

CHAPTER 7

Sorting

Sorting

Categories:

- (1) Internal or External Sorting?
- (2) Stable or Unstable Sorting?
- (3) Time Complexity

Quick Sort

How:

Given:

$(R_0, R_1, R_2, R_3, \dots, R_{n-3}, R_{n-2}, R_{n-1})$

i j

After first pass:

$R_1, \dots, R_{S(i)-1}, R_0, R_{S(i)}, R_{S(i)+1}, \dots, R_{S(n-1)}$

two partitions

Quick Sort

Example:

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	left	right
26	5	37	1	61	11	59	15	48	19	0	9
11	5	19	1	15	26	59	61	48	37	0	4
1	5	11	19	15	26	59	61	48	37	0	1
1	5	11	15	19	26	59	61	48	37	3	4
1	5	11	15	19	26	48	37	59	61	6	9
1	5	11	15	19	26	37	48	59	61	6	7
1	5	11	15	19	26	37	48	59	61	9	9
1	5	11	15	19	26	37	48	59	61		

Quick Sort (Algorithm)

void procedure QuickSort(list, m , n)

if ($m < n$) then

$i = m$, $j = n + 1$, $p.k. = \text{list}[m].\text{key}$

Repeat

repeat $i = i + 1$ until $\text{list}[i].\text{key} \geq p.k.$

repeat $j = j - 1$ until $\text{list}[j].\text{key} \leq p.k.$

if ($i < j$) then swap(list[i], list[j])

Until $i \geq j$

swap(list[m], list[j])

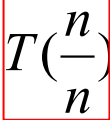
QuickSort(list, m , $j - 1$)

QuickSort(list, $j + 1$, n)

Quick Sort

Time Complexity:

Best Case

$$\begin{aligned}T(n) &= c \times n + 2 \times T\left(\frac{n}{2}\right) \\&= 4 \times T\left(\frac{n}{4}\right) + 2cn \\&\vdots \\&= n \times T\left(\frac{n}{n}\right) + \log n \times cn \\&= n + cn \log n \\&= O(n \log n)\end{aligned}$$


Quick Sort

Time Complexity:

Worst Case

$$T(n) = c \times n + T(0) + T(n-1)$$

$$= T(n-1) + cn$$

$$= \overset{0}{\boxed{T(0)}} + c \times (1 + 2 + 3 \dots + n)$$

$$= c \times \frac{n \times (n+1)}{2}$$

$$= O(n^2)$$

Quick Sort

Time Complexity:

Average Case

$$\begin{aligned} T(n) &= \frac{1}{n} \times \sum_{i=1}^n (T(i) + T(n-i)) + cn \\ &= O(n \log n) \end{aligned}$$

Min-Max Heap

A *double-ended priority queue* is a data structure that supports the following operation:

- (1) Insert
 - (2) Delete Max
 - (3) Delete Min
- 
- Why?

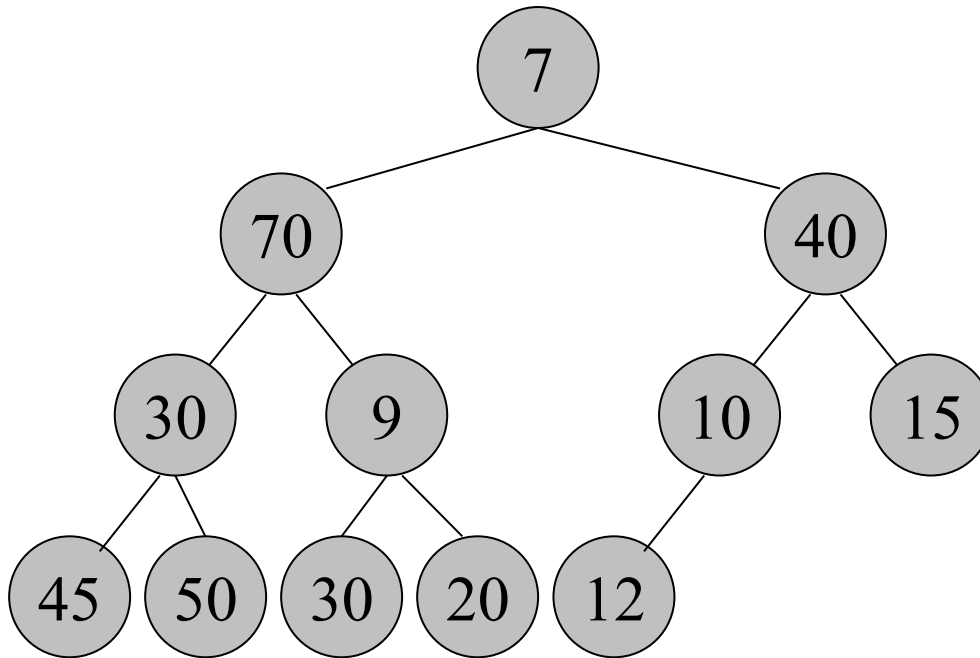
Min-Max Heap

Definition:

A *min-max heap* is a complete binary tree such that if it is not empty, each element has a field called *key*.

- (1) Alternating levels of this tree are min levels and max levels, respectively.
- (2) The root is on a min level
- (3) Let x be any node in a min-max heap. If x is on a min (max) level then the element in x has the minimum (maximum) key from among all elements in the subtree with root x . We call this node a *min* (*max*) node.

Ex)



..... min

..... max

..... min

..... max

Algorithm Insert:

Step1: 將 x 置於 last node 之後

Step2: 假設 x 的 parent 為 p

case 1: 若 p 位於 max-level

if ($x > p$) then

swap (x, p)

verify max (heap, p, x)

else

verify min (heap, n, x)

case 2: 若 p 位於 min-level

if ($x < p$) then

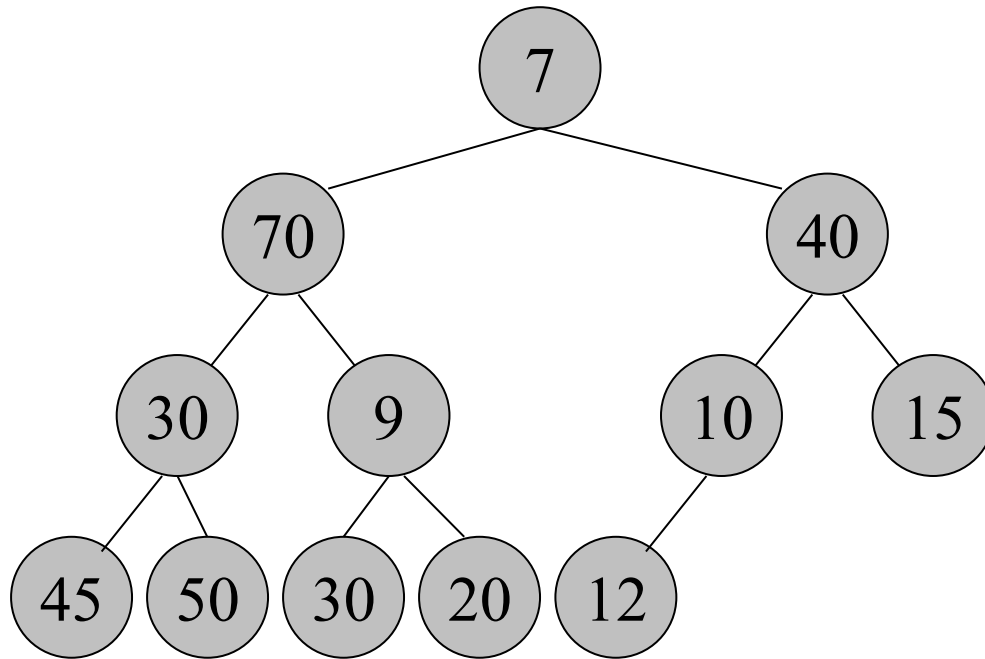
swap (x, p)

verify min (heap, p, x)

else

verify max (heap, n, x)

Ex)



Insert 5, then insert 80?

Ex)

依據下列資料建立 min-max heap

26, 5, 33, 77, 2, 6, 19

Procedure min-max heap (h, n, x):

$n = n + 1$

$p = n / 2$

if $p == 0$ then $h[1] = x$

else case level (p)

min:

if $x < h[p]$ then

swap ($h[n]$, $h[p]$)

verify-min (h , p , x)

else

verify-max (h , n , x)

max:

if $x > h[p]$ then

swap ($h[n]$, $h[p]$)

verify-max (h , p , x)

else

verify-min (h , n , x)

所在位置

Procedure verify-max (h, i, x):

```
Gp=(i/4)
while (Gp ≠ 0) do
  if x > h[Gp] then
    swap (h[i], h[Gp])
    i=Gp
    Gp=(i/4)
  else
    Gp=0
```


Procedure verify-min (h, i, x):

```
Gp=(i/4)
while (Gp ≠ 0) do
  if x < h[Gp] then
    swap (h[i], h[Gp])
    i=Gp
    Gp=(i/4)
  else
    Gp=0
```

Time Complexity:

$$O(\log n)$$

約比較樹高的一半



Algorithm Delete minimum element:

Step1: Remove root  Why?

Step2: 令 last node 為 x , 將 x 插入一個 root 為空的 min-max heap 中

case 1: 若 root 沒有 child, 則 x 直接置於 root

case 2: 若 root 有 child, 但沒有 Gchild, 則找出 child 中最小值, 令為 k

if ($k < x$) then swap (k, x)

else 將 x 直接置於 root

case 3: 若有 Gchild, 則找出孫子中最小值, 令為 k , 且假設 k 的 parent node 為 p

if ($k < x$) then swap (k, x)

if ($x > p$) then swap (p, x)

else goto step 2

else 將 x 直接置於 root

Algorithm Delete maximum element:

Step1: Remove $h[2]$ or $h[3]$ 中最大者

Step2: 令 last node 為 x , 將 x 插入一個 root 為空的 max-min heap 中

case 1: 若 root 沒有 child, 則 x 直接置於 root

case 2: 若 root 有 child, 但沒有 Gchild, 則找出 child 中最大值, 令為 k

if ($k > x$) then swap (k, x)

else 將 x 直接置於 root

case 3: 若有 Gchild, 則找出孫子中最大值, 令為 k , 且假設 k 的 parent node 為 p

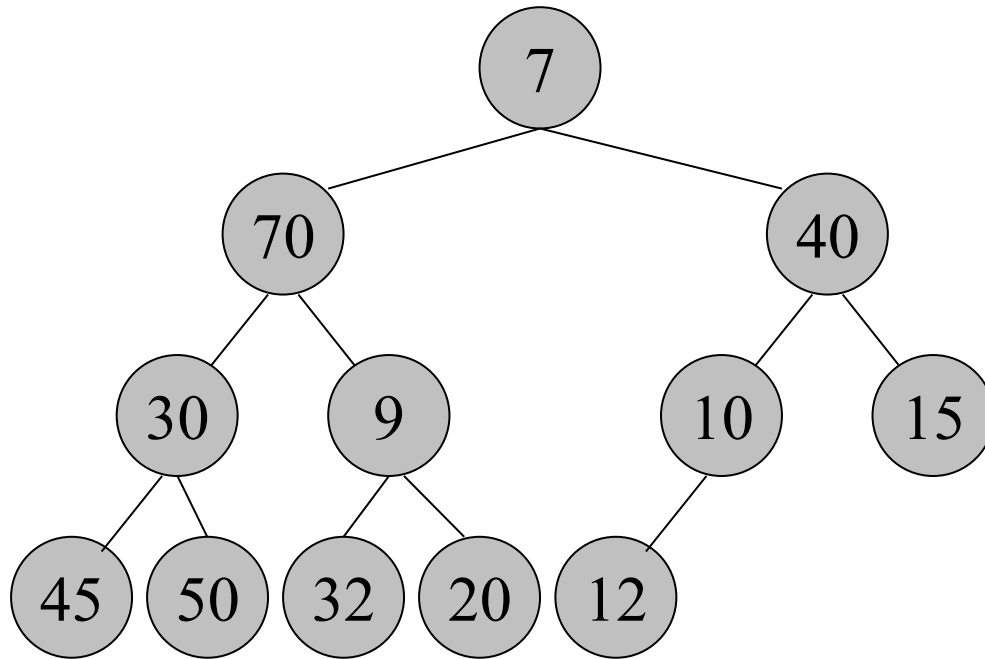
if ($k > x$) then swap (k, x)

if ($x < p$) then swap (p, x)

else goto step 2

else 將 x 直接置於 root

Ex)



Delete minimum element twice, then delete maximum?

Time Complexity:

$$O(\log n)$$

Double-ended Heap (Deap)

A *double-ended priority queue* is a data structure that supports the following operation:

- (1) Insert
- (2) Delete Max
- (3) Delete Min

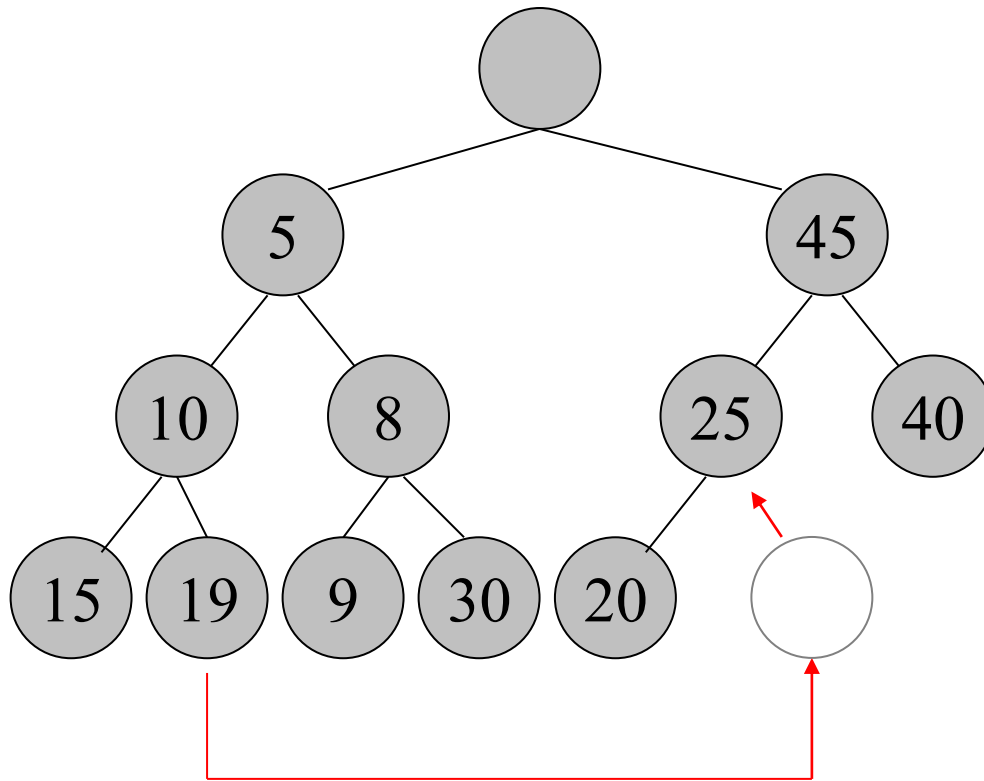
Deap

Definition:

A *deap* is a complete binary tree that is either empty or satisfies the following properties:

- (1) The root contains no element.
- (2) The left subtree is a min-heap
- (3) The right subtree is a max-heap
- (4) If the right subtree is not empty, then let i be any node in the left subtree. Let j be the corresponding node in the right subtree. If such a j does not exist, then let j be the node in the right subtree that corresponds to the parent of i . The key in node i is less than or equal to the key in j .

Ex)



min

max

$$j = i + 2^{\lfloor \log(i+1) \rfloor - 1}$$
$$\text{if } (j > n) \text{ then } (j = \frac{j}{2})$$

Why?

Insert:

Step1: 將 x 置於 last node 之後

Step2: case 1: 若 x 位於 min-heap 中, 找出 n 在 max-heap 中相對應 node j

if ($x.key \leq j.key$) then

min-heap-insert (D, n, x)

else

swap ($x.key, j.key$)

max-heap-insert (D, j, x)

case 2: 若 x 位於 max-heap 中, 找出 n 在 min-heap 中相對應 node j

if ($x.key \geq j.key$) then

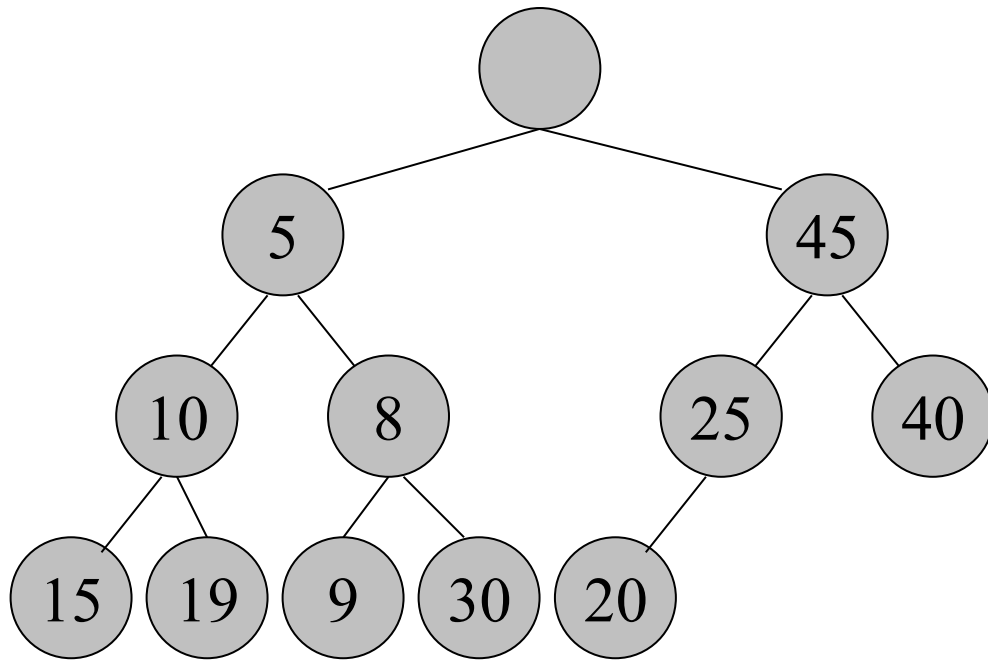
max-heap-insert (D, n, x)

else

swap ($x.key, j.key$)

min-heap-insert (D, j, x)

Ex)



Insert 4, 60?

Ex)

依據下列資料建立 Deap

26, 5, 33, 77, 19, 2, 8

Time Complexity:

$$O(\log n)$$

Algorithm Delete minimum:

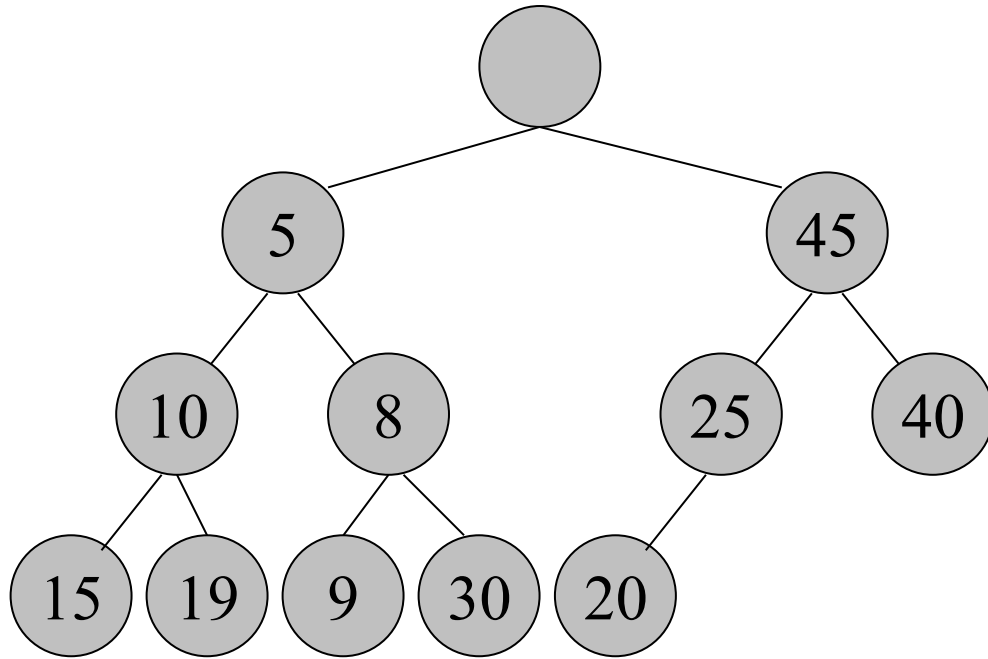
Step1: 移走 Deap[2]

Step2: 將 last node 設為 x

Step3: 在 min-heap 中, 逐次找出子點中最小值,
往上遞補父點之空缺, 最後必在 leaf 中
有一空缺 (假設此點位於 i)

Step4: Deap_Insert (D, i, x)

Ex)



Delete minimum twice?

Algorithm Delete maximum:

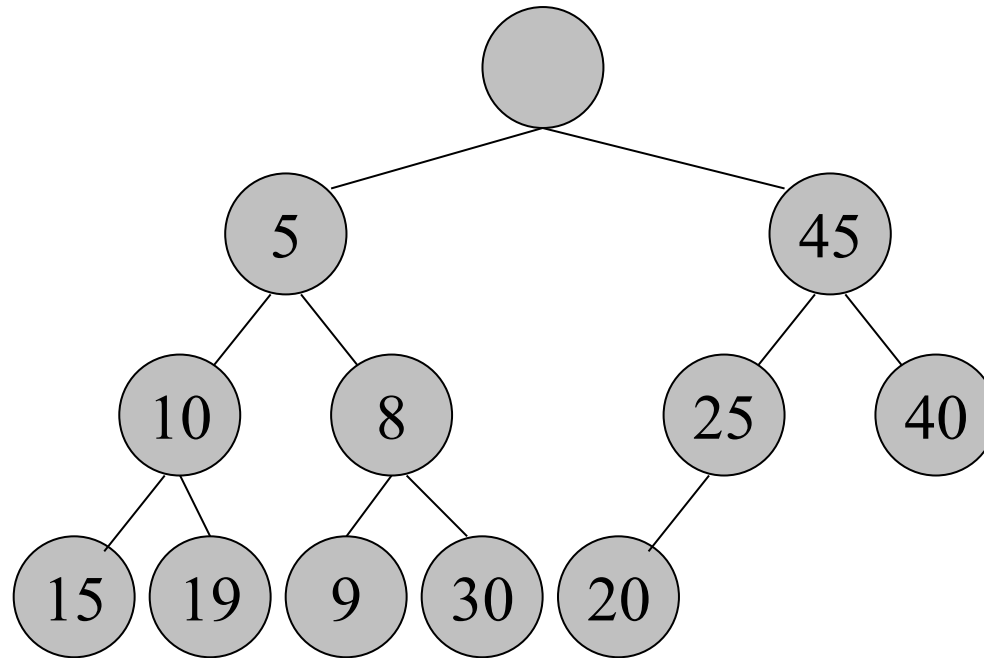
Step1: 移走 Deap[3]

Step2: 將 last node 設為 x

Step3: 在 max-heap 中, 逐次找出子點中最大值,
往上遞補父點之空缺, 最後必在 leaf 中
有一空缺 (假設此點位於 i)

Step4: Deap_Insert (D, i, x)

Ex)



Delete maximum?

Time Complexity:

$$O(\log n)$$

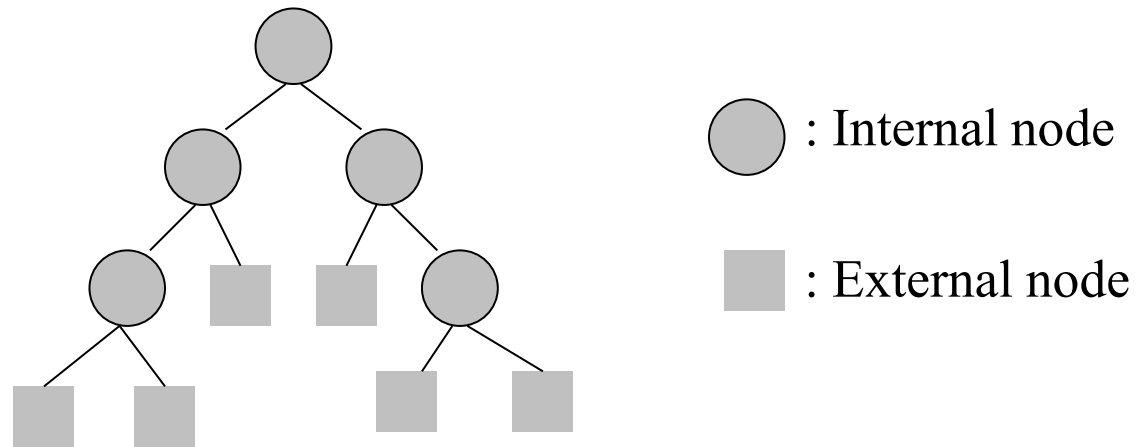
Summary:

	Min-Max Heap and Deap
Insert	$O(\log n)$
Delete Max	$O(\log n)$
Delete Min	$O(\log n)$
Search Max	$O(1)$
Search Min	$O(1)$

Extend Binary Tree

一二元樹具有 n 個 nodes, 若以 link-list 表示, 則會有 $n+1$ 條空 link, 在這些 null link 上加上一個特定 node 稱為 External node (or Failure node), 而此種具有 External node 的二元樹稱之為 Extended Binary Tree.

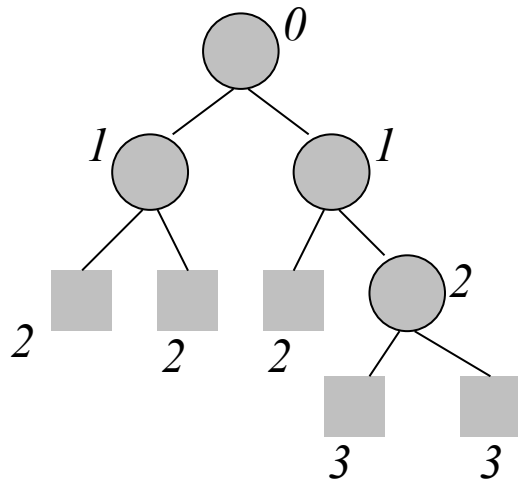
Why?



Internal path and External path:

$$I = \sum_{i=1}^n (\text{root到内部node之路徑長度})$$

$$E = \sum_{i=1}^{n+1} (\text{root到内部node之路徑長度})$$



$$I=0+1+1+2=4$$

$$E=2+2+2+3+3=12$$

$$**E=I+2n**$$

Theorem:

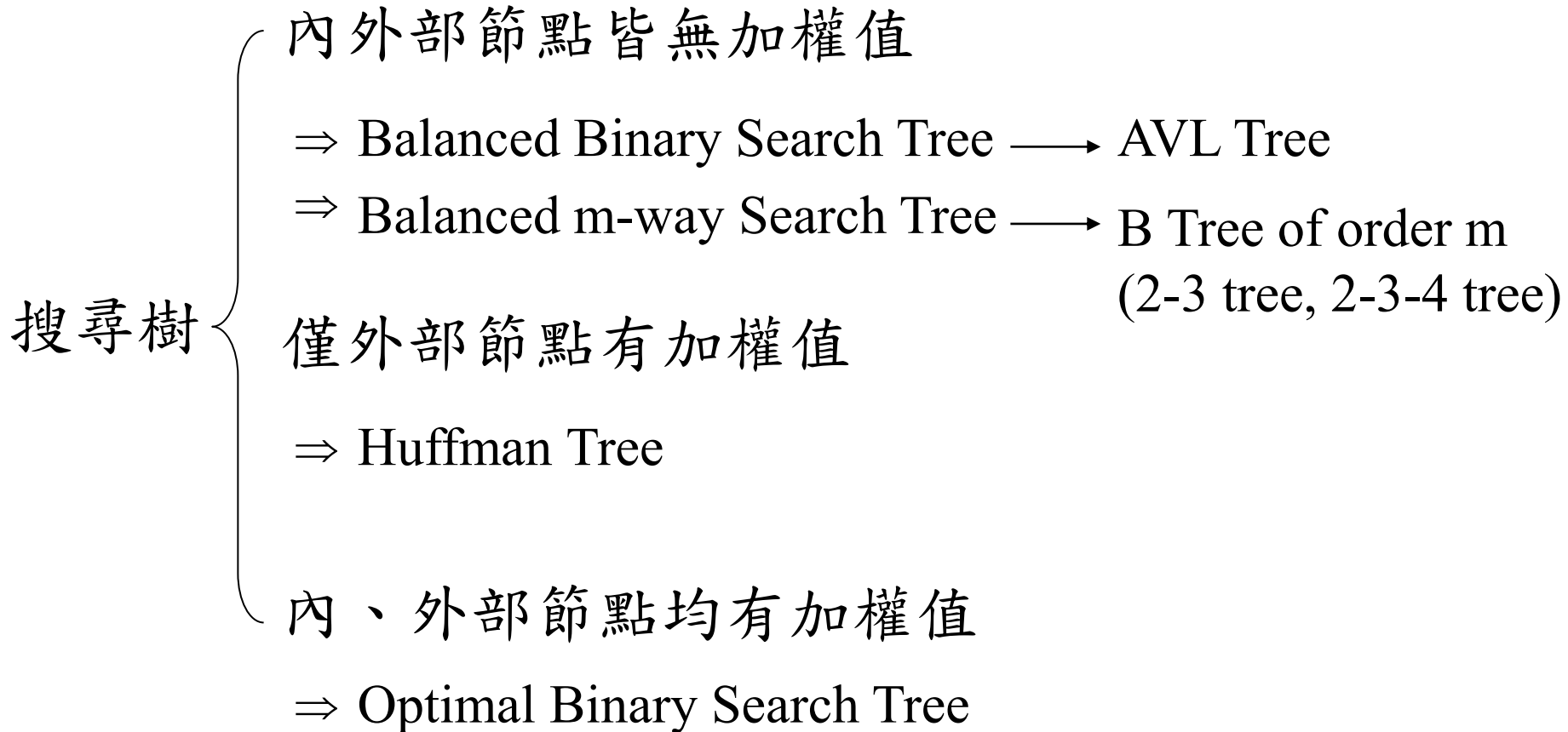
$$E=I+2n$$

Hint: induction by n

Ex) 在一個具有 n 個內部節點的 Skewed Extended B.T. 其 I, E 值各為何？

Ex) 在一個高度為 k 的 Full Extended B.T. 中,
I, E 值各為何?

高等搜尋樹其搜尋成本之探討:



AVL Tree

Definition:

An empty binary tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height balanced *iff*

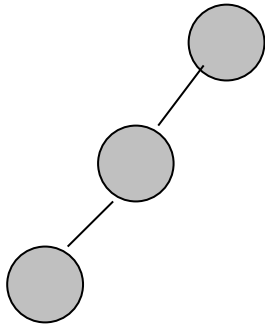
(1) $|h_L - h_R| \leq 1$

Balance factor

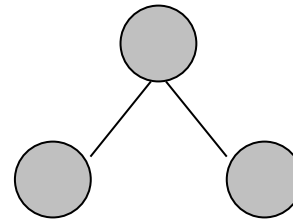
(3) T_L and T_R are height balanced

Recursive Definition

Ex)



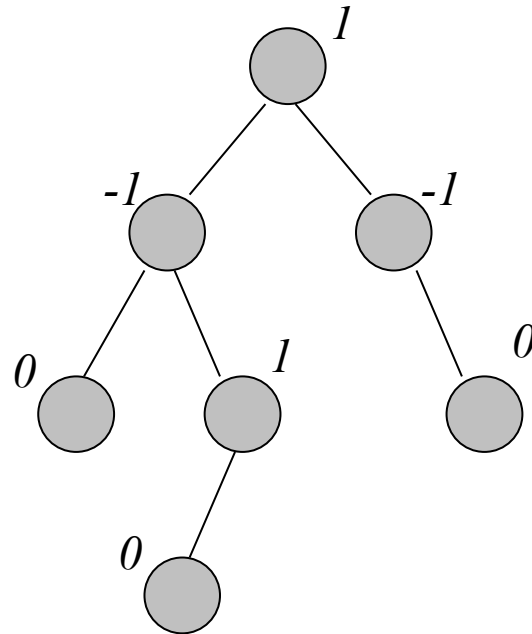
Searching cost = 2



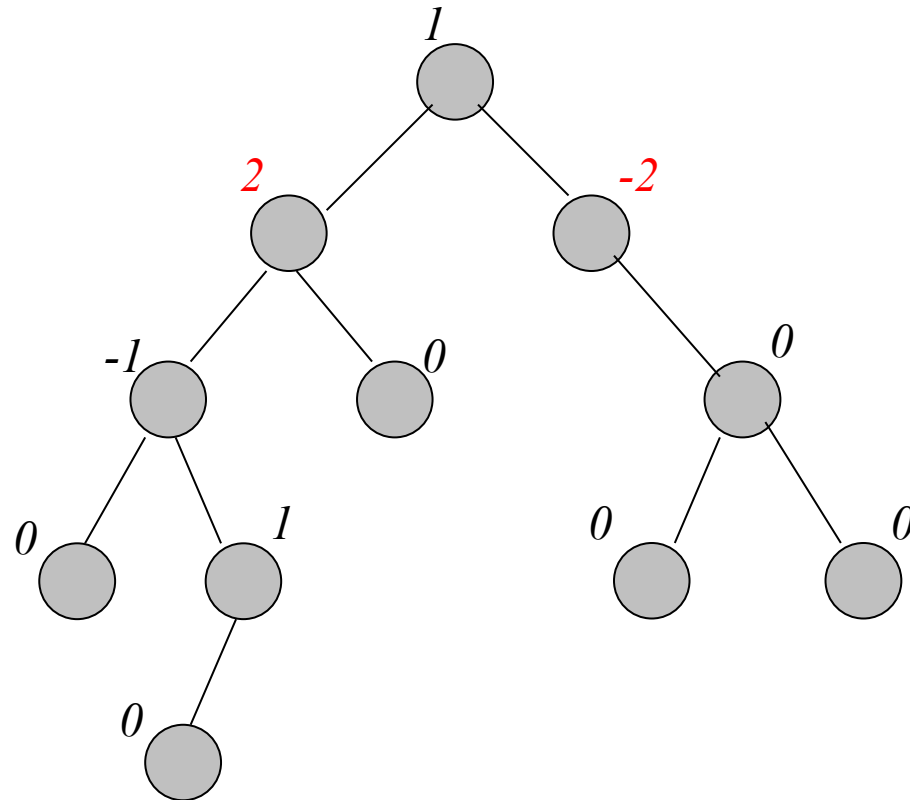
Searching cost = 1.67

⇒ 若內、外節點均無權重則越平衡越好

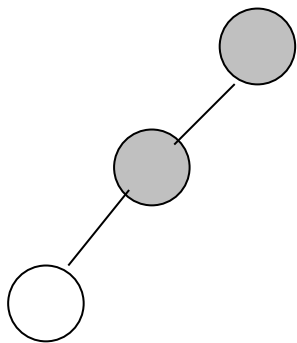
Ex) AVL Tree



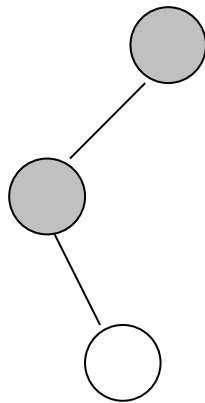
Ex) Non-AVL Tree



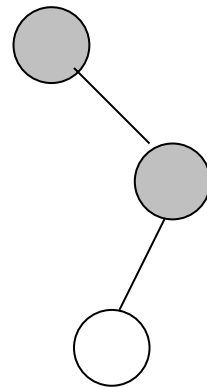
AVL Tree 不平衡的情況:



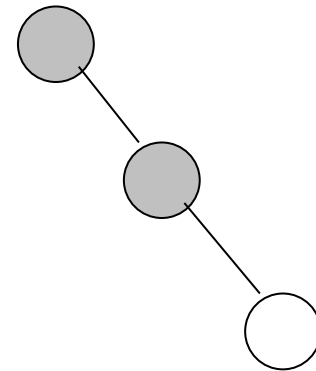
LL



LR



RL



RR

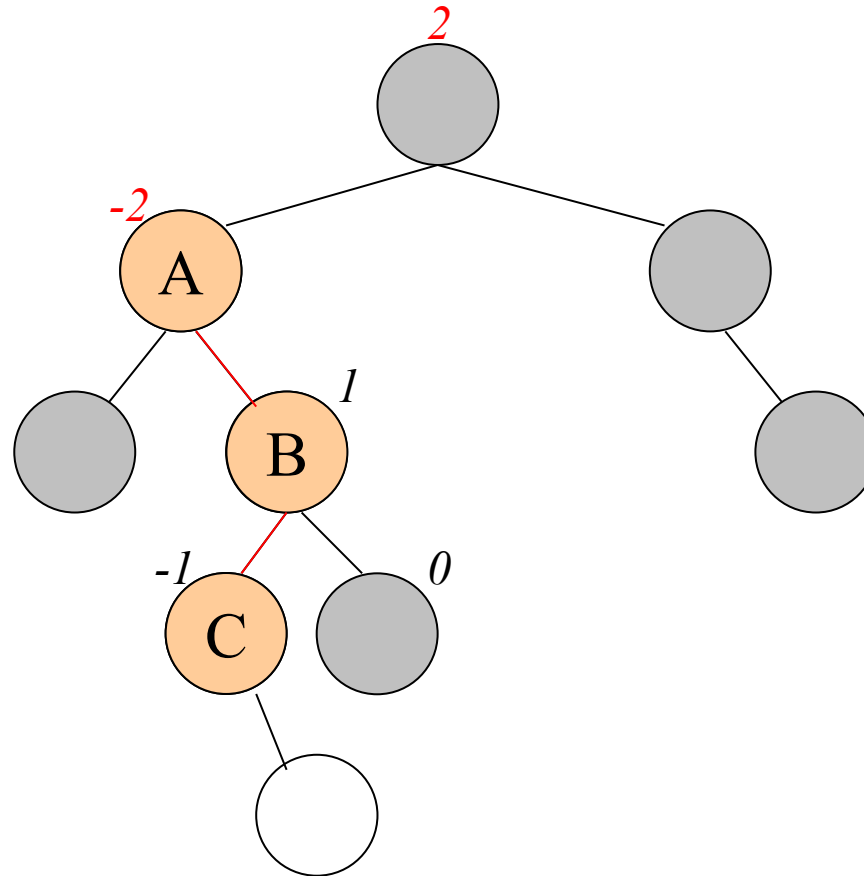
調整原則:

Step1: 找出最近之不平衡點, 再找出相關的三個點

Step2: 相關的三個點中, 將中間鍵值點往上拉, 小的在左, 大的在右

Step3: 左邊歸左邊, 右邊歸右邊

Ex)



Ex) Mar, May, Nov, Aug, Apr, Jan, Dec, Jul, Feb, Jan, Oct, Sep
依序輸入, 建立 AVL Tree

Ex) 依據下列資料 AVL Tree

26, 33, 77, 19, 2, 5, 4, 8, 6, 10

Time Complexity:

Operation	Sequential list	Linked list	AVL Tree
Search for x	$O(\log n)$	$O(n)$	$O(\log n)$
Search for k th item	$O(1)$	$O(k)$	$O(\log n)$
Delete x	$O(n)$	$O(1)$	$O(\log n)$
Delete k th item	$O(n-k)$	$O(k)$	$O(\log n)$
Insert x	$O(n)$	$O(1)$	$O(\log n)$
Output in order	$O(n)$	$O(n)$	$O(n)$

因為平衡



Theorem: 形成高度為 h 的 AVL Tree, 最少需要 $F_{h+2} - 1$ 個 node

Ex) 形成高度為 5 的 AVL Tree, 最少的 node 數? 最多的 node 數?

Ex) 一 AVL Tree 具有 15 個 node, 則
最大高度? 最小高度?

台大資工87) Give the following sequence of data, how many rotations are needed to build an AVL tree and how many pointers are rearranged during rotation?

Mon, Tue, Wed, Thu, Fri, Sat, Sun

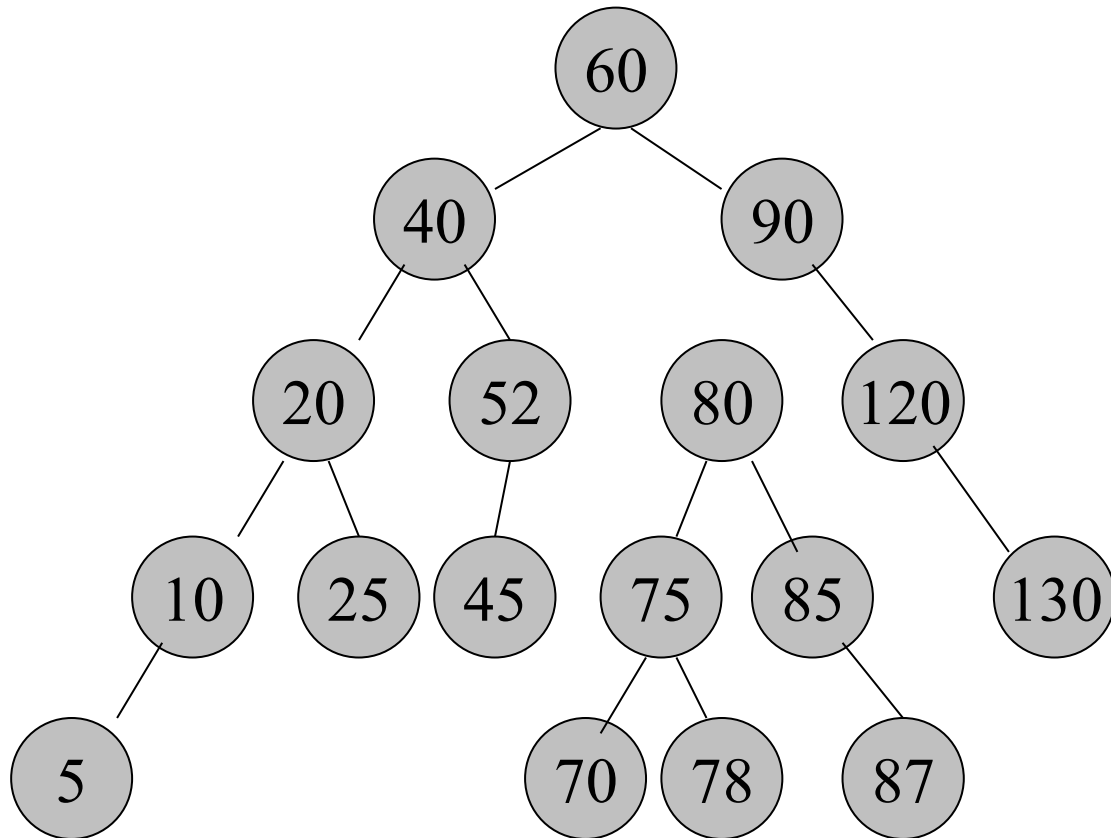
成大電機89) DAS, JFK, ZRH, HKG, KHH, FRA, ARN, LBA, MEX,
GLA, ORY, MAN, TPE, ORD, NAP

(1) Draw the AVL tree

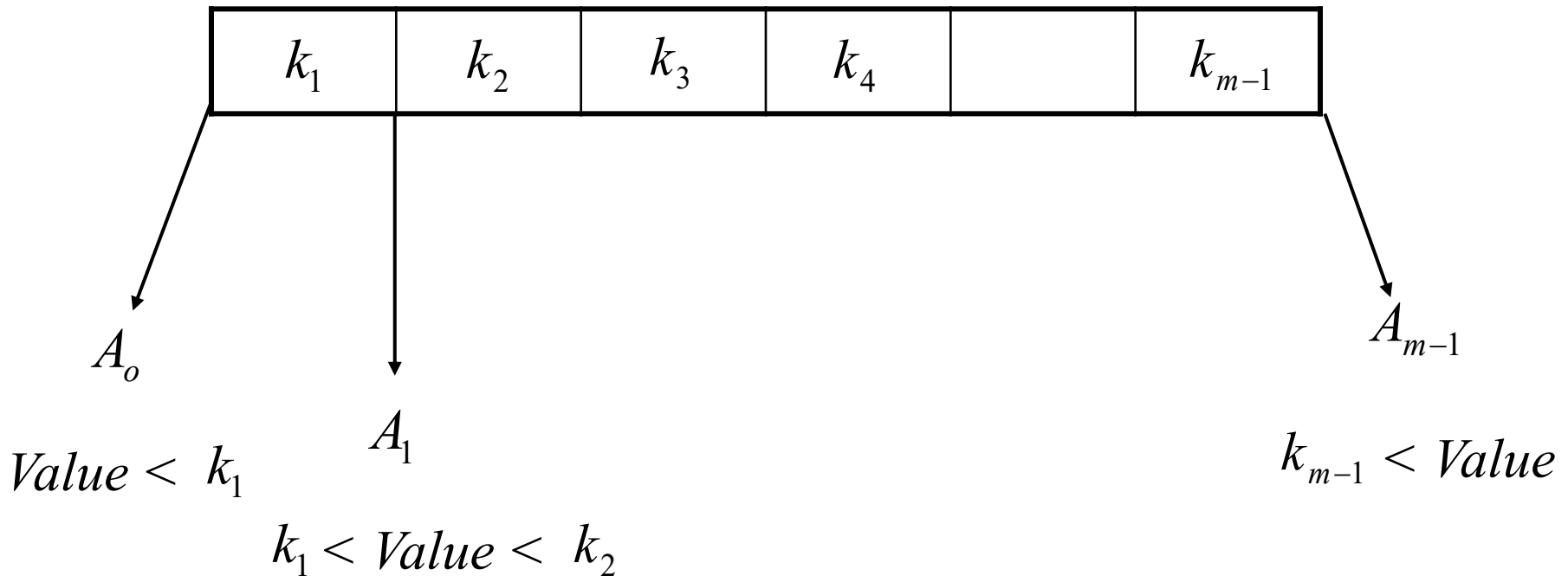
(2) Insert NYC, what kind of rotation is required?

Sketch this tree

高科大90) Insert 72



m-way Search Tree



(1) $k_1 \leq k_2 \leq k_3 \dots \leq k_{m-1}$

(3) A_i 亦為 m-way Search tree

Ex) 高度為 h 的 m -way Search tree, 其最多 node 個數為?
其最多 key 之個數為?

B Tree of order m

Definition:

為一 Balanced m-way Search tree, 常用在 External Search, 可以為空, 若不為空, 則需滿足

(1) root 至少有兩個 childs

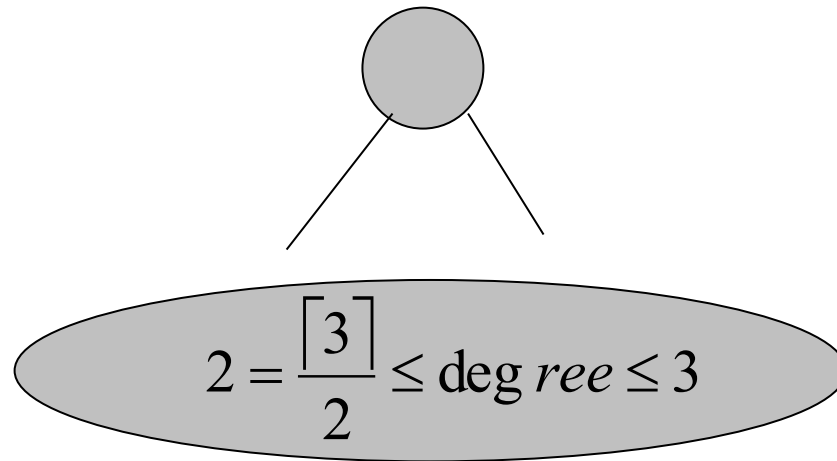
(2) 除了 root 及 failure node 之外, 其餘 node 之 degree 需介於 $\frac{\lceil m \rceil}{2}$ 到 m 之間

$$\frac{\lceil m \rceil}{2} \leq \text{degree} \leq m$$

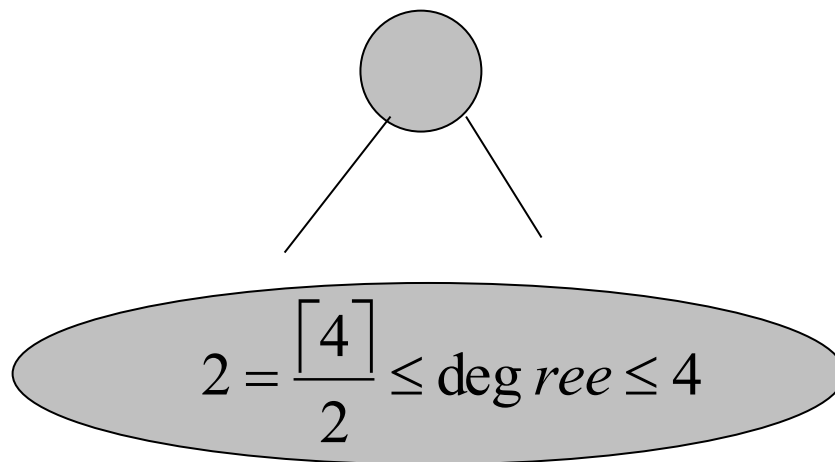
$$\frac{\lceil m \rceil}{2} - 1 \leq \text{No. of Key} \leq m - 1$$

(3) 所有的 Failure node (or leaf node) 皆位於同一 level

Ex) B tree of order 3



Ex) B tree of order 4



2-3-4 tree 所對應的 Binary Search Tree 就是 紅黑樹
若該 link 出現在原來的 2-3-4 樹中即為黑色, 否則為紅色

Insert:

Step1: 找到適當位置插入 x

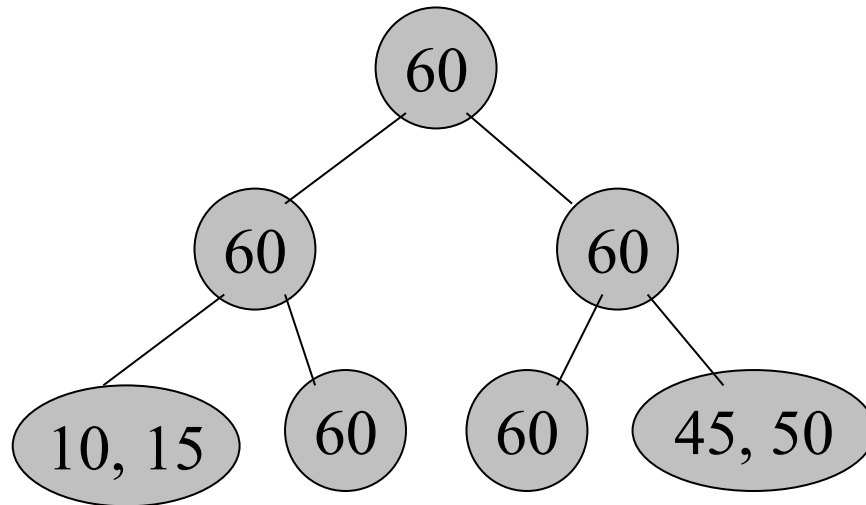
Step2: 檢查此 node 是否 overflow

case 1: 若沒有 overflow, 則 stop

case 2: 有 overflow, 則做 split 處理, 且針對
parent node, goto step 2

Split: 將 overflow node 中第 $\lceil \frac{m}{2} \rceil$ 個 key 上拉至父點,
其餘則分成兩個節點

Ex) B tree of order 3



(1) Insert 38

(2) Insert 55

(3) Insert 37

Ex) 給序 26, 5, 33, 49, 17, 2, 8, 6, 10 建立 B tree of order 3

台大資工90) Read the following data in the given order, and show the corresponding trees: 7, 8, 9, 2, 1, 5, 3, 6, 4

(1) Binary Search tree

(2) AVL tree

(3) 2-3 tree

Delete:

Step1: 找到 x 所在之 node

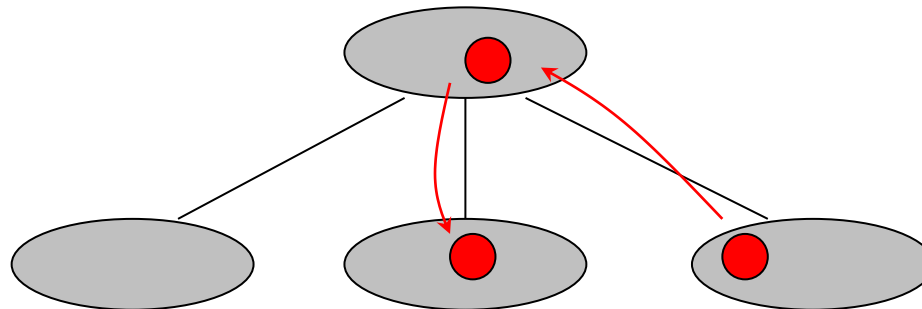
Step2: case 1: 若 x 位於 leaf node

(1) 刪除 x

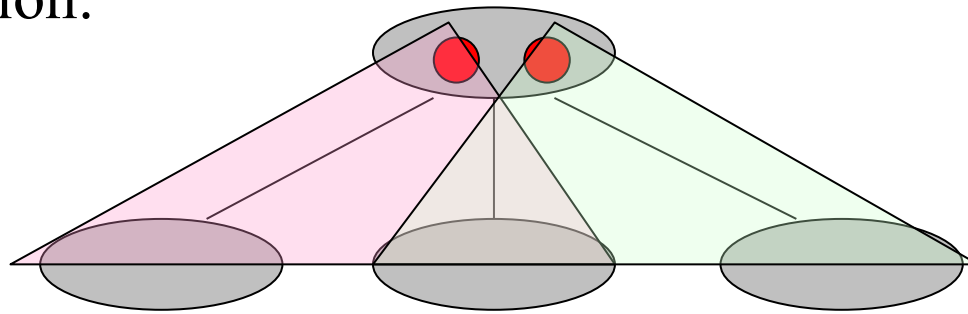
(2) 檢查有無 underflow, 若沒有, 則 stop, 有, 則做 rotation, 若無法做 rotation, 則做 combination, 做完後針對父點, goto (2)

case 2: 若 x 位於 Non-leaf node, 則取右子樹中最小鍵值或左子樹中最大鍵值取代 x , goto case 1.

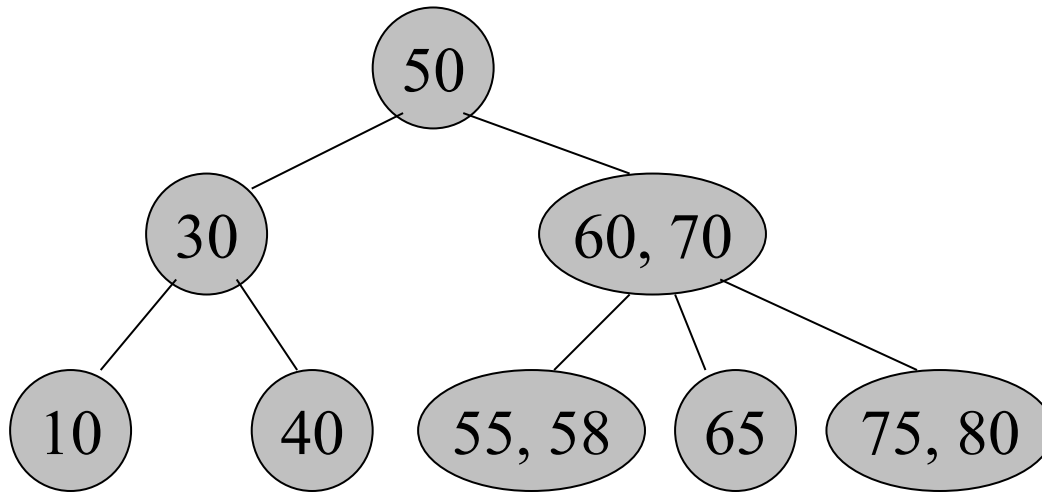
rotation:



combination:



Ex) B tree of order 3 如下

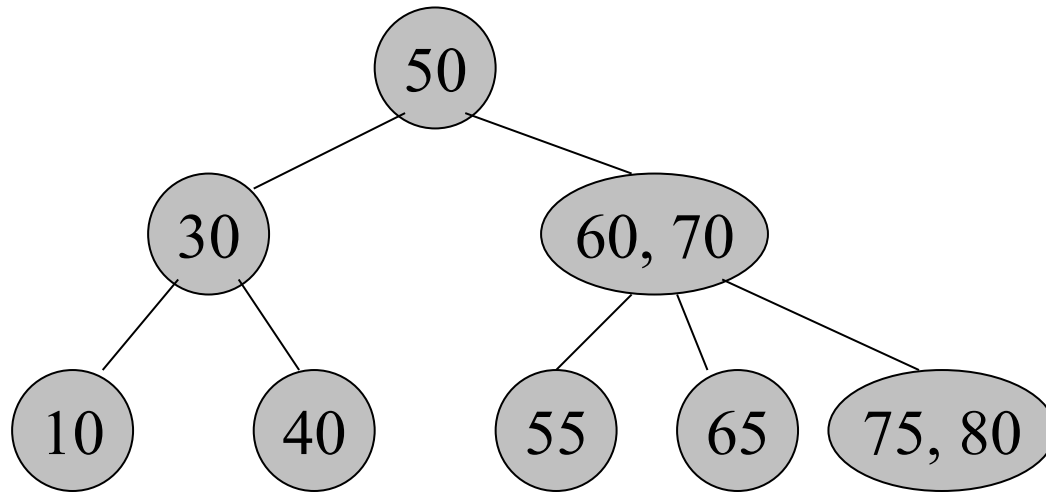


(1) Delete 58

(2) delete 65

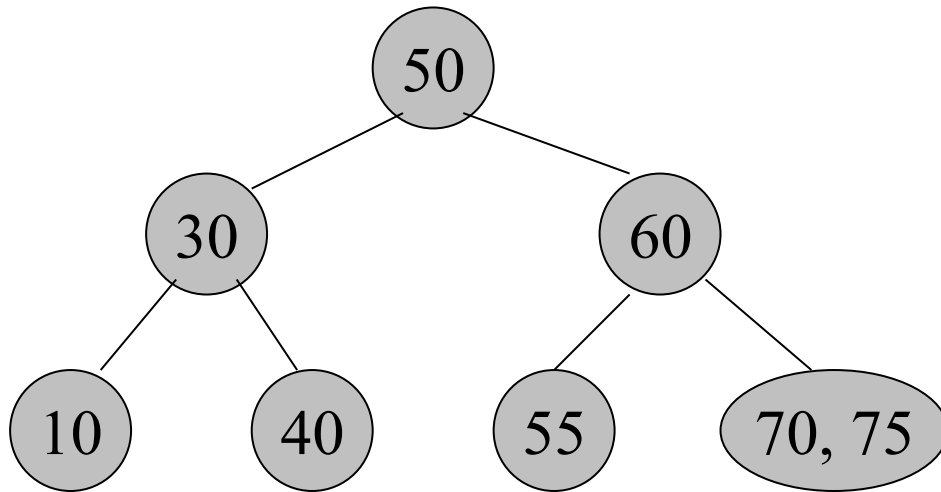
(3) Delete 80

Ex) B tree of order 3 如下



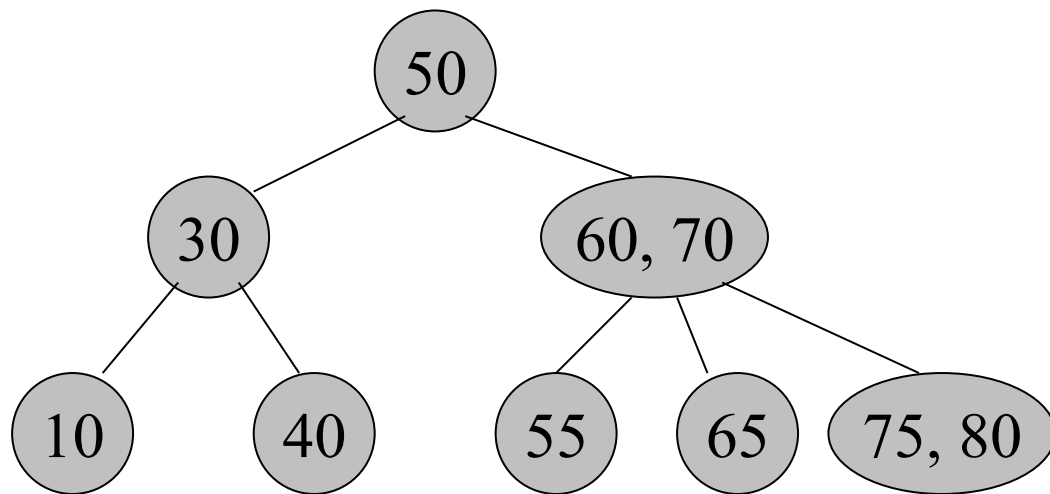
(1) Delete 10

Ex) B tree of order 3 如下



(1) Delete 10

Ex) B tree of order 3 如下



(1) Delete 70

(2) Delete 50

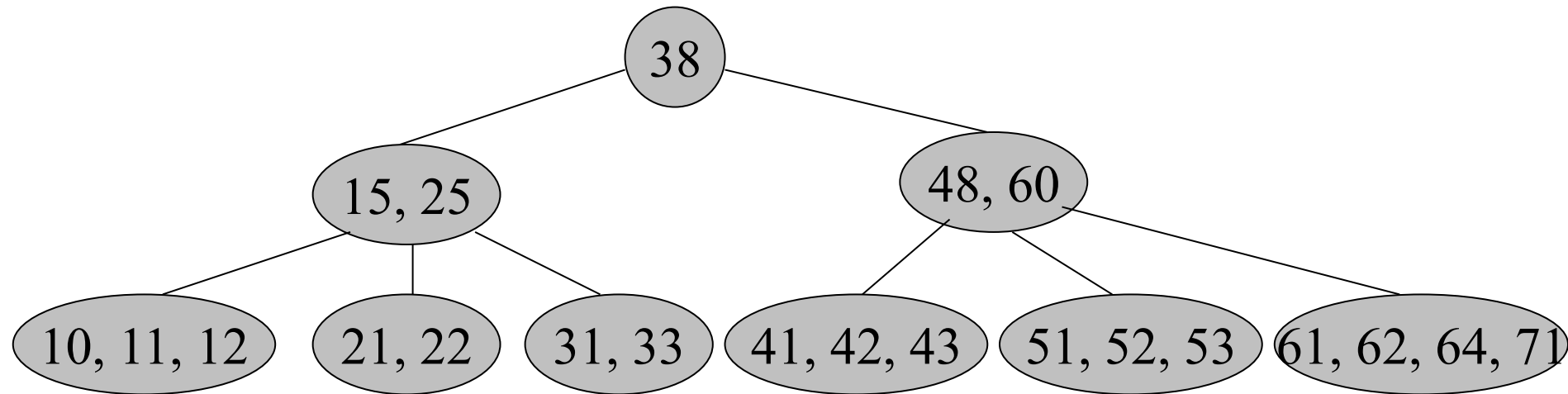
- 交大資工88) (1) Show the 2-3 tree results if we insert into an empty set, represented as 2-3 tree, the elements of
5, 2, 7, 0, 3, 4, 6, 1, 8, 9
(2) Show the result of deleting 3 from the 2-3 tree that results from (1)

台大資工) B-tree of order 5

(1) Insert 65

(2) Delete 21 from the original B-tree

(3) Delete 33 from the original B-tree



Ex) B tree of order m , 高度為 h , 則

- (1) 最多之節點個數
- (2) 最多之鍵值個數
- (3) 最少之節點個數
- (4) 最少之鍵值個數

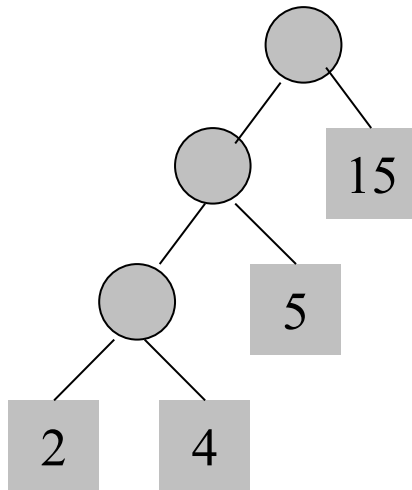
Weighted External Path Length

Definition:

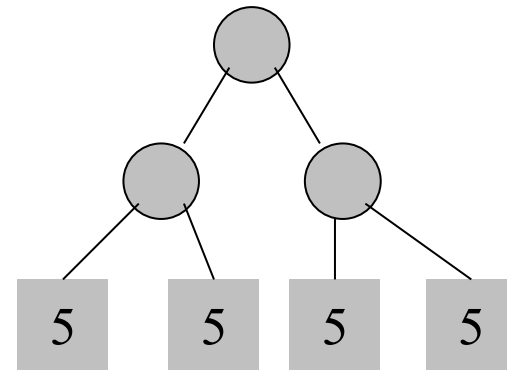
給予 $(n+1)$ 個外部節點不同的加權值 q_j ($j=1\sim(n+1)$),
則

$$W.E.P.L. = \sum_{j=1}^{n+1} [(root \text{ 到外部節點 } j \text{ 之路徑長}) \times q_j]$$

Ex)



W.E.P.L. = 43



W.E.P.L. = 52

Huffman Tree

Algorithm:

令 W = 所有外部節點之加權值集合

Step1: 自 W 中取出兩個最小加權值之外部節點

Step2: 對此兩個節點, 建立 Extended B. T. 並將此兩個加權值之和加入 W 中

Step3: Repeat Step 1~2 直到 W 中只剩唯一的加權值為止

Ex) 給予6個外部節點之加權值如下：

$$q_1 = 2 \quad q_2 = 3 \quad q_3 = 5 \quad q_4 = 7 \quad q_5 = 9 \quad q_6 = 13$$

求其 minimum W.E.P.L.

Ex) 6個 message 要傳輸 $m_1 m_2 m_3 m_4 m_5 m_6$, 其出現之機率分別為 $2/39, 3/39, 5/39, 7/39, 5/39, 13/39$, 則

- (1) Encoding/Decoding Tree 為何？
- (2) 各 message 之編碼內容為何？
- (3) 平均所需的編碼位元長度為何？

Procedure Huffman (w, n):

Begin

for (i=1; i<n; i++)

Begin

new (t)

t.left_child=least(w)

t.right_child=least(w)

t.weight=t.left_child.weight+t.right_child.weight

Insert (w, t)

End

End

Time complexity:

$$O(n \log n)$$

交大資工) Briefly describe Huffman encoding and illustrate it for the following data step by step

A B A D C D A C A B B C B B B A

交大資工) Construct Huffman binary codes for eight messages whose probabilities of appearance are 0.2, 0.19, 0.18, 0.14, 0.11, 0.08, 0.06, and 0.04

清大資工) Consider a new design cpu with six instructions which have the probability of occurrence

$$I_1 = 0.4 \quad I_2 = 0.3 \quad I_3 = 0.01 \quad I_4 = 0.08 \quad I_5 = 0.06 \quad I_6 = 0.06$$

- (1) Construct the opcodes using Huffman's method
- (2) Calculate the average number of bits per instruction