Example Code

Mission

霍夫曼編碼的時間複雜度為何？
(若優先權佇列是以二元堆疊完成)

建立最大/小堆疊：$O(n \log_2 n)$

取出最小值：$O(\log_2 n)$
合併成新節點：$O(1)$

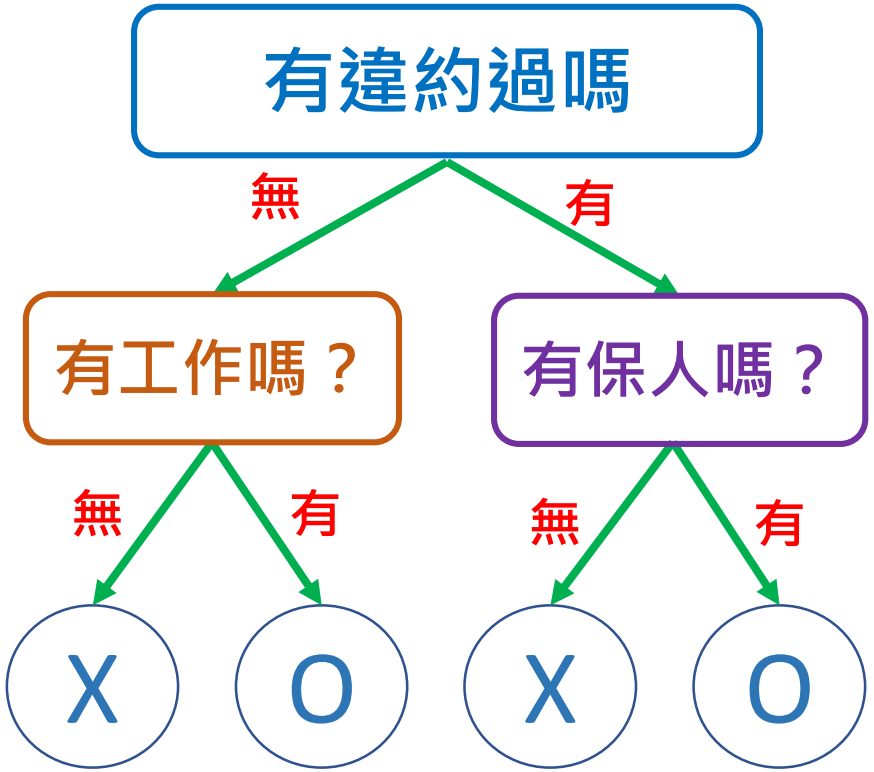
取出&合併次數：O(n-1)=O(n)

總計：$O(n \log_2 n)$

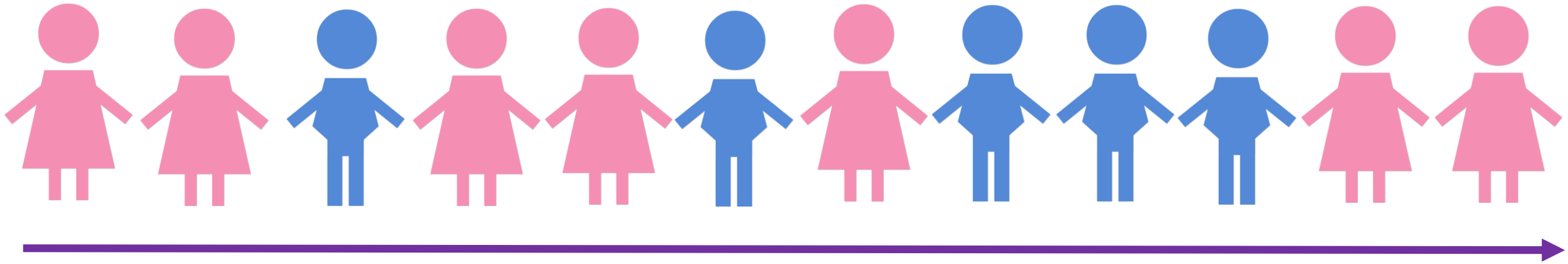
決策樹 Decision Tree

決策樹 Decision Tree

- 處理分類問題的樹狀結構
- 接近人類的思考方式
- 每次都有一個「準則」供判斷
- 透過訓練資料集自動生成二元樹
  - ✓ 嚴格二元樹
  - ✓ Leaf node：判斷結果
  - ✓ Non-leaf node：分類器

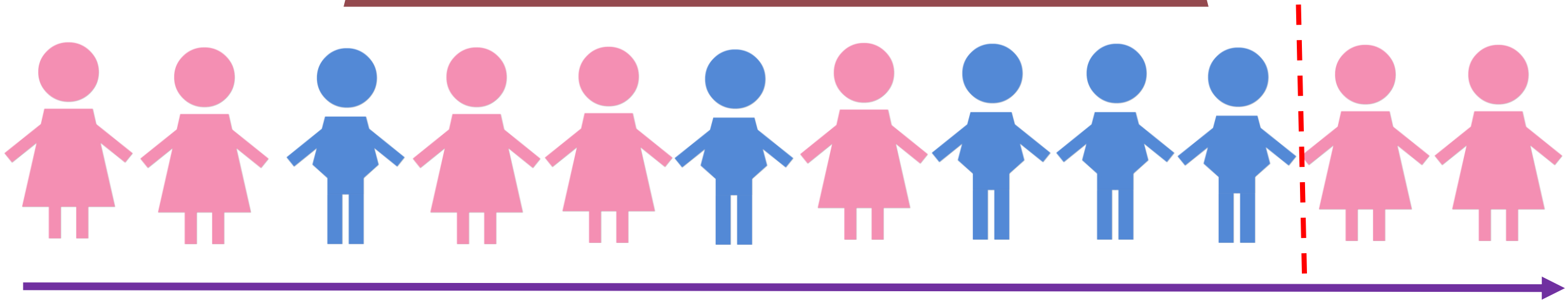


決策樹 Decision Tree

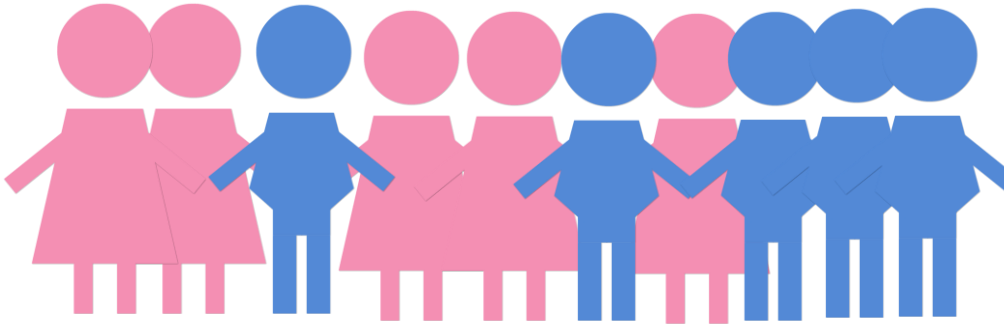


- 要怎麼知道切在哪裡？
  - ✓ 要怎麼定義資料的「純度 (Purity)」？
  - ✓ 純度的定義會決定分類的結果
    1. Entropy
    2. Gini-index

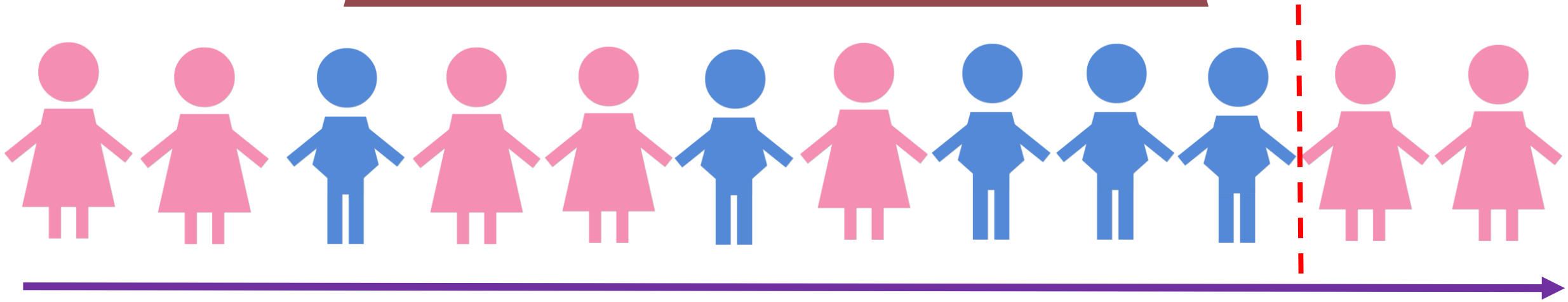
決策樹 Decision Tree



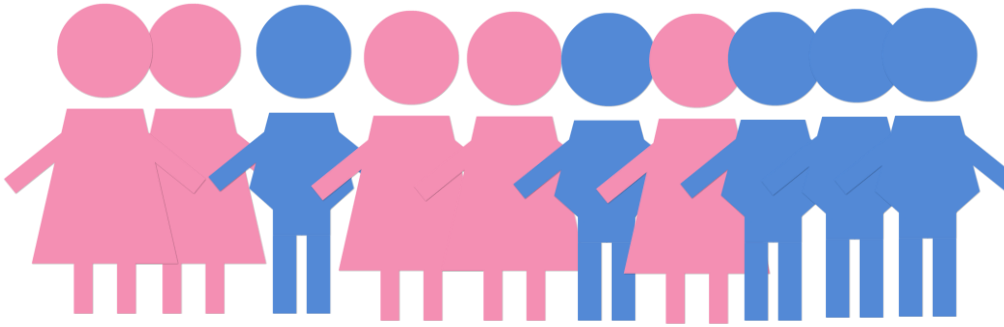
- Entropy
  - ✓ $-p * log_2 p - q * log_2 q$
  - ✓ p：正確機率、q：失敗機率
  - ✓ 右：$-\frac{2}{2} log_2 \frac{2}{2} - \frac{0}{2} * log_2 \frac{0}{2} \approx 0$
  - ✓ 左：$-\frac{5}{10} log_2 \frac{5}{10} - \frac{5}{10} * log_2 \frac{5}{10} \approx 1$



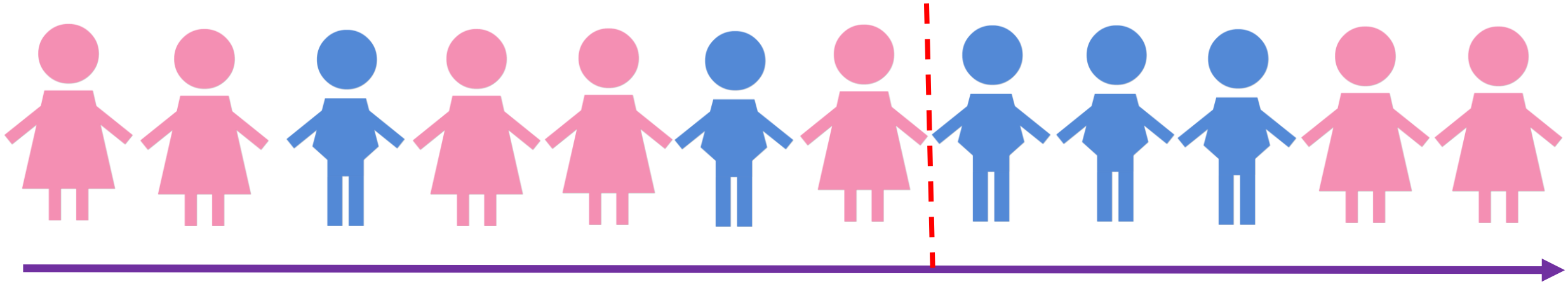
決策樹 Decision Tree



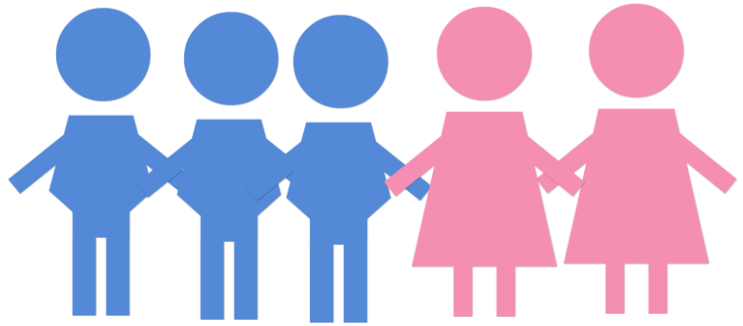
- Entropy
  - ✓ 右：0
  - ✓ 左： 1
  - ✓ 加權：$0 \times \frac{2}{12} + 1 \times \frac{10}{12} \approx 0.833$

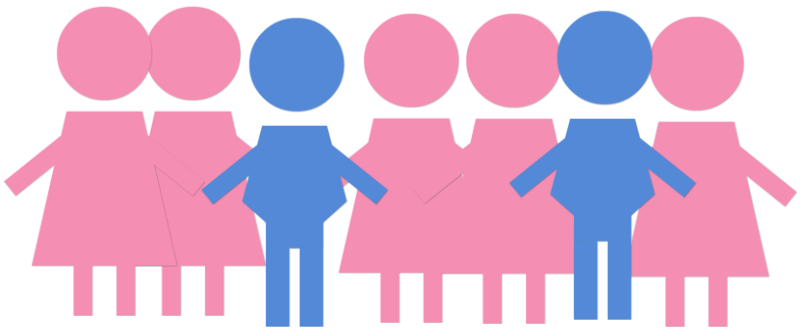


決策樹 Decision Tree

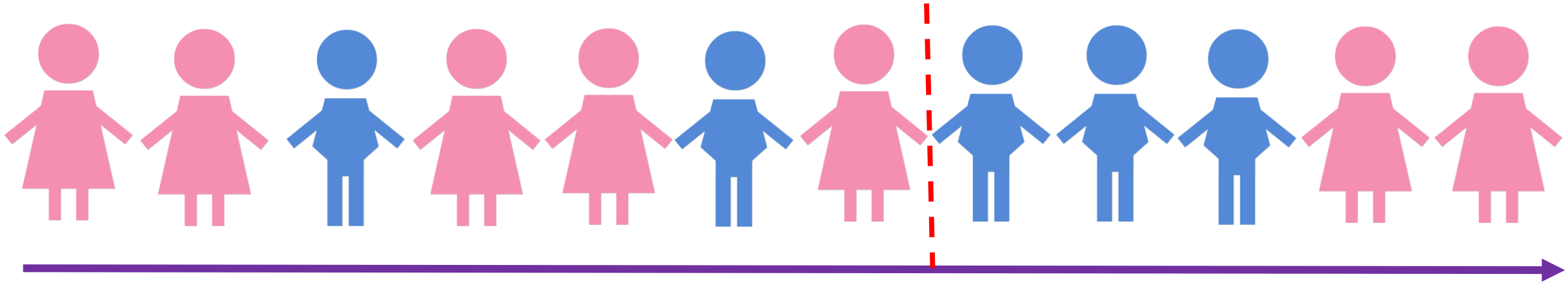


- Entropy
  - ✓ $-p * log_2 p - q * log_2 q$
  - ✓ p：正確機率、q：失敗機率
  - ✓ 右：$-\frac{3}{5} log_2 \frac{3}{5} - \frac{2}{5} * log_2 \frac{2}{5} \approx 0.971$
  - ✓ 左：$-\frac{5}{7} log_2 \frac{5}{7} - \frac{2}{7} * log_2 \frac{2}{7} \approx 0.863$

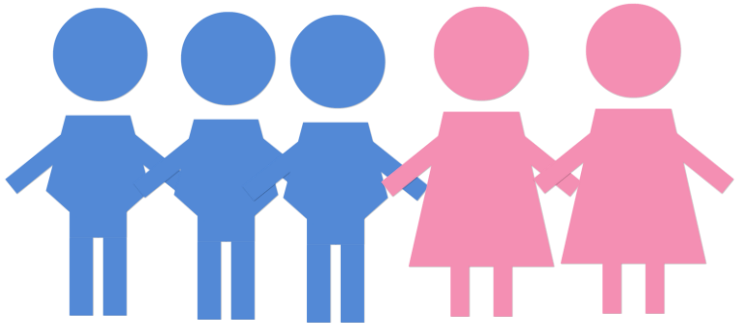


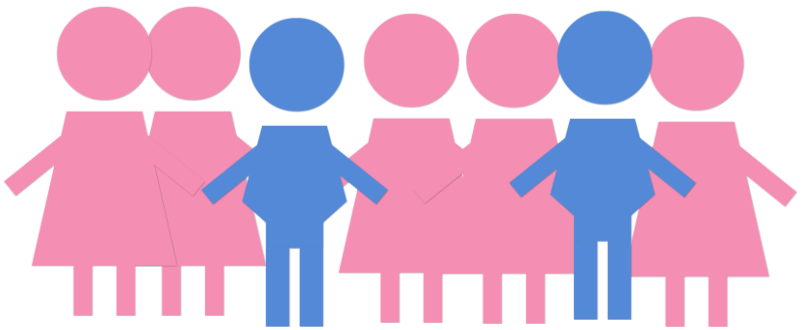


決策樹 Decision Tree

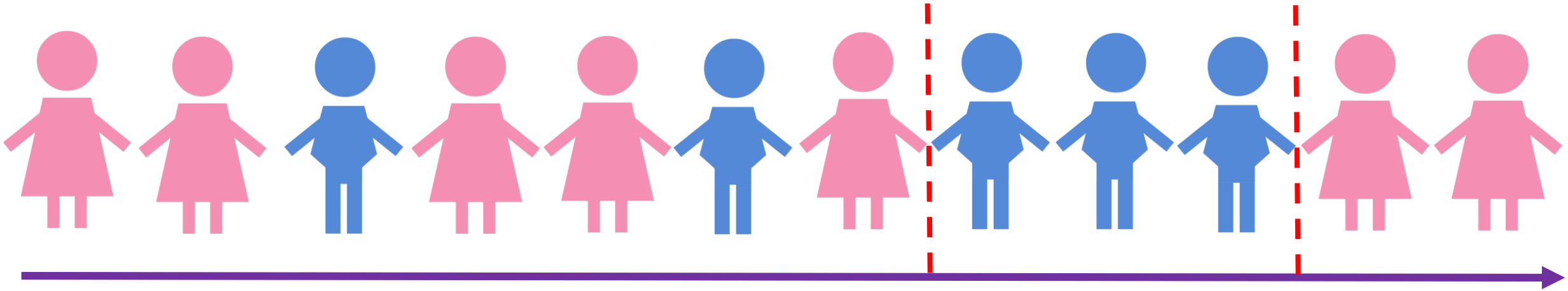


- Entropy
  - ✓ 右：0.971
  - ✓ 左： 0.863
  - ✓ 加權：$0.971 \times \frac{5}{12} + 0.863 \times \frac{7}{12} \sim 0.908$





決策樹 Decision Tree

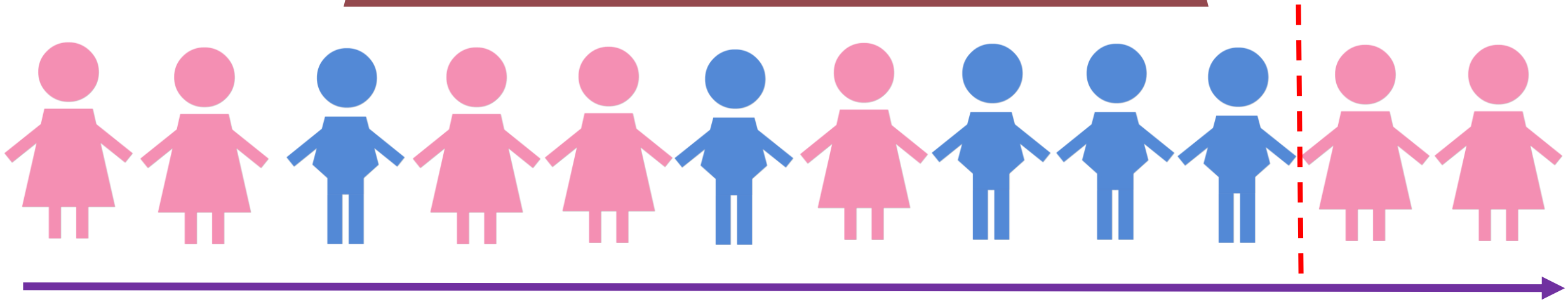


0.908

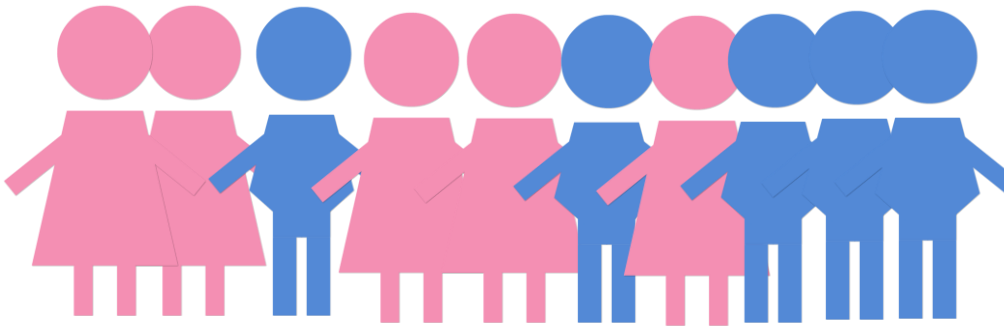
0.833

選亂度 (Entropy) 小的

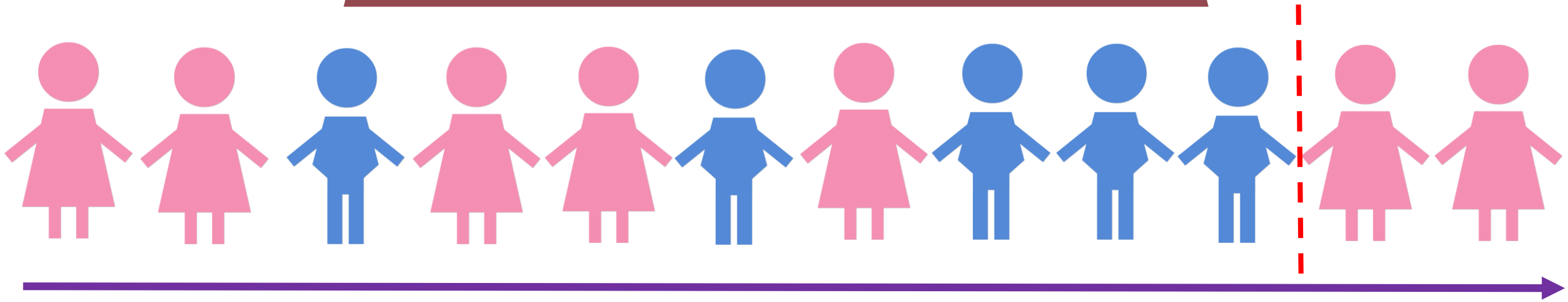
決策樹 Decision Tree



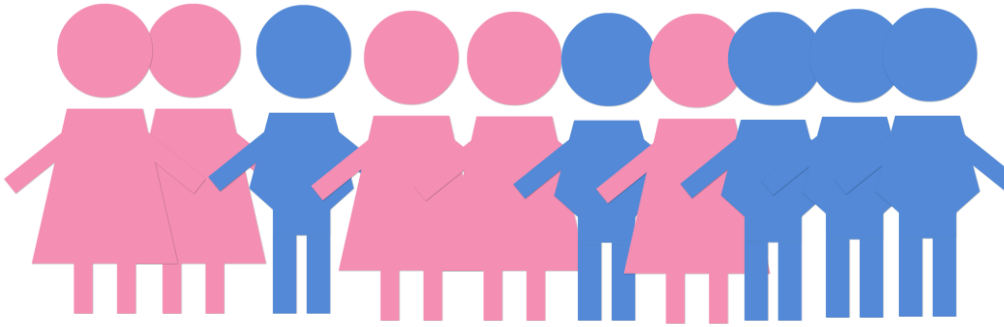
- Gini index
  - ✓ $p^2 + q^2$
  - ✓ p：正確機率、q：失敗機率
  - ✓ 右：$1^2 + 0^2 = 1$
  - ✓ 左：$\left(\frac{5}{10}\right)^2 + \left(\frac{5}{10}\right)^2 = 0.5$



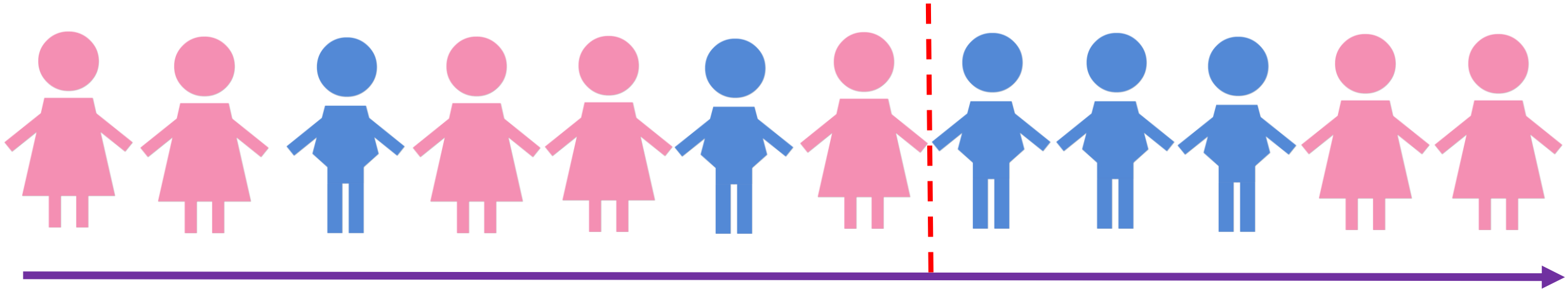
決策樹 Decision Tree



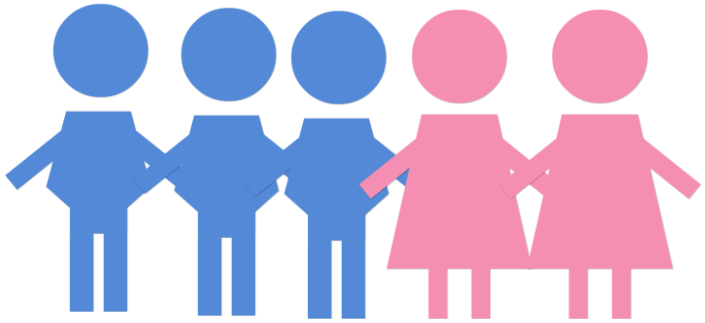
- Gini index
  - ✓ 右：1
  - ✓ 左：0.5
  - ✓ 加權：$1 \times \frac{2}{12} + 0.5 \times \frac{10}{12} \sim 0.58$

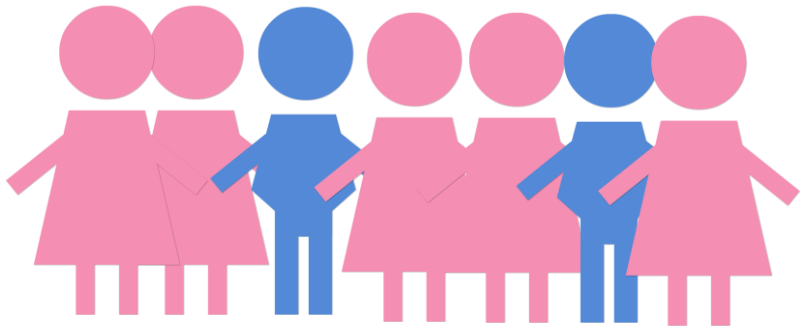


決策樹 Decision Tree

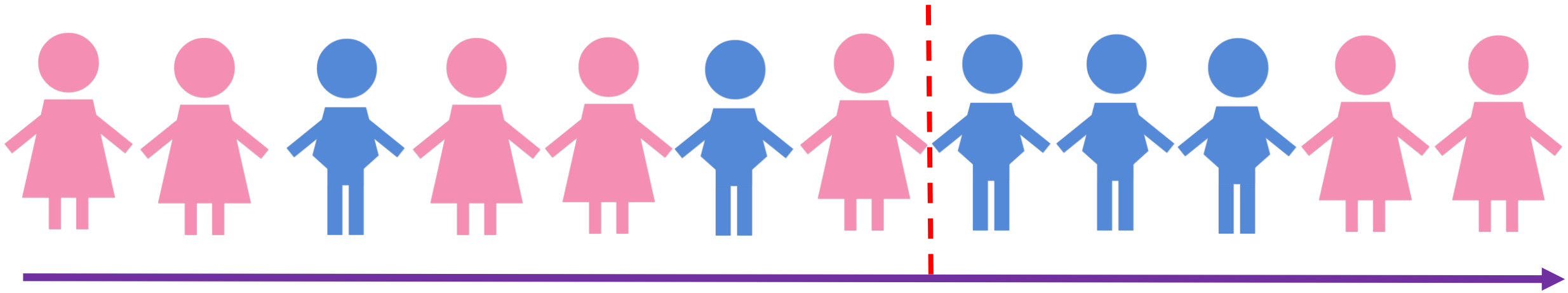


- Gini index
  - ✓ $p^2 + q^2$
  - ✓ p：正確機率、q：失敗機率
  - ✓ 右：$0.6^2 + 0.4^2 = 0.52$
  - ✓ 左：$\left(\frac{5}{7}\right)^2 + \left(\frac{2}{7}\right)^2 = 0.59$

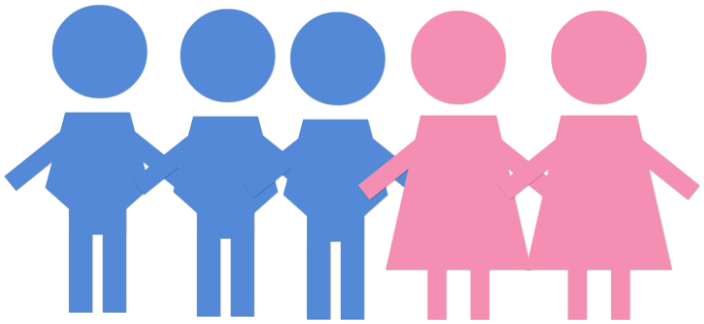


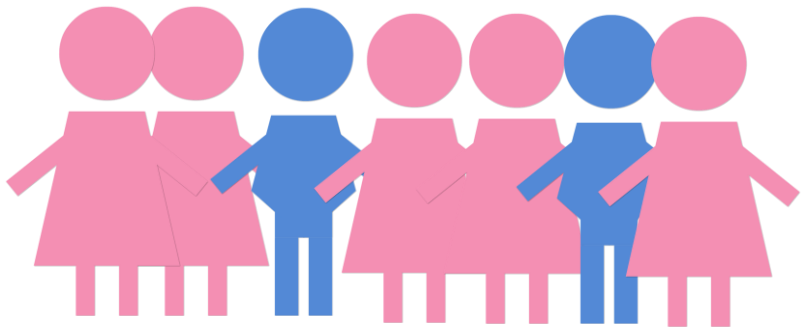


決策樹 Decision Tree

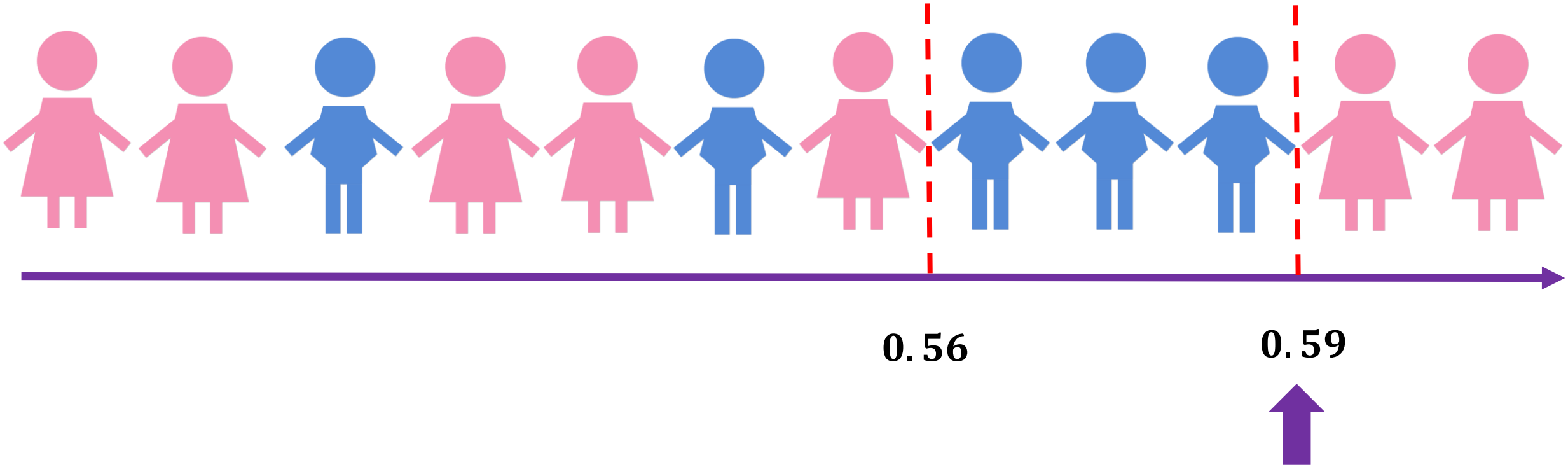


- Gini index
  - ✓ 右：0.52
  - ✓ 左： 0.59
  - ✓ 加權：$0.52 \times \frac{5}{12} + 0.59 \times \frac{7}{12} \sim 0.56$

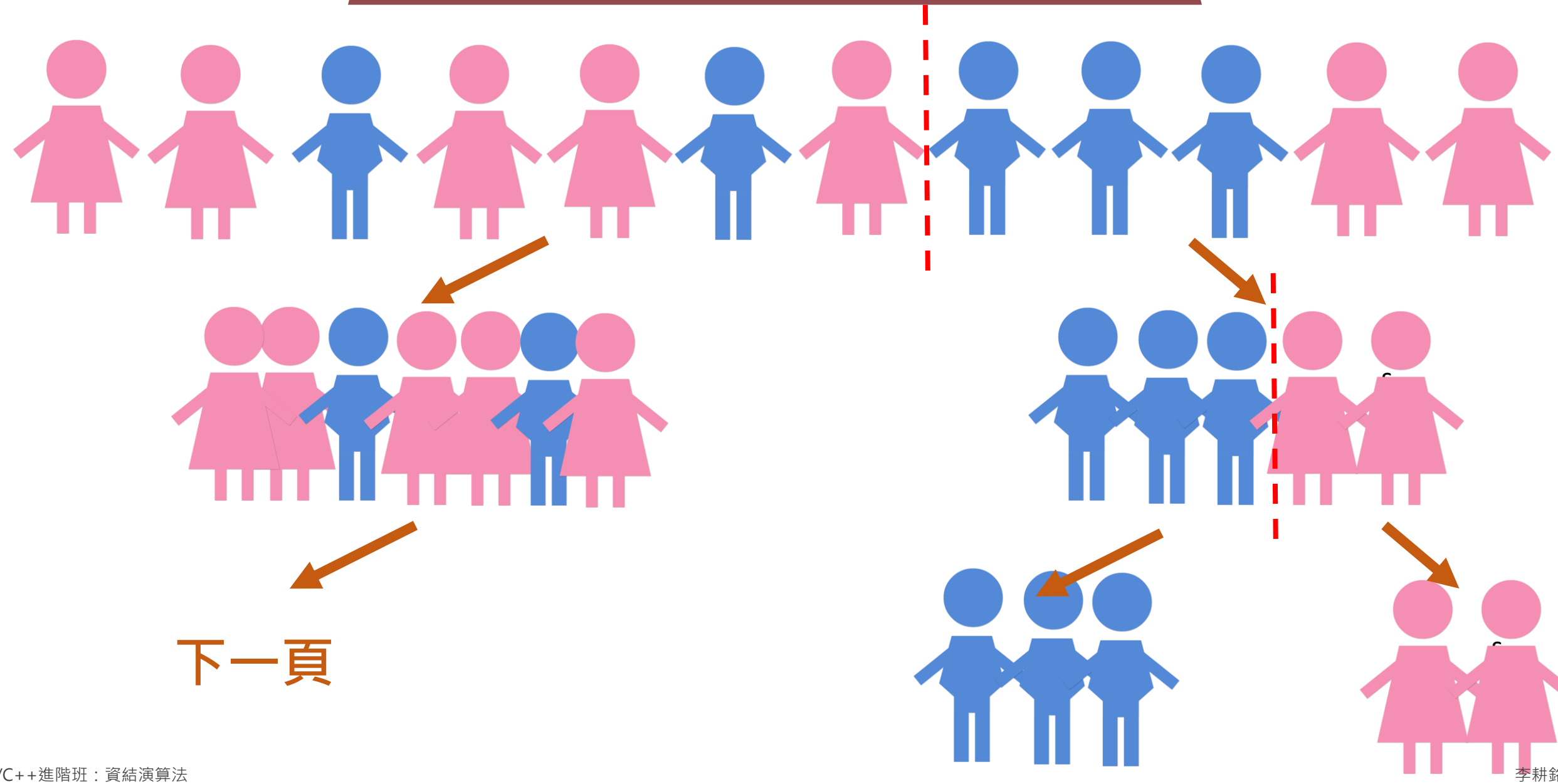




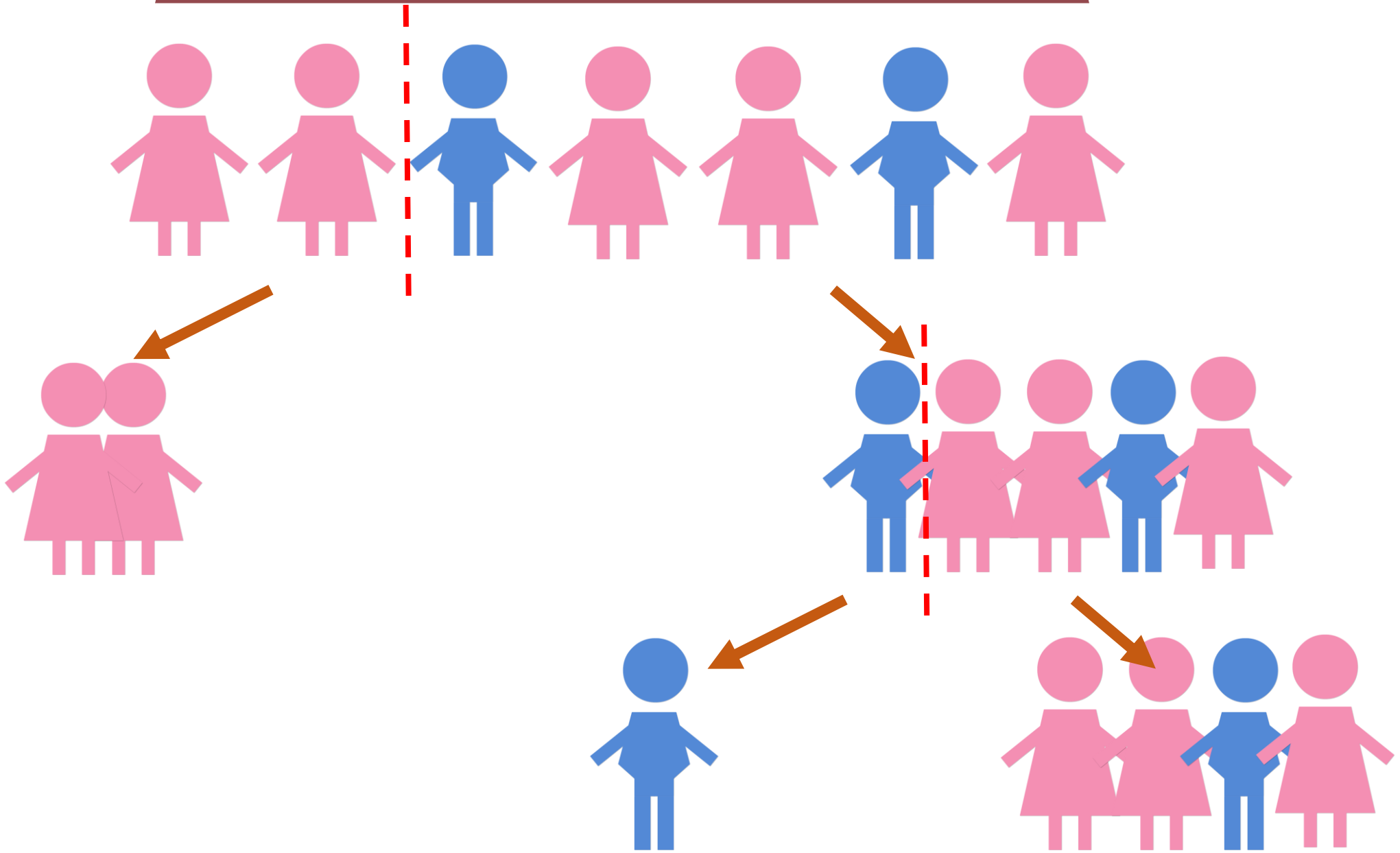
決策樹 Decision Tree



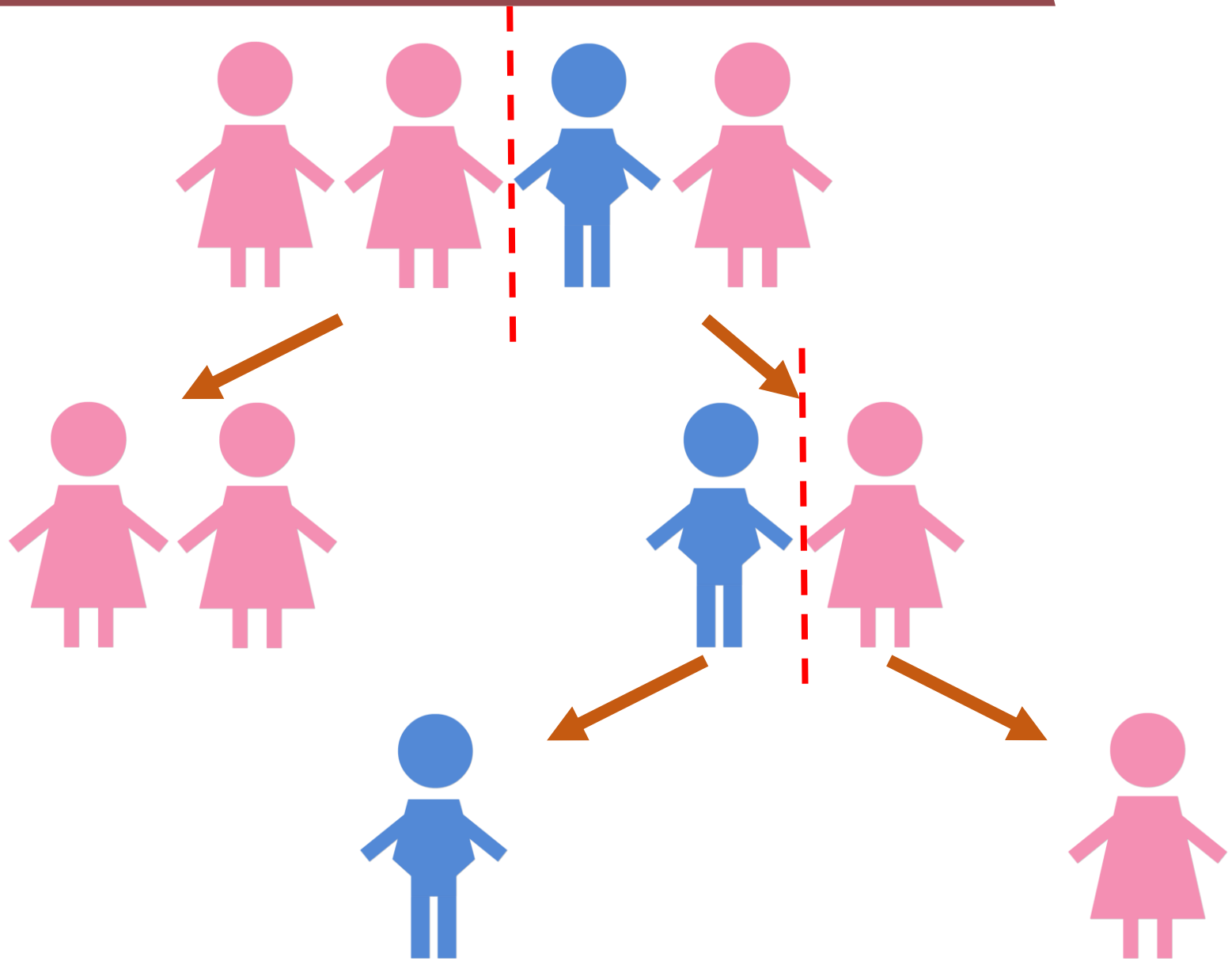
決策樹 Decision Tree



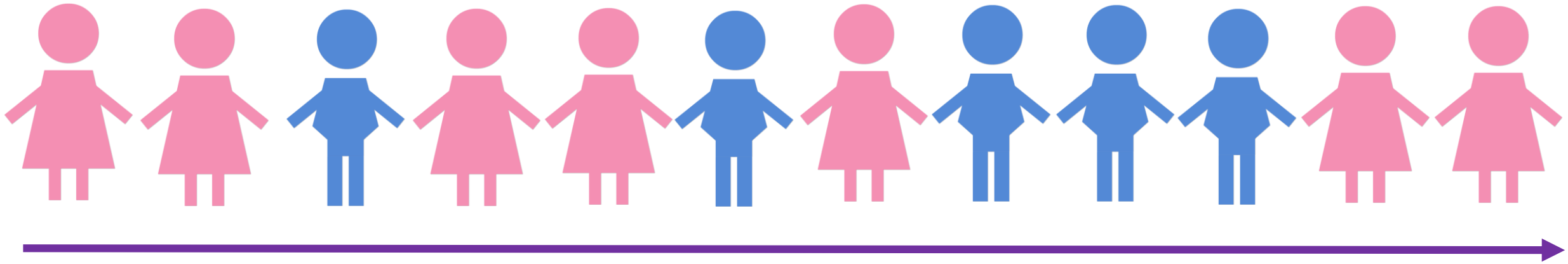
決策樹 Decision Tree



決策樹 Decision Tree



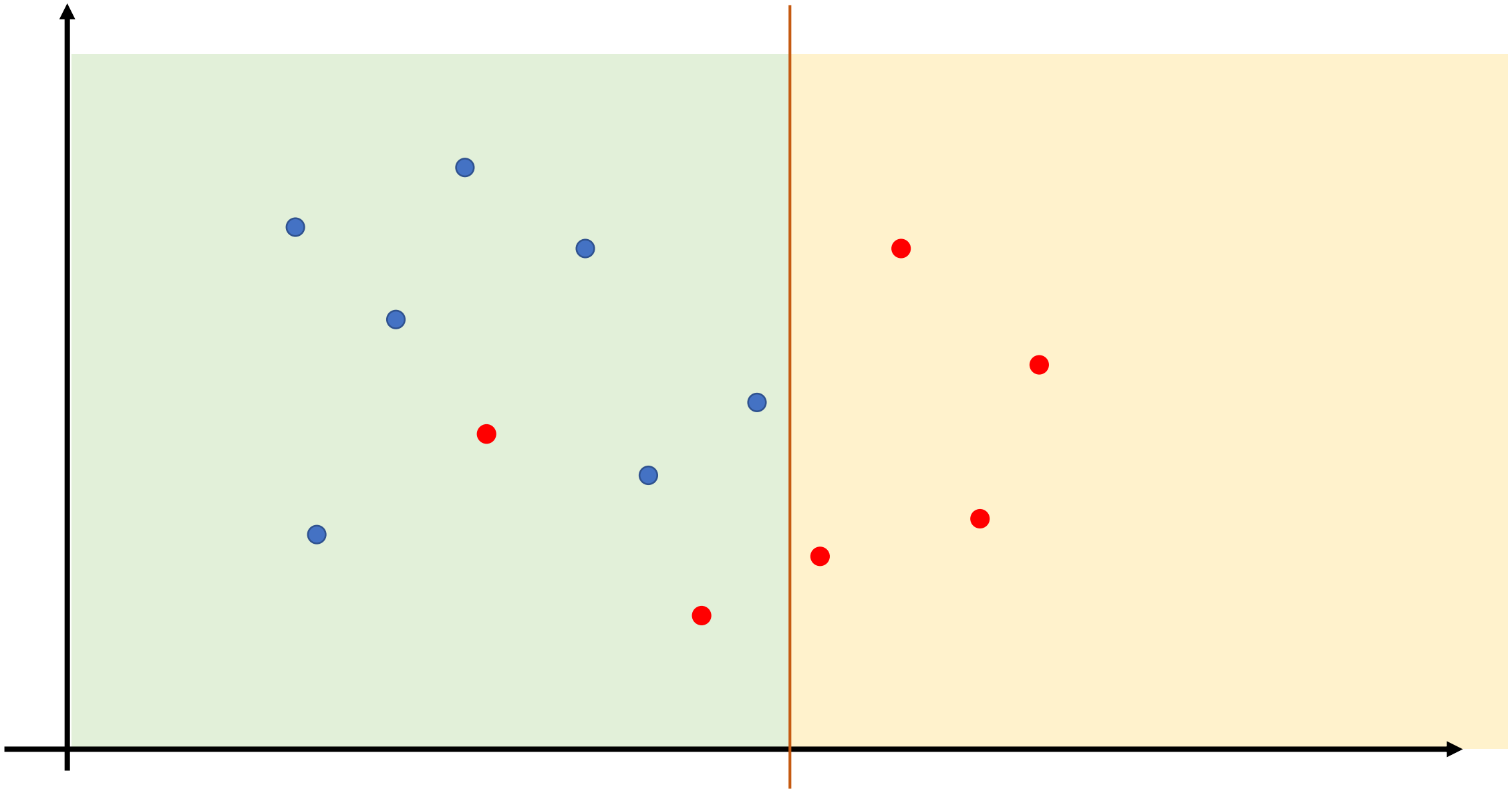
決策樹 Decision Tree



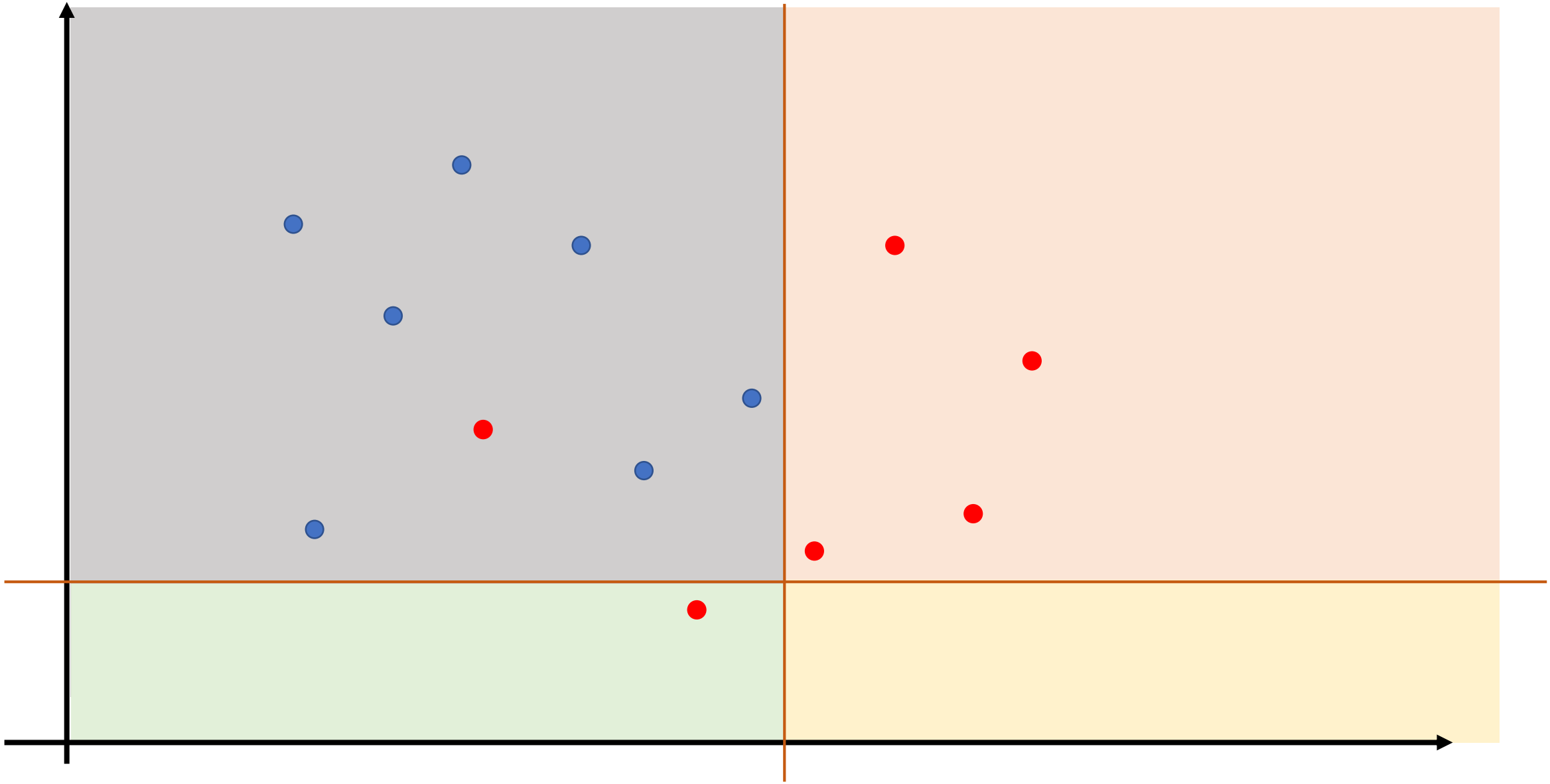
- 決策樹的生成步驟

1. 把屬於該節點的資料依照大小排序
2. 算出每個間距間的 Entropy 或 Gini index
3. 若不只一個維度，把所有維度做過一次
4. 取其中 Purity 最高或 Impurity 最小的點
5. 重複 1~4 步驟直到每個 Leaf node 下的資料都只有一種

決策樹 Decision Tree



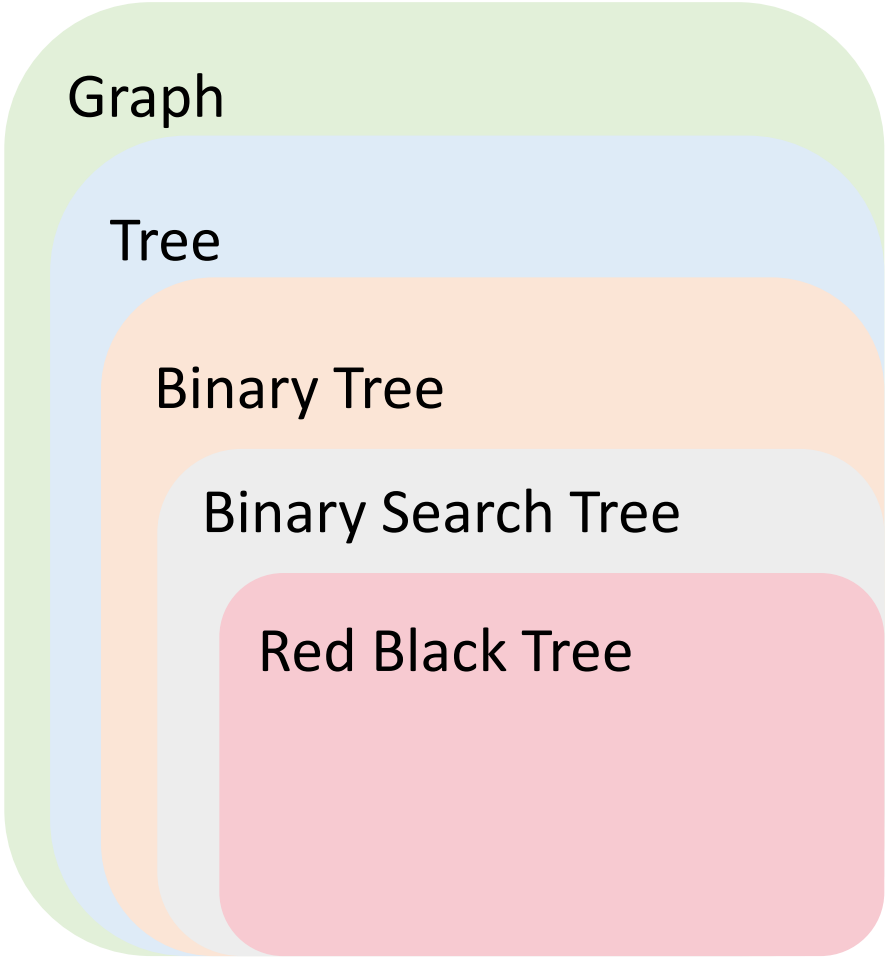
決策樹 Decision Tree



樹 (Tree) 與圖 (Graph)

樹的簡介&定義

- Graph
  - 由 node 跟 edge 組成
- Tree
  - 不能形成 cycle
  - 除根節點外的節點都僅有一個父節點
- Binary Tree
  - 每個節點的分歧度 ≤ 2



樹的簡介&定義

- Binary Search Tree (BST)
  - 資料的插入有次序性 (左小右大)
- Red Black Tree
  - 在 BST 上的每個節點添加紅或黑
  - 平衡左右兩子樹的發展
  - 避免形成斜曲二元樹