第六章

樹狀結構

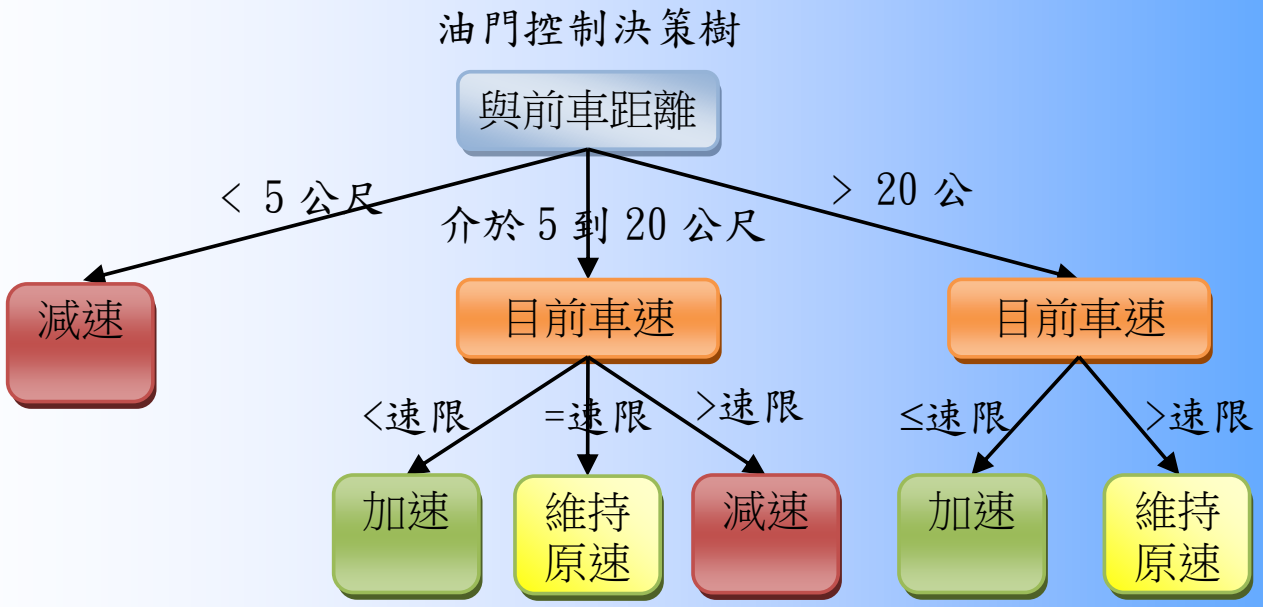
Tree

版權屬作者所有，非經作者
同意不得用於教學以外用途

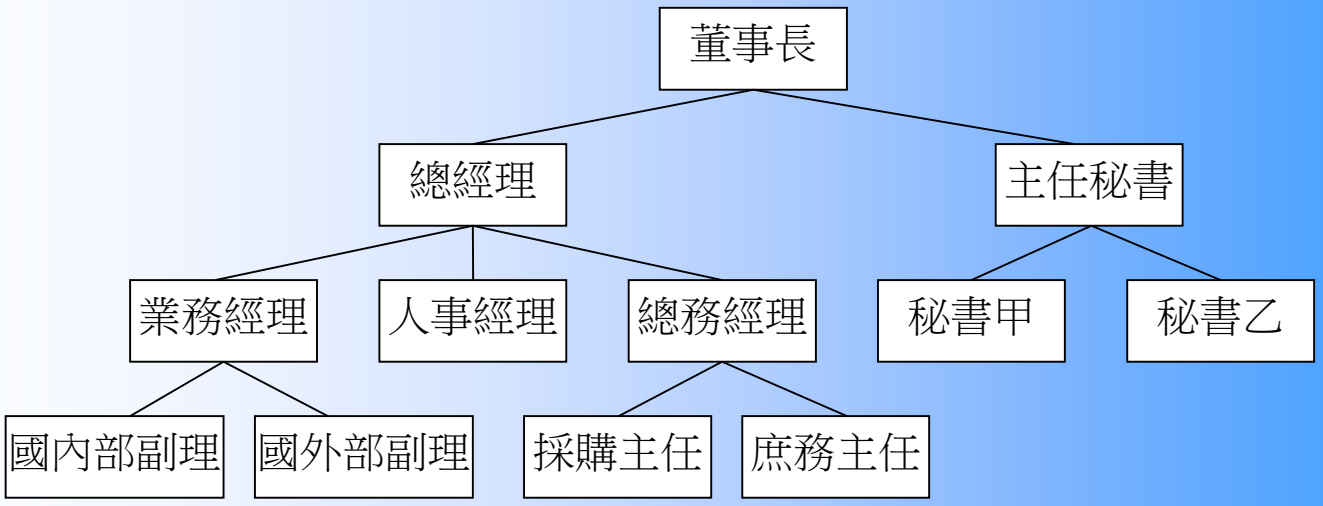
本章內容

人工智慧 AI 是現今資訊科技應用的主流，無人車將要開始在街上趴趴走。無人車上的軟體設計者是如何與周圍的交通環境互動，讓車子不管走到哪裡，遇見什麼狀況都能做出正確的反應呢？

答案是使用「樹狀結構」的資料結構和演算法。例如車子在行進 中，要判斷是加速、減速、或是維持原速，可以使用「決策樹」來建立邏輯。



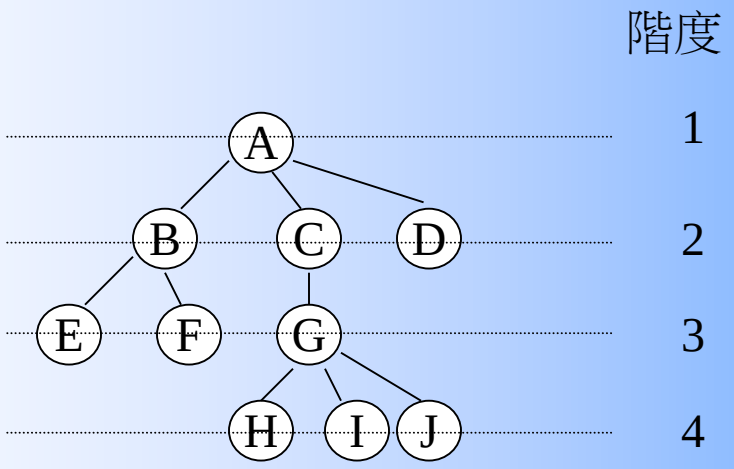
6-1 樹及定義



- 第五章提到，樹在數學上的定義是：沒有環路的連通圖，我們稱這種樹為「一般樹」(general tree) 或是「無根樹」(unrooted tree)

- 在資料結構中所應用的樹，我們稱之為「有根樹」(rooted tree)，有一個特定的節點稱為「樹根」(root)。樹根通常畫在一棵樹的最上方。樹根又包含零個到多個「子樹」(subtree)，畫在樹根的下方。

- 例如圖6.1中，董事長即為樹根。此樹根包含兩個子樹，分別是以總經理為樹根的子樹，和以主任秘書為樹根的另一子樹。



1. 內部節點 (internal node ，或非終端節點 ， non-terminal node) ：
   節點 A、B、C、G

2. 外部節點 (external node ，或終端節點 terminal node ，或樹葉 leaf node) ：
   節點 D、E、F、H、I、J

3. 節點 A 是節點 B、C、D 的父節點。節點 B、C、D 是節點 A 的子節點

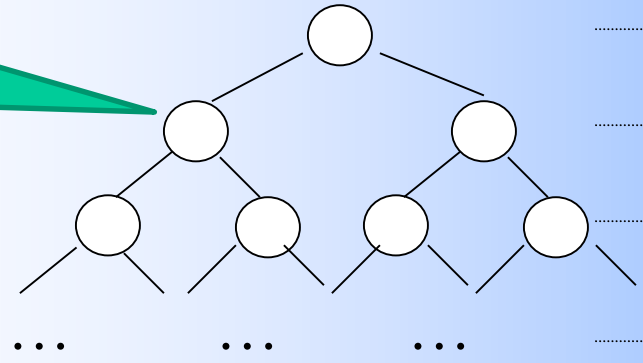
4. 在第 2 階的節點有 B、C、D

5. 這棵樹的高度是 4

6. 所有的樹，**節點數 (V) = 邊數 (E) + 1**
除了樹根以外，其餘每個節點都有一個對應的邊從它的父節點指向它自己，因此節點數會比邊數多1，而多出來的節點就是樹根

6-2 二元樹的基本性質

二元樹：每個內部節點最多有2個兒子

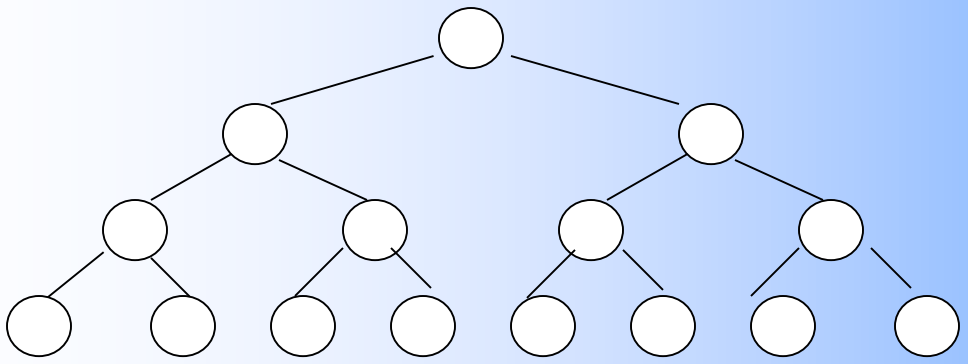


階度 (i)	2^{i-1}
1	1 (2^{1-1})
2	2 (2^{2-1})
3	4 (2^{3-1})
4	8 (2^{4-1})

1. 二元樹**第 i 階**上所有節點的數目最多為 2^{i-1} 個

2. 高度為 k 的二元樹上所有節點的數目最多為 2^k -1個

3. **完滿(full)二元樹**：高度為 k 且總節點數為 2^k -1的二元樹

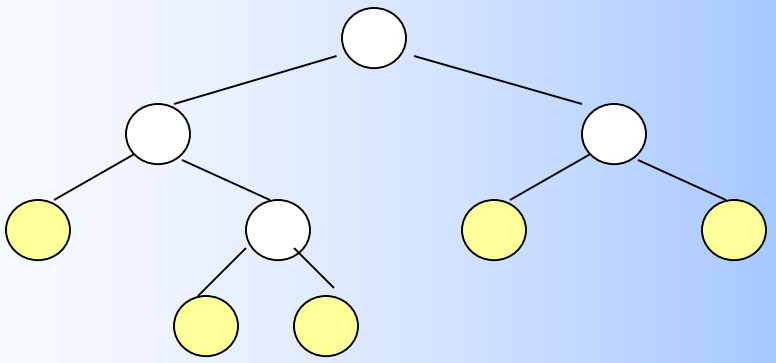


高度為 4 的完滿二元樹

總節點數為 $2^4 - 1 = 15$

4. 高度為 k 的二元樹上所有節點的數目最少為 k 個

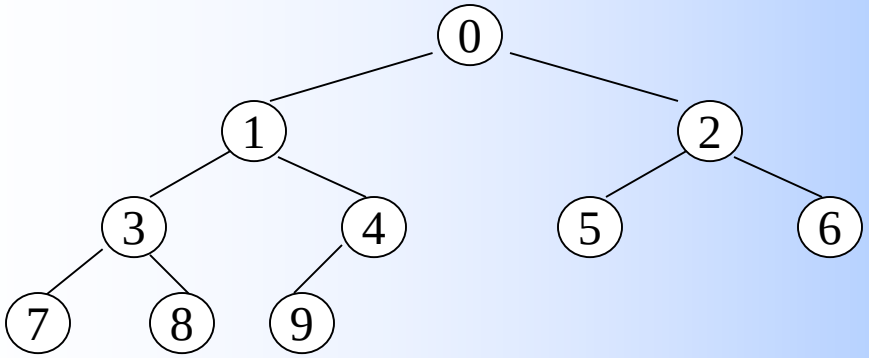
5.若是所有內部節點都正好有兩個子節點，這種二元樹稱為**正規二元樹**
(formal binary tree)



外部節點數目 t 比內部節點數目 i 多一

$$t = i + 1$$

6.如果二元樹的排列情形如下：第 k 階滿了才能排到第 k+1 階，並且每一階的節點都是往左靠 ，這種二元樹稱為**完整二元樹**
(complete binary tree)

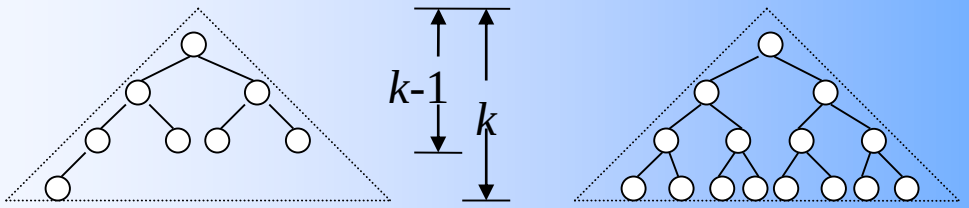


※ 編號 i 的左兒子編號為 2i + 1，
右兒子編號為 2i + 2

※編號 i 的父節點編號為 (i-1)/2

7. 具有n個節點的完整二元樹，其高度為$\lfloor \lg n \rfloor + 1$。

假設完整二元樹的節點數目為 n，高度為 k。
根據二元樹性質(第2點及第3點)，高度為 k 的完整二元樹，節點最多的狀況就是高度為 k 的完滿二元樹，總節點數是2^k-1，節點最少的狀況就是高度為 k-1的完滿二元樹再加一個節點，節點數是$2^{k-1}-1+1=2^{k-1}$。



節點最少　　節點最多

因此：$0 < 2^{k-1} \le n \le 2^{k-1}$

$0 < 2^{k-1} \le n \le 2^{k-1} < 2^k$

$0 < 2^{k-1} \le n < 2^k$

取對數 $\lg(2^{k-1}) \le \lg n < \lg(2^k)$

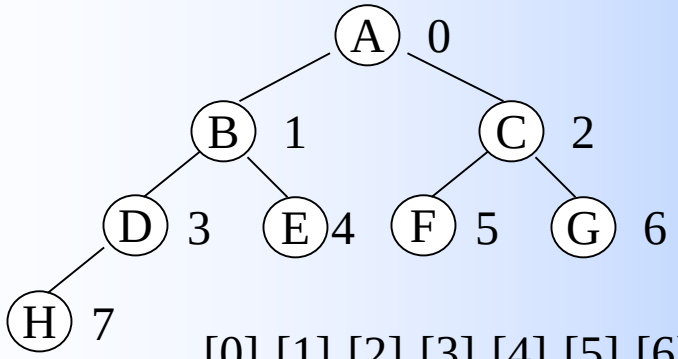
$k-1 \le \lg n < k$

$\lfloor \lg n \rfloor = k-1$

$k = \lfloor \lg n \rfloor + 1$

6-3 二元樹的儲存方式

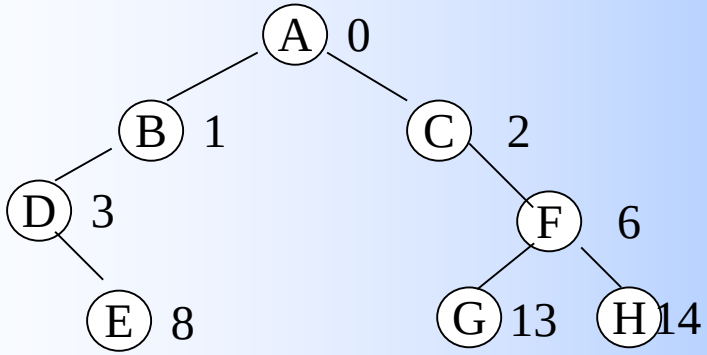
一維陣列表示法



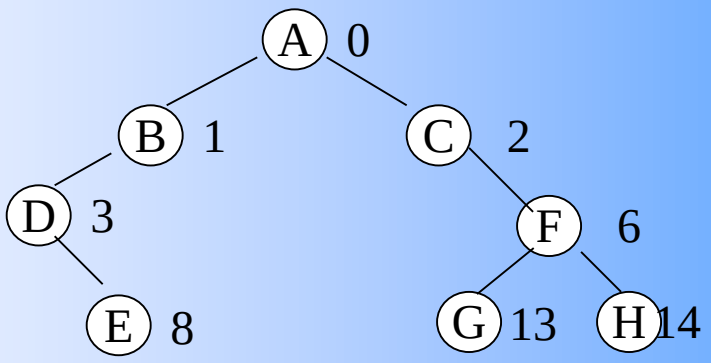
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Tree1[]	A	B	C	D	E	F	G	H

1. 使用完整二元樹的編號方式。

2. 如果二元樹不很「完整」，將會有很多儲存空間的浪費。



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Tree2[]	A	B	C	D			F		E					G	H

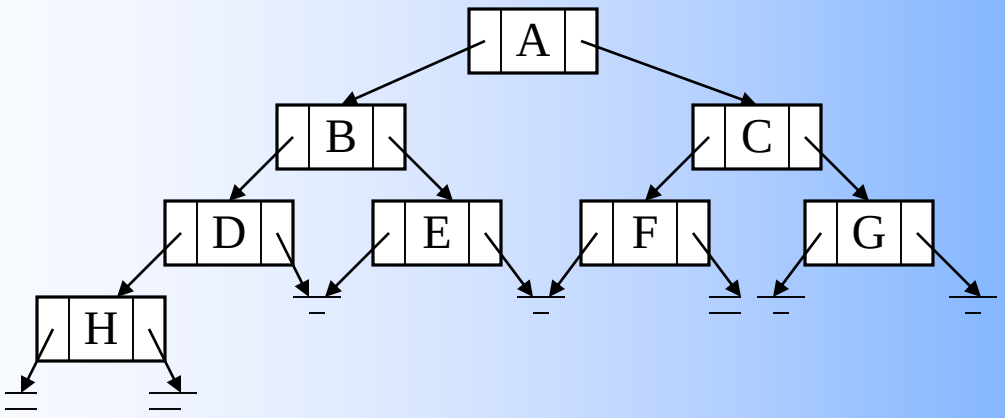


	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Tree2[]	A	B	C	D			F		E					G	H

如果所有的節點資料都往陣列的前端靠緊，例如圖 6.8(b)把 F、E、G、H 搬到 Tree2[4]—Tree2[7]，不就可以節省空間的浪費了嗎？這個說法錯在哪裡？

鏈結表示法—動態配置節點

```
typedef struct tagTnode
        {
            struct tagTnode *left_c;       // 指向左兒子
            char    data;                  //資料
            struct tagTnode *right_c;      //指向右兒子
        }TNode;
```



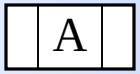
6-4 二元樹的建立

建立二元樹的步驟

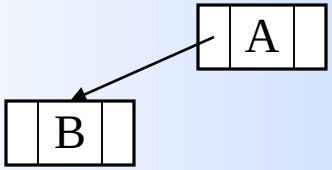
1. 讀入資料，如果是 0 則表示停止
2. 否則以此資料建立一個節點，並且以同樣原則建立此節點的左子樹，直到不能繼續為止。
3. 接著建立此節點的右子樹，直到不能繼續為止。

輸入資料為：ABDH000E00CF00G00，則建立此二元樹的步驟如下：

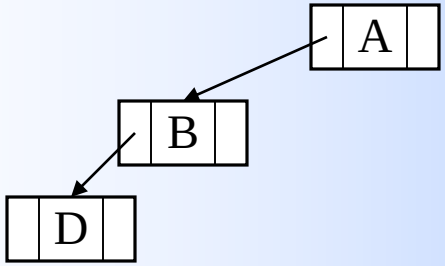
1. 讀入資料A，以此資料建立節點，接著建立此節點的左子樹



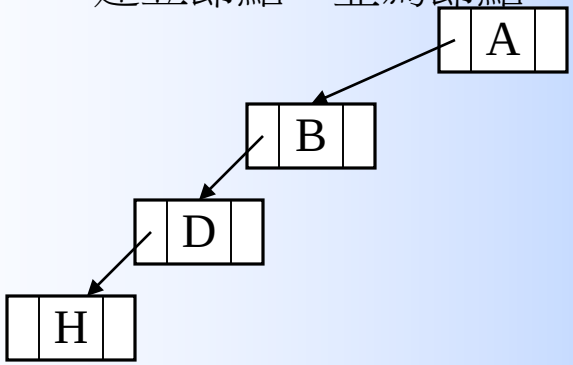
2.讀入資料 B，建立節點，並為節點 A 的左子節點，接著建立此節點 B 的左子樹



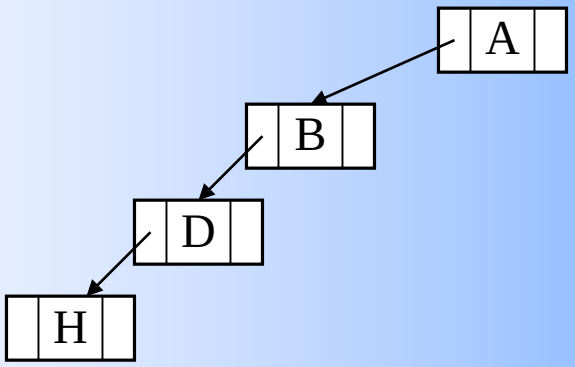
3.讀入資料 D，建立節點，並為節點 B 的左子節點，接著建立此節點 D 的左子樹



4.讀入資料 H，建立節點，並為節點 D 的左子節點，接著建立此節點 H 的左子樹



ABDH000E00CF00G00

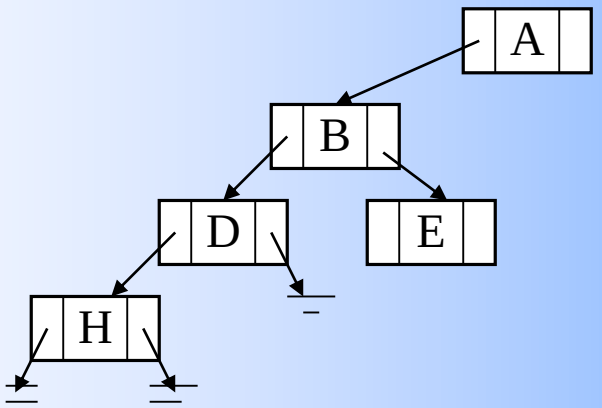


5. 讀入0，表示 H 無左子節點。節點H的左子樹已無法再長，接著建立節點H的右子樹

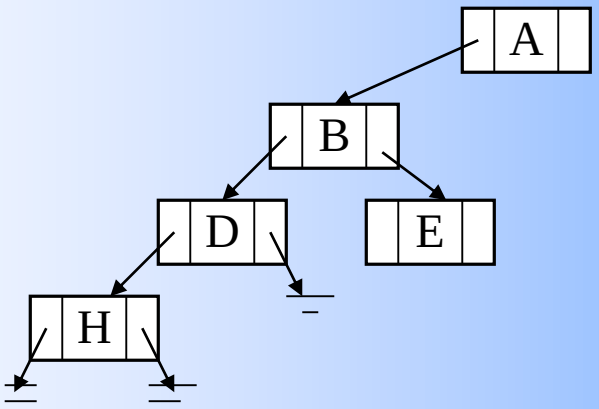
6. 讀入0，表示 H 無右子節點。節點D的左子樹已無法再長，接著建立節點D的右子樹

7. 讀入0，表示 D 無右子節點。此時節點 B 的左子樹已無法繼續再長

8. 讀入資料 E，以此資料建立節點，並為節點 B 的右子節點



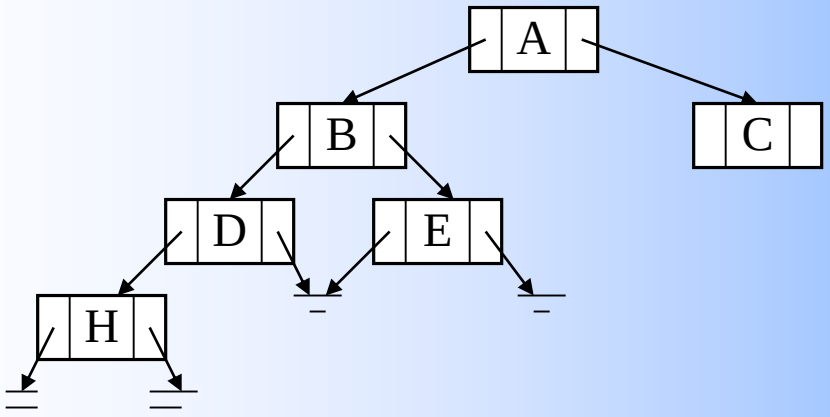
ABDH000E00CF00G00



9. 讀入0，表示 E 無左子節點

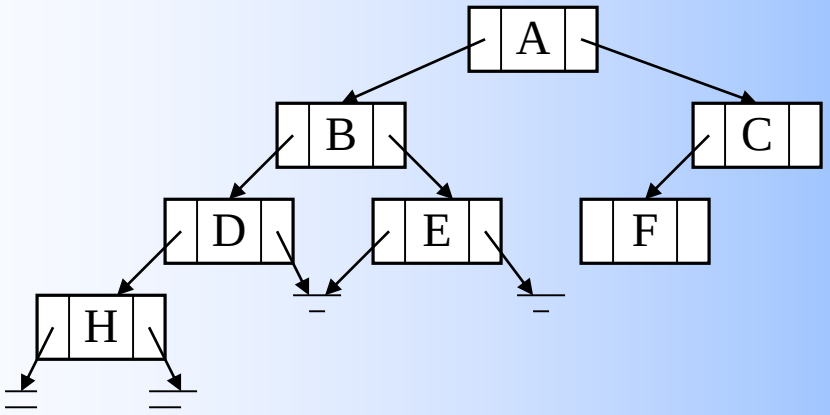
10. 讀入0，表示 E 無右子節點。節點A的左子樹已無法再長

11. 讀入資料 C，以此資料建立節點，並為節點 A 的右子節點

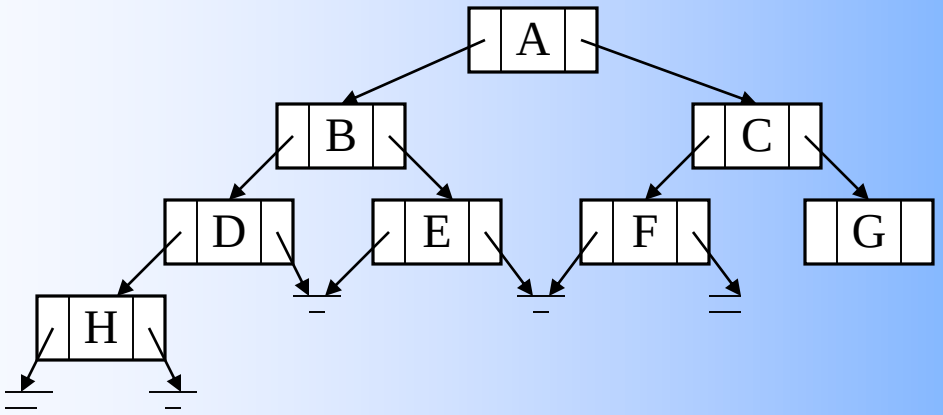


ABDH000E00CF00G00

12. 讀入資料 F，以此資料建立節點，並為節點 C 的左子節點



13. 讀入0，表示 F 無左子節點
14. 讀入0，表示 F 無右子節點。節點 C 的左子樹已無法再長
15. 讀入資料 G，以此資料建立節點，並為節點 C 的右子節點



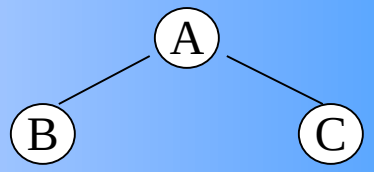
16. 讀入0，表示 G 無左子節點
17. 讀入0，表示 G 無右子節點。節點 A 的右子樹已無法再長。由於節點 A 是樹根，因此整棵樹也無法繼續再長，建立過程結束

6-5 二元樹的走訪

系統性走訪：「前序走訪」、「中序走訪」、和「後序走訪」

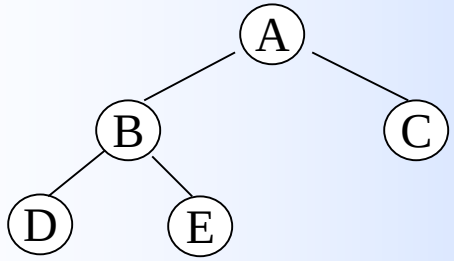
前序走訪 (preorder traverse)：

- 先拜訪目前的節點
- 再以前序順序拜訪左子樹
- 再以前序順序拜訪右子樹



前序走訪順序

A B C

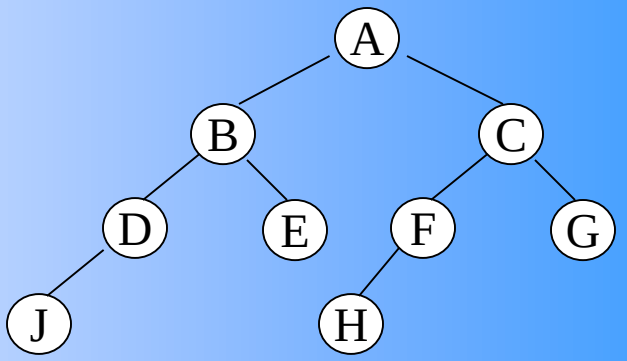


前序走訪順序

A B D E C

A的左子樹

A的右子樹

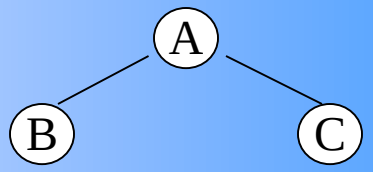


前序走訪順序

A B D J E C F H G

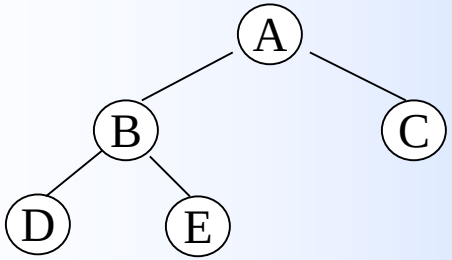
中序走訪 (inorder traverse)

- 先以中序順序拜訪左子樹
- 再拜訪目前的節點
- 再以中序順序拜訪右子樹



中序走訪順序

B A C

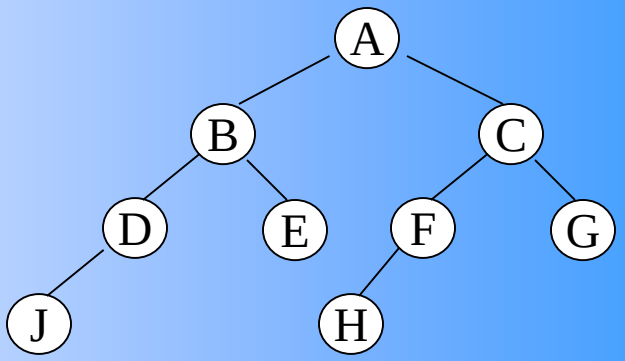


中序走訪順序

D B E A C

A的左子樹

A的右子樹

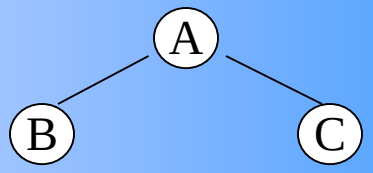


中序走訪順序

J D B E A H F C G

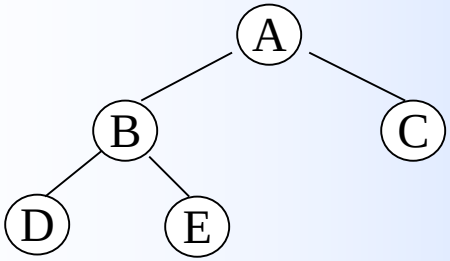
後序走訪 (postorder traverse)：

- 先以後序順序拜訪左子樹
- 再以後序順序拜訪右子樹
- 再拜訪目前的節點



後序走訪順序

B C A

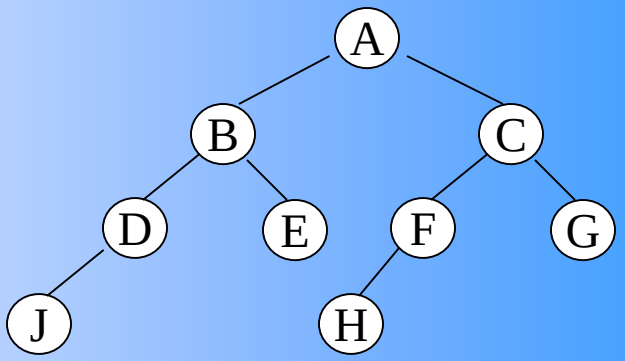


後序走訪順序

D E B C A

A的左子樹

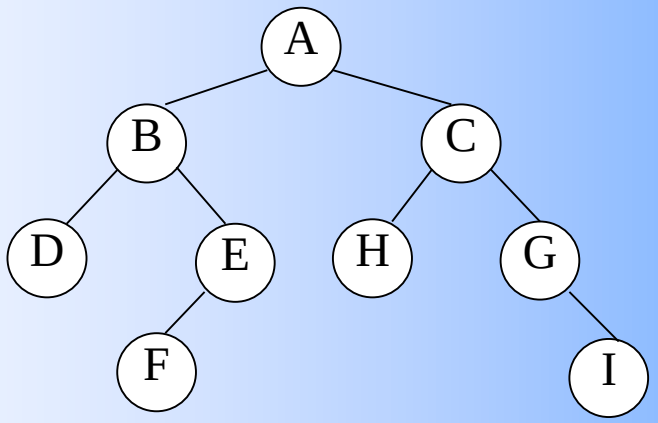
A的右子樹



後序走訪順序

J D E B H F G C A

例 6.1



前序走訪的順序為：ABDEFCHGI

中序走訪的順序為：DBFEAHCGI

後序走訪的順序為：DFEBHIGCA

前序走訪函式

```
1.  void preorder(NODE *p)
2.  {       if (p != NULL)
3.          {       visit(p);                   //拜訪目前的節點
4.                  preorder(p->left_c);        //遞迴拜訪左子樹
5.                  preorder(p->right_c);       //遞迴拜訪右子樹
6.          }
7.  }
```

中序走訪函式

```
1.  void inorder(NODE *p)
2.  {       if (p != NULL)
3.          {       inorder(p->left_c);         //遞迴拜訪左子樹
4.                  visit(p);                   //拜訪目前的節點
5.                  inorder(p->right_c);        //遞迴拜訪右子樹
6.          }
7.  }
```

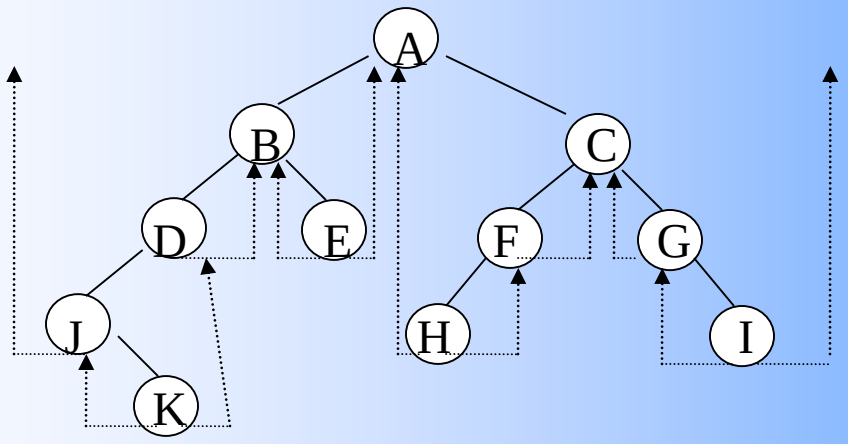
後序走訪函式

```
1.  void postorder(NODE *p)
2.  {       if (p != NULL)
3.          {       postorder(p->left_c);       //遞迴拜訪左子樹
4.                  postorder(p->right_c);      //遞迴拜訪右子樹
5.                  visit(p);                   //拜訪目前的節點
6.          }
7.  }
```

6-6 引線二元樹

※ n 個節點的二元樹，有 $2n-(n\text{-}1)=n+1$ 個指標是接地的

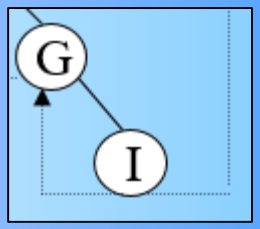
※ 將原本接地的指標作為**引線**，將左引線指向此節點的中序立即前行者 ，將右引線指向此節點的中序立即後繼者，稱為**引線二元樹**

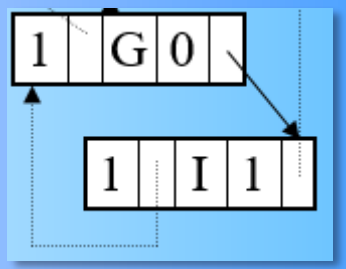


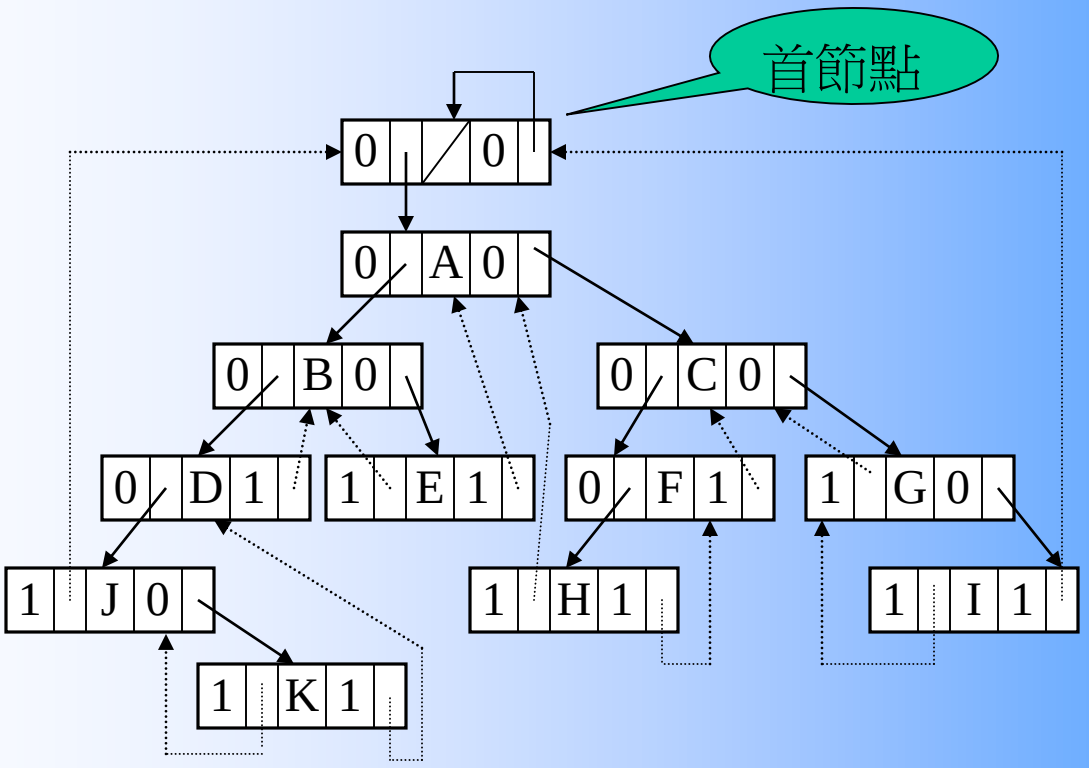
如何找節點的中序立即後繼者 ：

1. 如果是右引線，**照右引線所指即為後繼者**。例如節點K，節點K有右引線，因此照著它的右引線指向節點D，節點D即為節點K的後繼者。
2. 如果是右指標，則先往右，再一路沿著左指標往左，一直到沒有左指標，而是左引線的節點為止。例如節點A，節點A有右指標，則往右到節點C，再一路沿著左指標到達節點H，節點H沒有左指標，而是左引線，因此節點H即為節點A的後繼者。

```
#define THREAD 1
#define POINTER 0
typedef  struct  tagTh_Node{
   char  left_thread;
   struct  tagTh_Node  *left_c;
   char  data;
   struct  tagTh_Node  *right_c;
   char  right_thread;
} Th_Node;
```







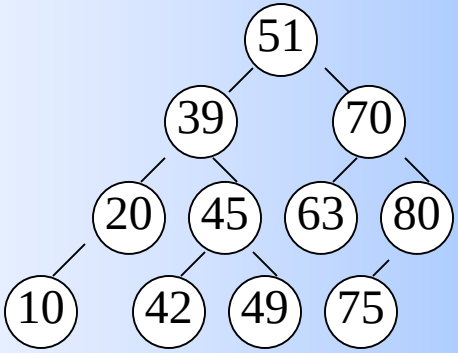
對整棵樹作中序走訪的過程：

1. 首節點的後繼者，先往右 (回到自己)，再一路往左，可找到節點 J
2. 節點 J 的後繼者，先往右(K)，再一路往左 (沒有)，因此節點 J 的後繼者是節點 K
3. 節點 K 的後繼者，往右是引線，因此節點 K 的後繼者是節點 D
4. ~ 11. 以此類推，順序為：B E A H F C G I
12. 節點 I 的後繼者，往右是引線，因此節點 I 的後繼者是首節點，走訪結束

6-7 二元搜尋樹

二元搜尋樹的定義：

對二元搜尋樹中每一個內部節點而言，它的左子樹所有節點的資料值，都小於它的資料值，它的右子樹所有節點的資料值，都大於它的資料值



※ 這是一棵二元搜尋樹，因為每個內部節點都符合定義

※ 對二元搜尋樹作中序走訪，可以得到由小到大排列的順序

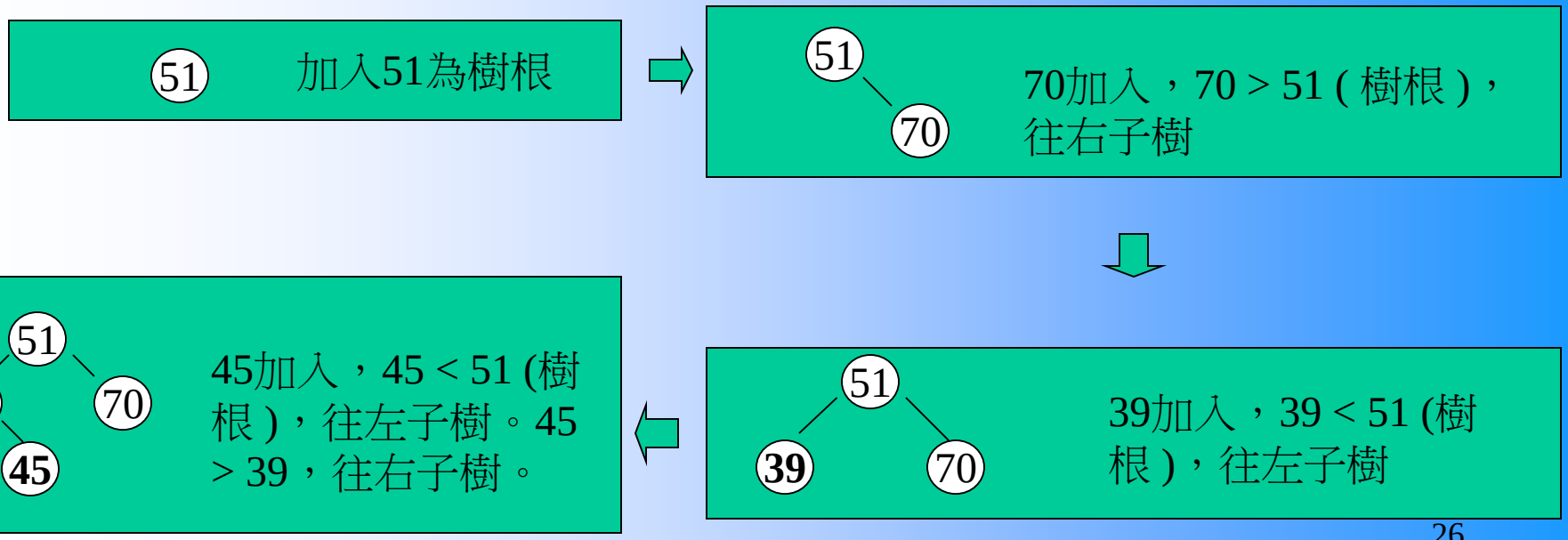
※ 將資料整理成二元搜尋樹，將可以有效率地找到所要的資料

二元搜尋樹的建立

每讀入一個資料就依循下列原則，將新節點插入擴充中的二元搜尋樹：

1. 如果是空樹，則新節點為樹根
2. 否則將新節點的資料與樹根相比，小於樹根則往左子樹，大於樹根則往右子樹
3. 重複與此子樹的樹根比較，直到指標接地為止
4. 將新節點加在最後停留之處

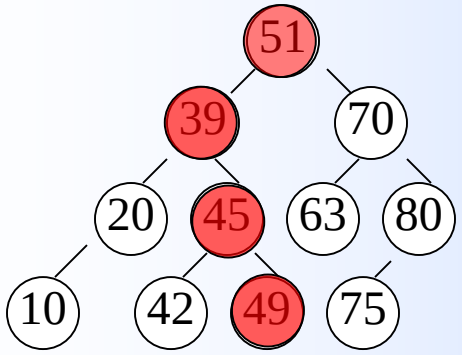
我們以下列的資料，示範建立二元搜尋樹的程序： 51、70、39、45



二元搜尋樹的節點搜尋

搜尋的過程和插入節點十分類似，依循下列原則：

1. 將搜尋的鍵值 (key) 與樹根相比，若相等則搜尋成功
2. 否則，小於樹根則往左子樹，大於樹根則往右子樹
3. 重複與此子樹的樹根比較，若相等則搜尋成功
4. 若直到指標接地為止都不相等，則搜尋失敗



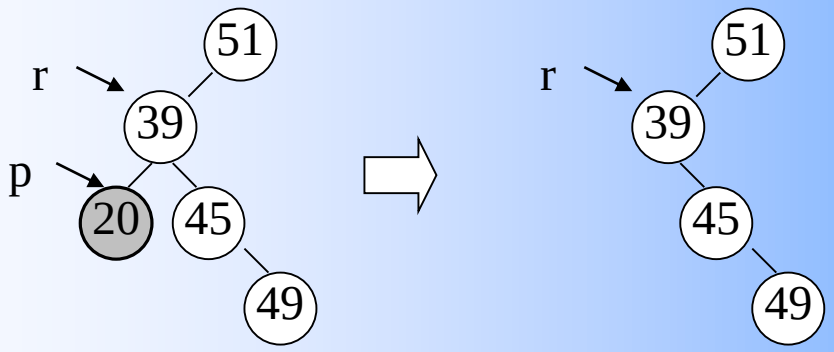
若搜尋的鍵值為 49

1. 首先鍵值49和樹根資料51相比。49比51小，因此往左子樹，因為根據定義，如果資料49在二元搜尋樹中，只可能在資料51的左子樹中出現
2. 鍵值49和資料39相比，49比39大，往右子樹。
3. 鍵值49和資料45相比，49比45大，往右子樹
4. 鍵值49和資料49相比，相等，搜尋成功

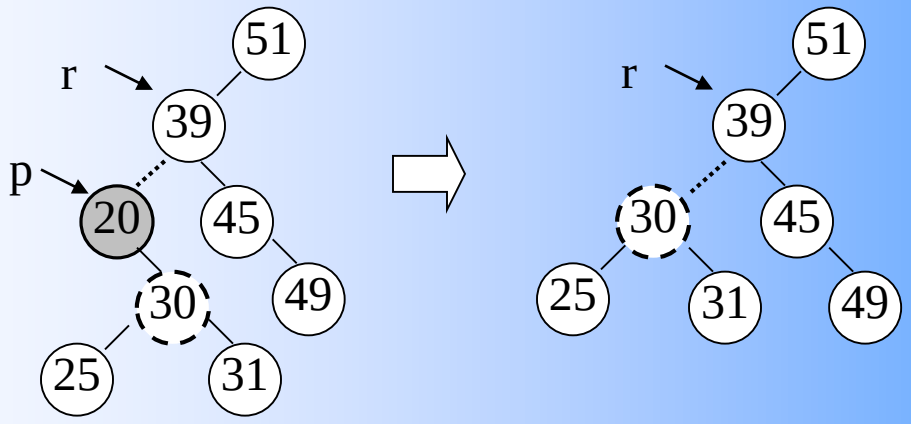
二元搜尋樹的節點刪除

依照要刪除節點所在的位置，分為三種狀況來討論：

狀況1. 要刪除的節點是**樹葉**：只要直接刪去即可，這是最單純的狀況。



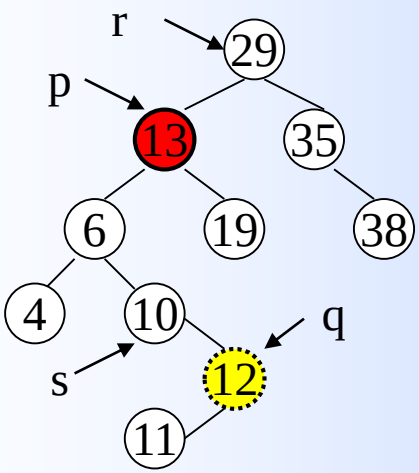
狀況2. 要刪除的節點**只有一棵子樹**（左或右皆同）：只要將其父節點的指標越過欲刪除節點，改指向此節點的子節點即可。



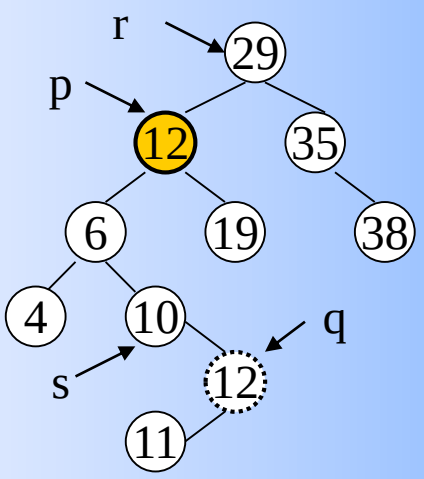
狀況 3. 要刪除的節點**有二棵子樹**，作法是：

(1) 找到此節點的「中序立即前行者」，並且以此前行者節點的資料，代替要刪除節點的資料。

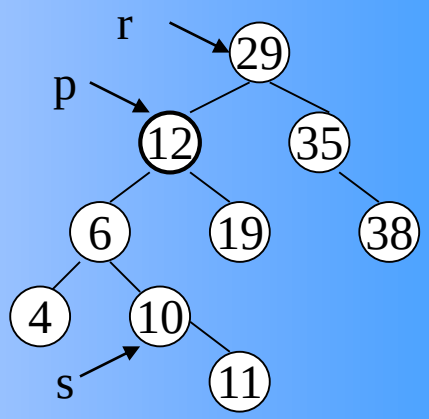
(2) 由於此前行者最多只可能有一棵子樹（否則它就不可能被找上），因此再按照前述狀況1或狀況2，直接刪除此前行者即可完成刪除動作。



(a) 欲刪除 13



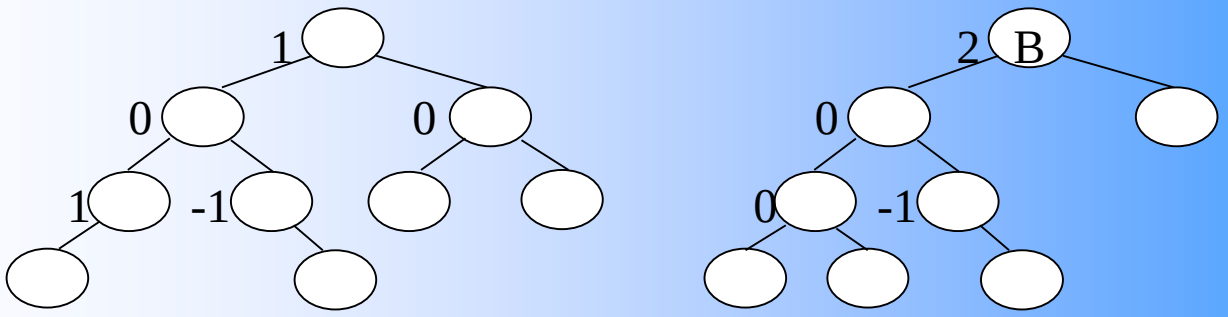
(b) 由12代替其位



(c)刪除原來的 12

6-8 高度平衡二元樹（AVL樹）

在AVL樹上，所有內部節點的左子樹高度和右子樹高度，相差小於或等於1。

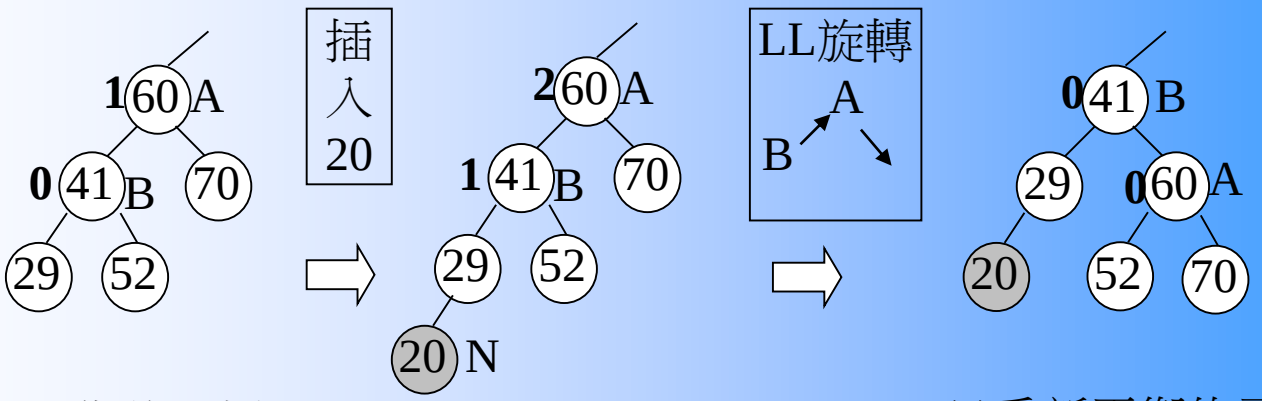


(a)AVL樹　　(b)非AVL樹

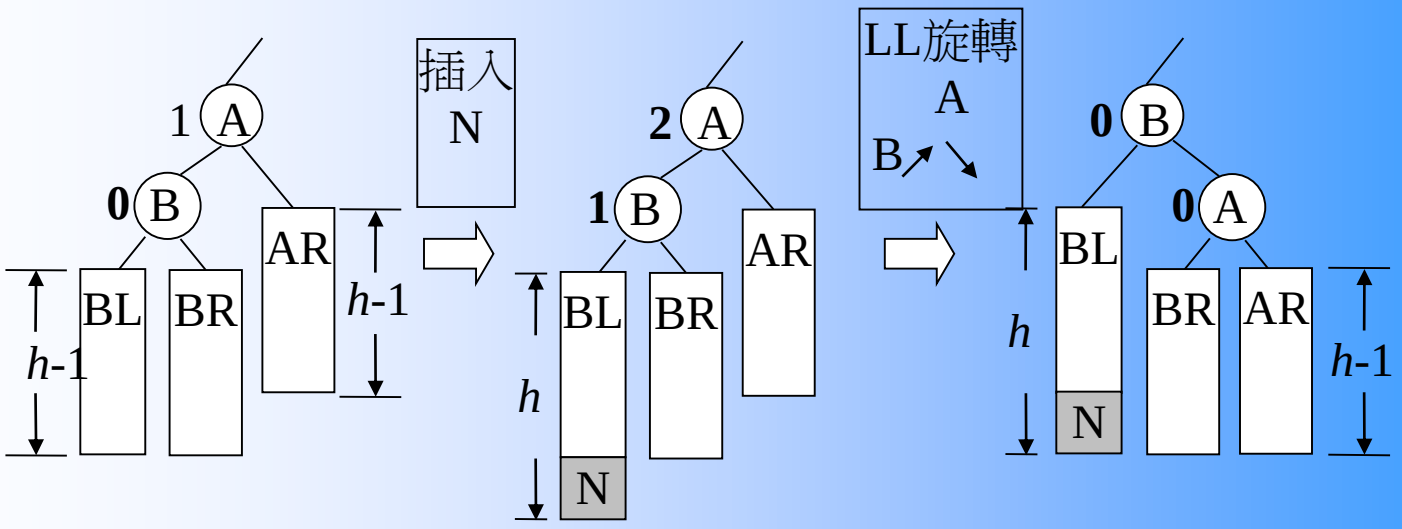
假設新加入節點為N，而由於新節點的加入，造成原本平衡係數BF為±1的節點其BF變為 ±2，違反了AVL性質。這時，就必須根據新節點插入的位置，分成四種狀況來**旋轉**調整，使其恢復AVL性質。

LL型：新節點N插入到節點A的左兒子的左子樹。
RR型：新節點N插入到節點A的右兒子的右子樹。
LR型：新節點N插入到節點A的左兒子的右子樹。
RL型：新節點N插入到節點A的右兒子的左子樹。

LL型

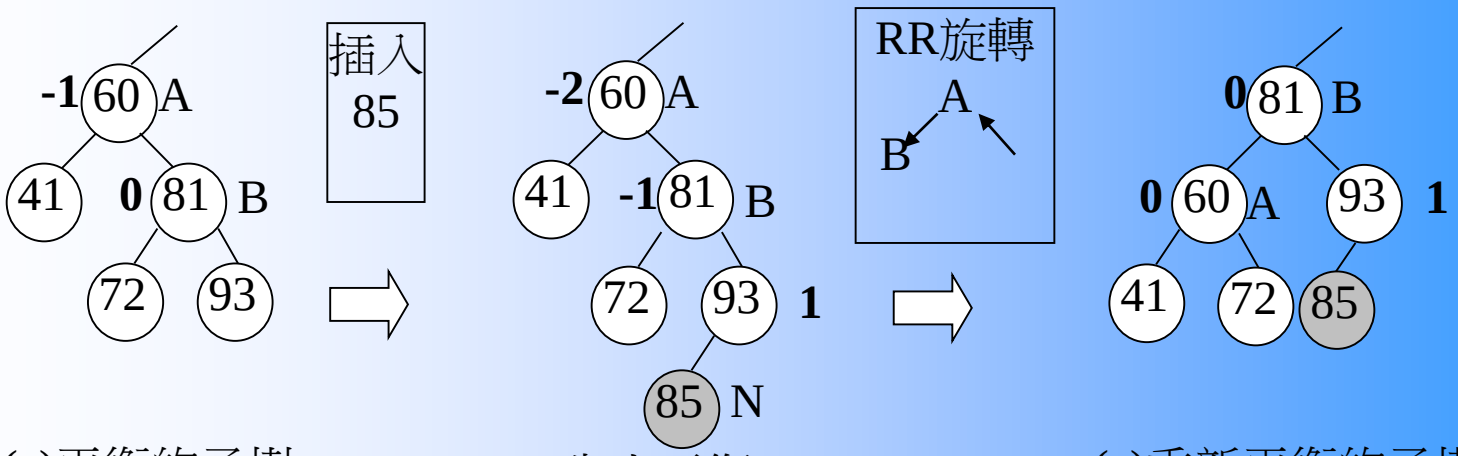


(a)平衡的子樹　(b)失去平衡　(c)重新平衡的子樹

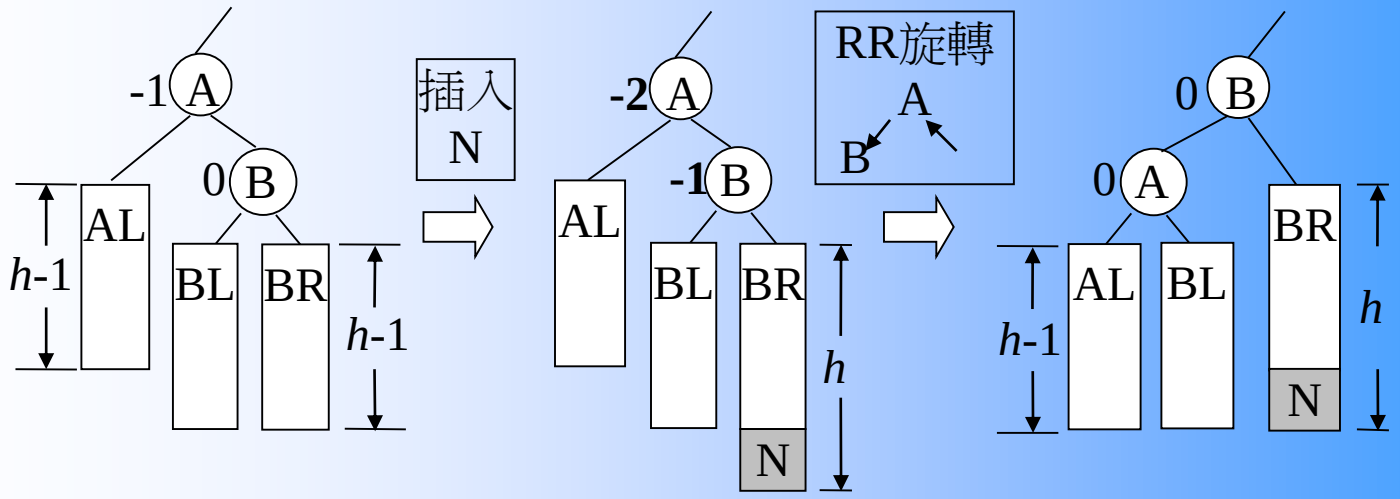


(a)平衡的子樹　(b)失去平衡　(c)重新平衡

RR型

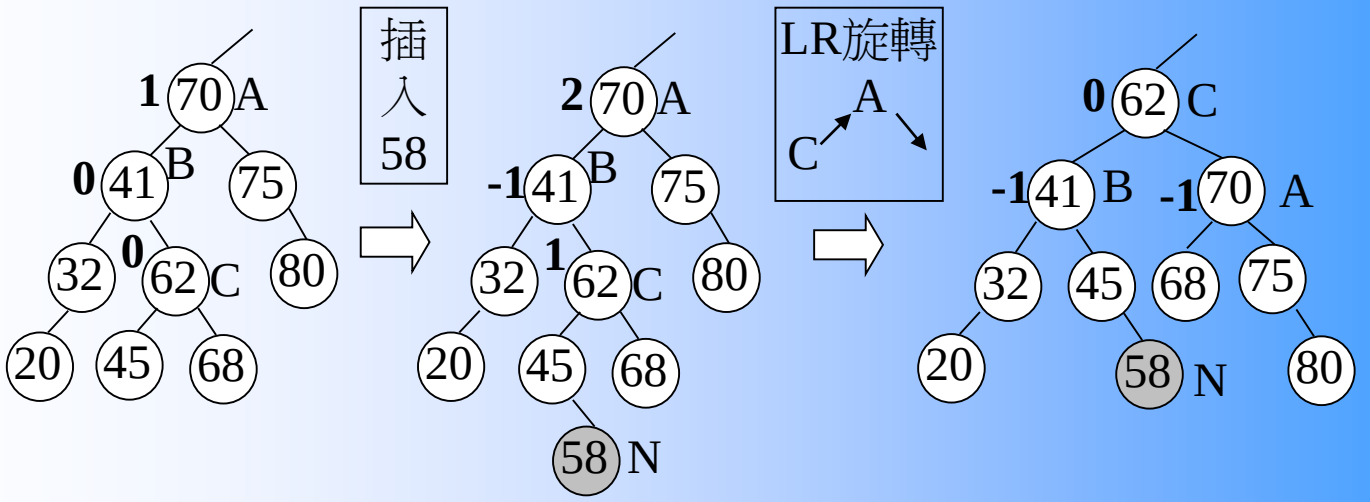


(a)平衡的子樹　(b)失去平衡　(c)重新平衡的子樹

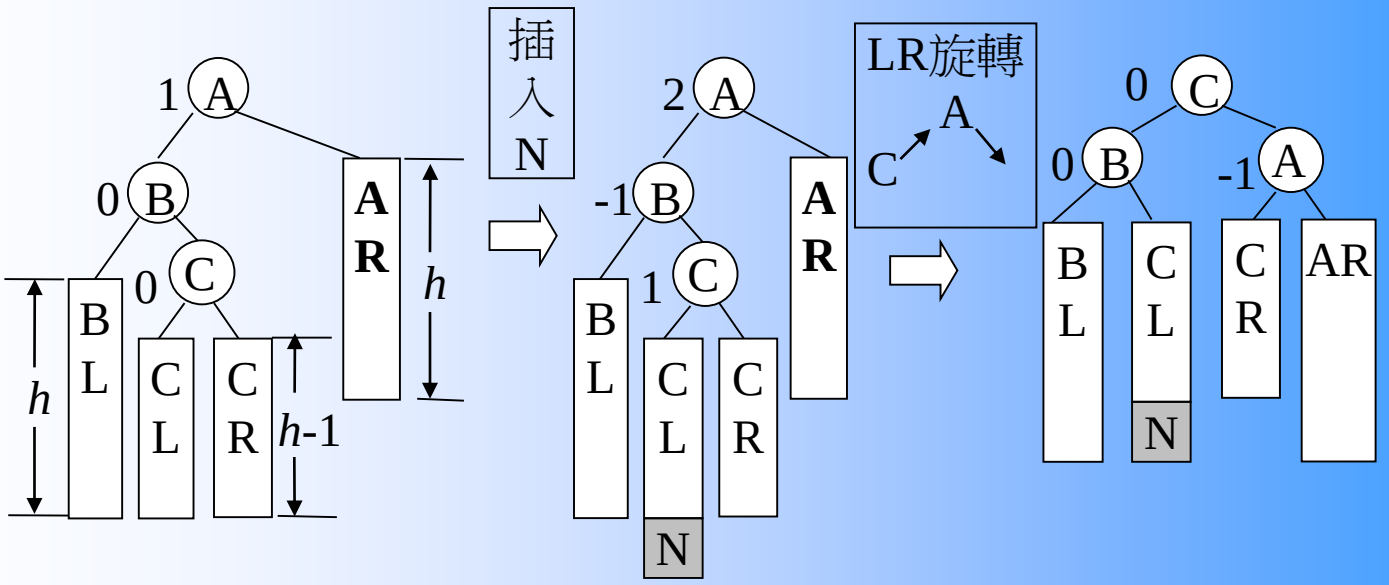


(a)平衡的子樹　(b)失去平衡　(c)重新平衡

LR型

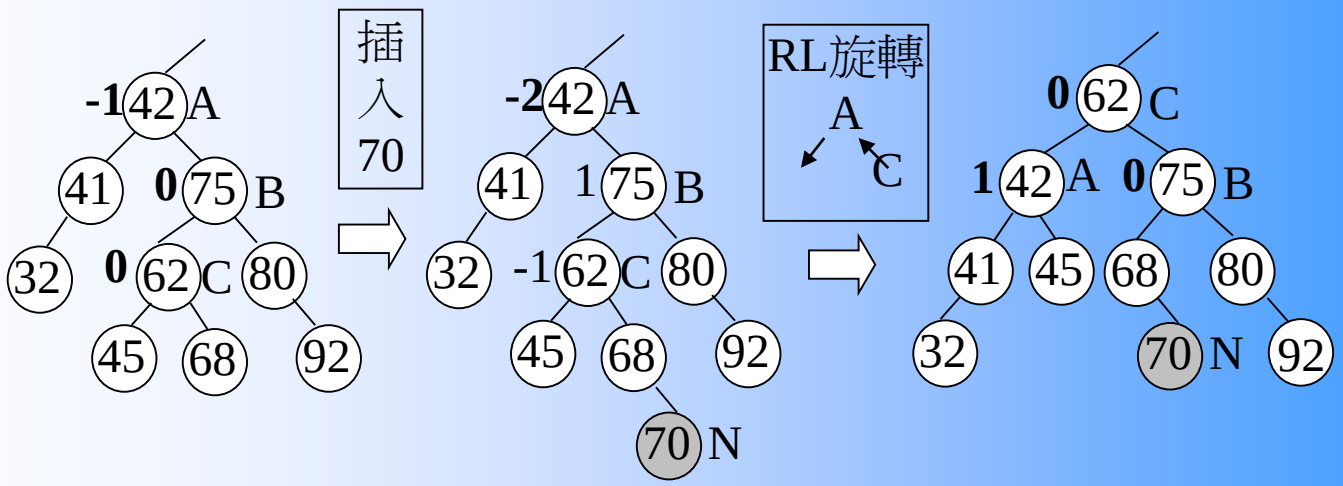


(a)平衡的子樹　(b)失去平衡　(c)重新平衡

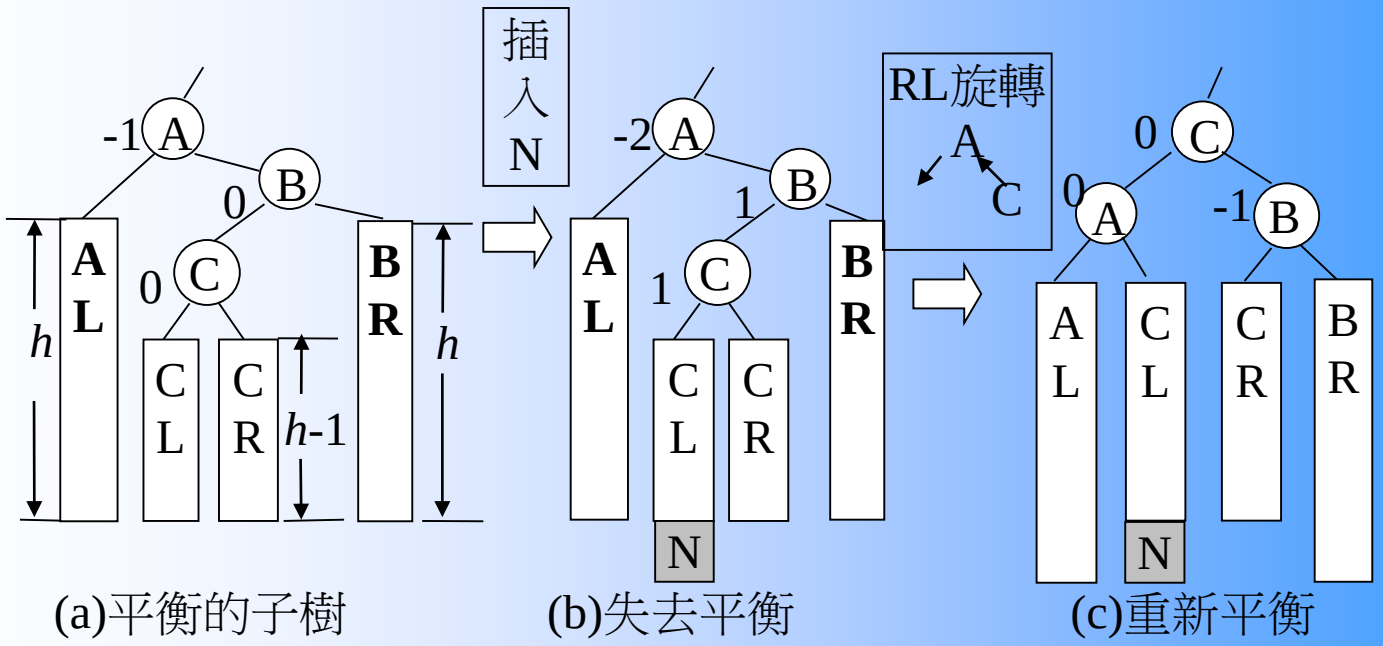


(a)平衡的子樹　(b)失去平衡　(c)重新平衡

RL型



(a)平衡的子樹　(b)失去平衡　(c)重新平衡

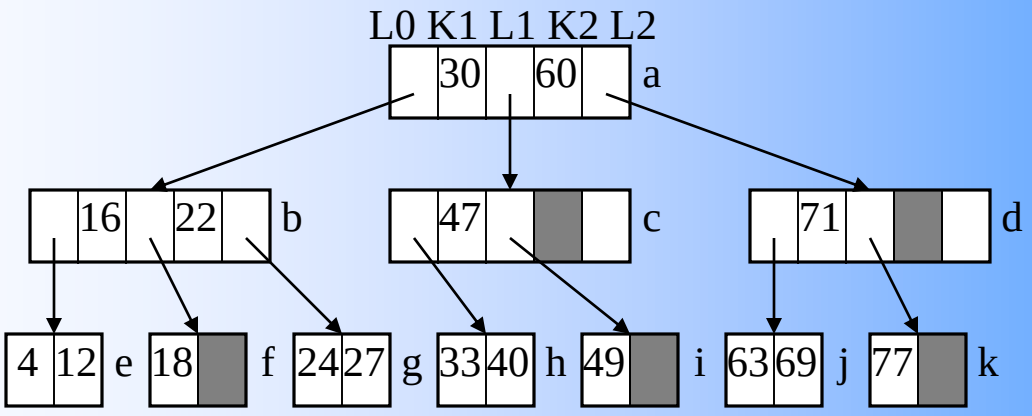


(a)平衡的子樹　(b)失去平衡　(c)重新平衡

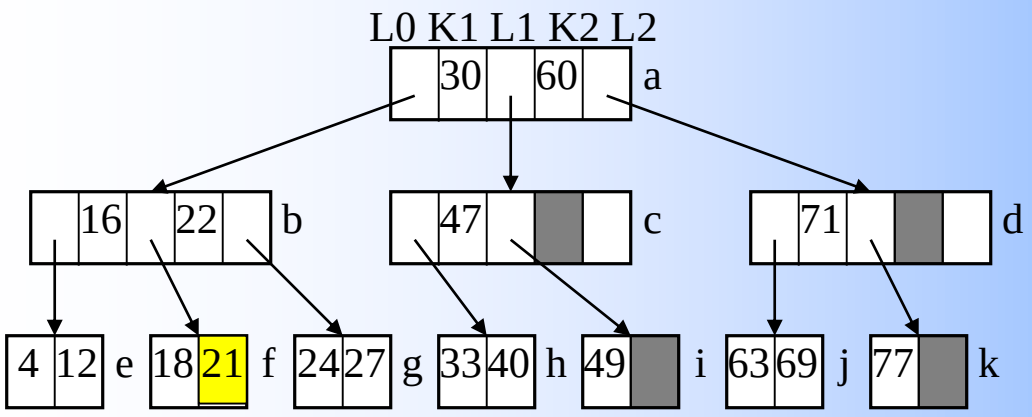
6-9 *m*元搜尋樹及 B 樹

m 階 B 樹符合下列條件：

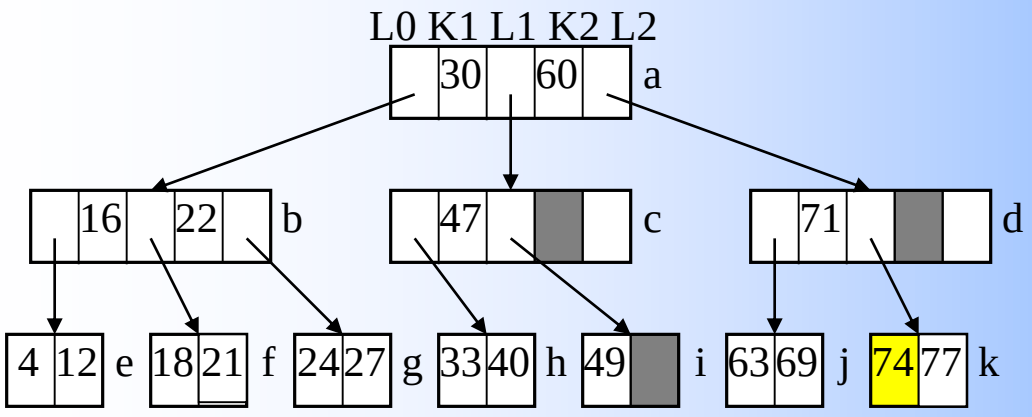
1. 每個節點至多有 *m* 個子樹
2. 樹根至少有 2 個子樹，除非它也是樹葉
3. 除了樹根和樹葉之外，其餘的內部節點，至少有⌈*m* / 2⌉個子樹
4. 所有的樹葉都在同一階，亦即從樹根到任一個樹葉所經過的路徑長度均相同



加入鍵值21



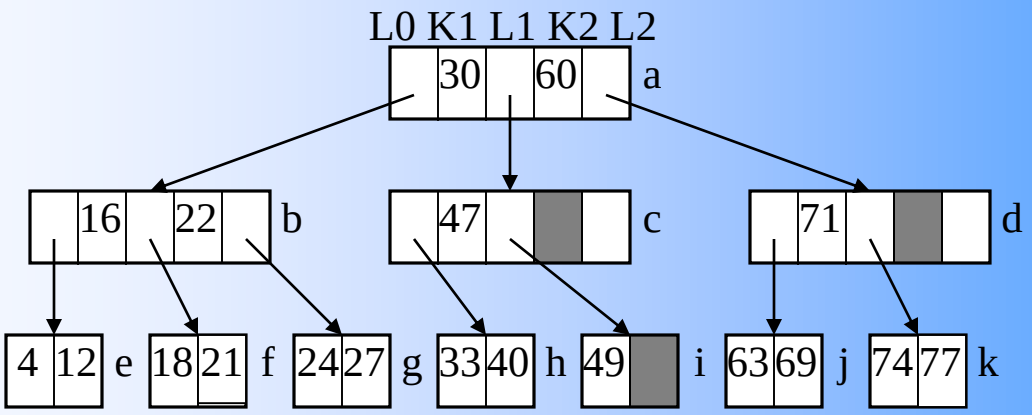
加入鍵值74

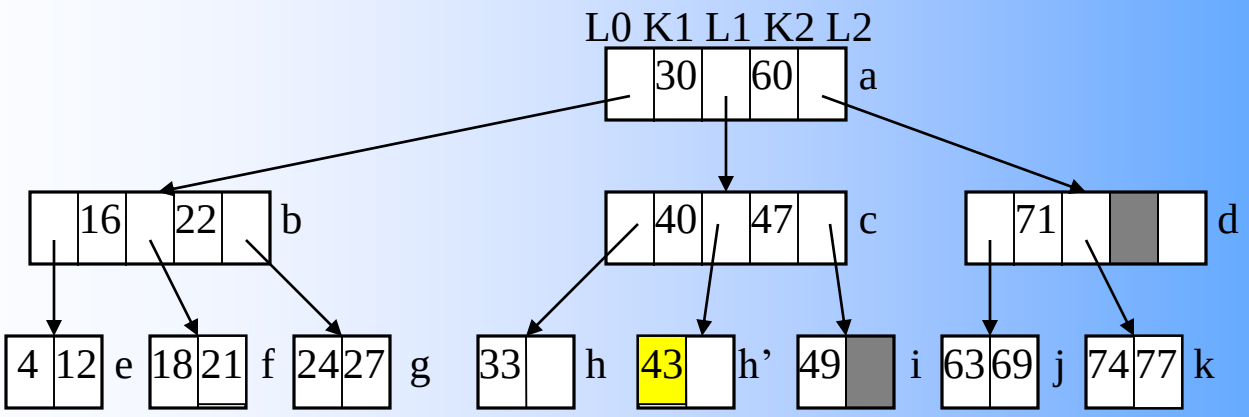


B樹鍵值插入的原則，是必須確保插入後仍符合B樹的特性：

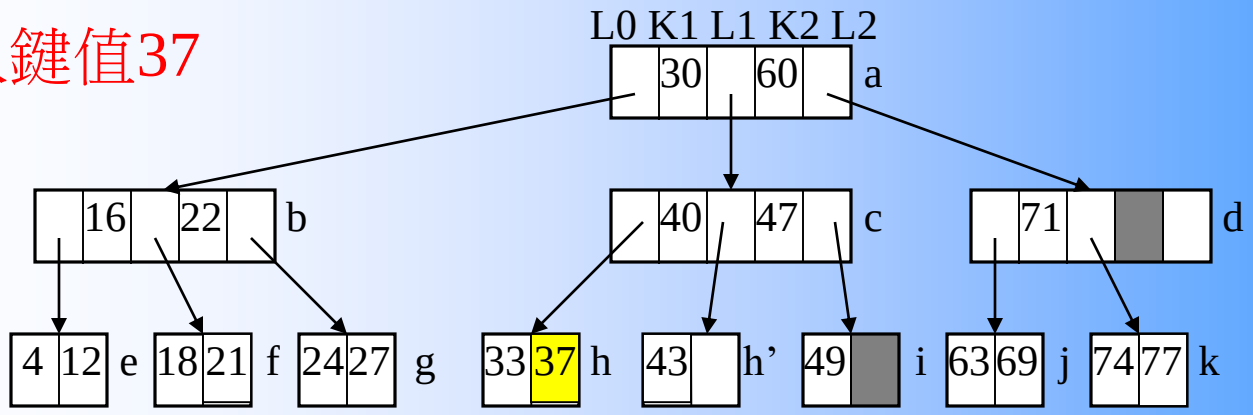
1. 經過類似搜尋的過程，找到第一個可插入點（一定是樹葉）。
2. 如果此樹葉的資料欄位有空位，將新鍵值插入適當順序的資料欄位。
3. 如果樹葉的資料欄位已經額滿沒有空位，則樹葉進行**節點分裂** (split)，並且分配資料鍵值到兩個樹葉，同時將中間的鍵值（第$\lceil m/2 \rceil$個）往上插入父節點。
4. 父節點的插入過程如同步驟2和步驟3，如果此節點還需要分裂，則一直重複。因此每次插入最多可能使B樹的高度h增加1。

加入鍵值43

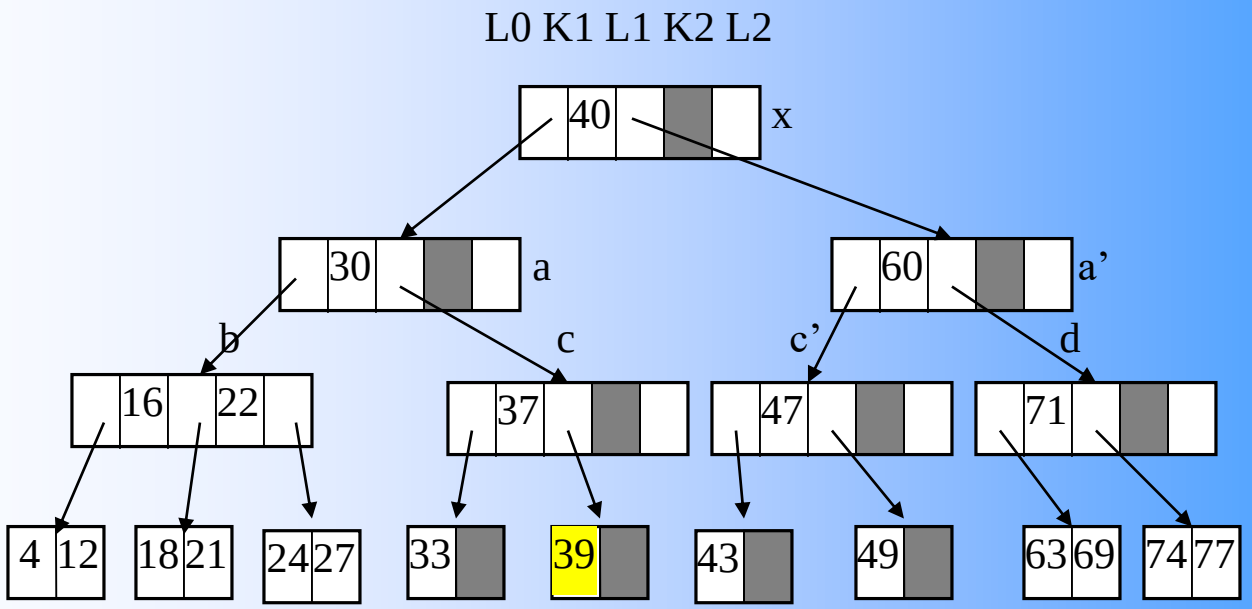

L0 K1 L1 K2 L2
30
60
a
16
22
b
47
c
71
d
4
12
e
18
21
f
24
27
g
33
40
h
49
i
63
69
j
74
77
k


L0 K1 L1 K2 L2
30
60
a
16
22
b
40
47
c
71
d
4
12
e
18
21
f
24
27
g
33
h
43
h'
49
i
63
69
j
74
77
k

加入鍵值37



加入鍵值39

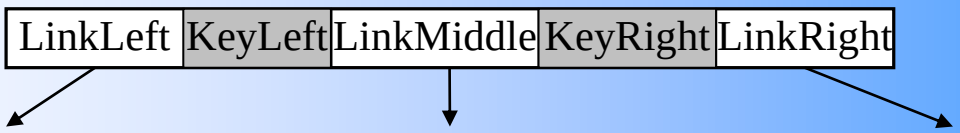


6-10 2-3 樹

- 2 – 3 樹其實就是「3階的B樹」(B tree of order 3)。因此在2 – 3樹上的每個節點最多有2個鍵值、3個指標。
  具有1個鍵值（2個指標）的節點稱為 “2-node”（2-節點）
  具有2個鍵值（3個指標）的節點稱為 ” 3-node”（3-節點）。
- 2 – 3樹的節點結構可以宣告為：

```
1. typedef struct tagTwo3Node
2. {        int          KeyLeft, KeyRight ;
3.          struct       tagTwo3Node   *LinkLeft, *LinkMiddle, *LinkRight;
4. } Two3Node;
```

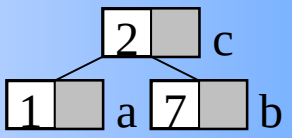
- 2 – 3樹的節點結構可以圖示為：



2 – 3樹的節點鍵值插入，原則如下（與B樹同）：
1. 新鍵值的第一個可插入點一定是樹葉。
2. 如果樹葉的資料欄位有空位，將鍵值插入適當的資料欄位即告完成。
3. 如果樹葉的資料欄位已經額滿沒有空位，則樹葉進行節點分裂 (split) 。
4. 每次插入最多可能使2 – 3樹的高度*h*增加1。

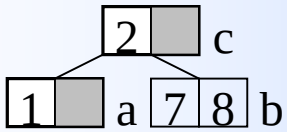
(1)加入2, 7

(2)加入1

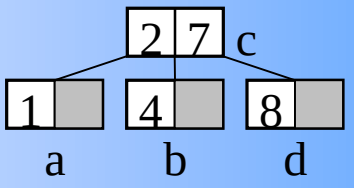


節點a分裂，鍵值2上提

(3)加入8

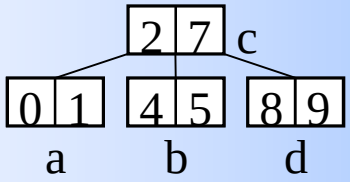


(4)加入4

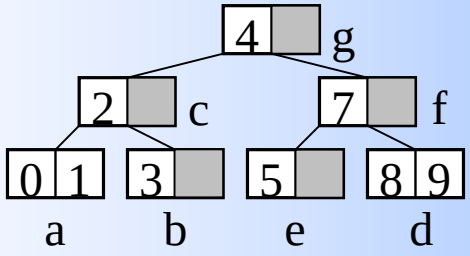


節點b分裂，鍵值7上提

(5)加入5, 9, 0



(6)加入3

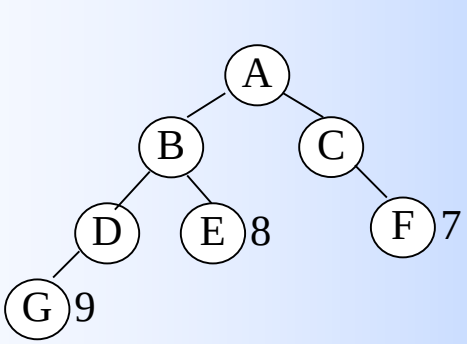


節點b分裂，鍵值4上提

節點c分裂，鍵值4上提

6-11 Huffman 樹

外部路徑長 (external path length) ：由樹根到每個外部節點的路徑長度之總和



路徑長

0

1

2

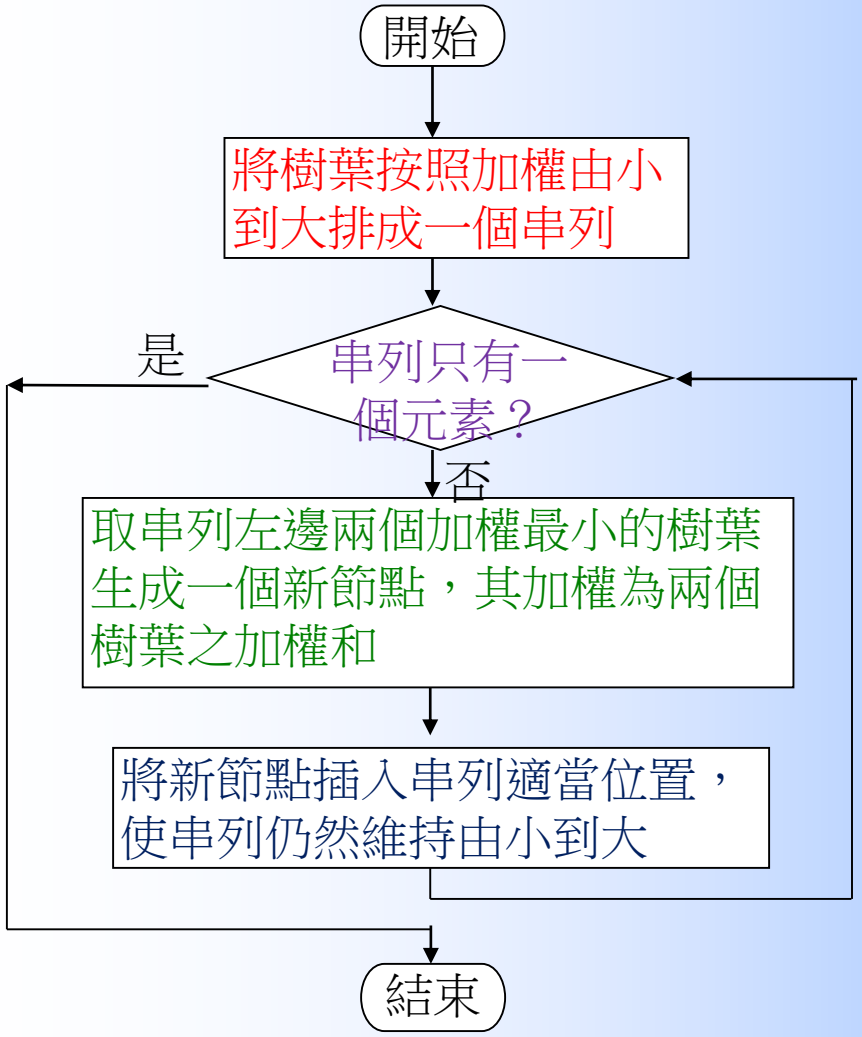
3

此樹的外部路徑長 = 2 (E) + 2 (F) + 3 (G) = 7

加權外部路徑長 ：由樹根到每個外部節點的路徑長乘上該節點的加權之總和
此圖的加權外部路徑長WE = 2 * 8 (E) + 2 * 7 (F) + 3 * 9 (G) = **57**

對具有相同外部節點同時加權也相同的不同二元樹，其加權外部路徑長也不同。其中**加權外部路徑長最小**的二元樹稱為Huffman樹，或稱**最佳二元樹**。

Huffman演算法 ：由已知的外部節點建構 Huffman 樹（加權外部路徑長最小的樹 ）的過程



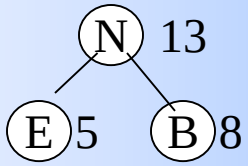
演算法：Huffman法建立編碼樹
(編碼字母表與頻率)
將樹葉(字母)按照加權(頻率)由小到大排成一個串列
當串列中多於一個元素，重複迴圈
從串列取出加權最小的兩個子樹生成一個新子樹，其加權為兩個子樹之加權和
將新子樹樹根插入串列適當位置，使串列仍然維持由小到大
迴圈結束
演算法結束

假設有ABCDEF六個樹葉，其加權分別為15, 8, 30, 27, 5, 15

1. 將樹葉按照加權，由小到大排序好，成為一有序串列。

E / 5,　B / 8,　A / 15,　F / 15,　D / 27,　C / 30

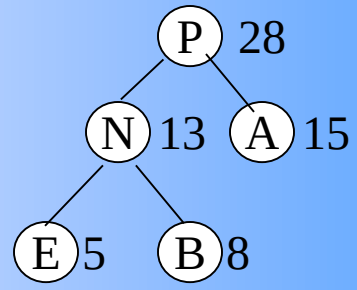
2. 取串列最左邊兩個加權最小的樹葉，作為一新節點N的兩個子節點，新節點N的加權為兩個樹葉的加權之和



3. 將節點N放入有序串列的適當位置，使串列保持次序。

N / 13,　A / 15,　F / 15,　D / 27,　C / 30

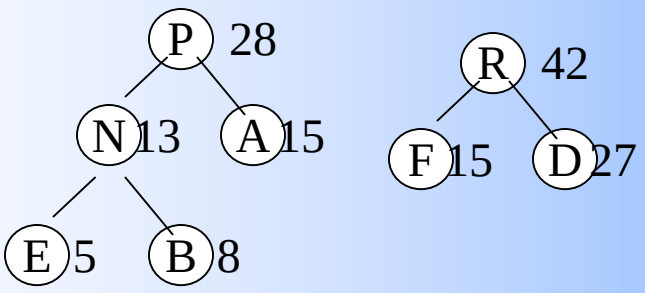
4. 取最左邊兩個節點 N 和 A，合成新節點 P



5. 將節點 P 放入有序串列的適當位置，使串列保持次序。

F / 15,　D / 27,　P / 28,　C / 30

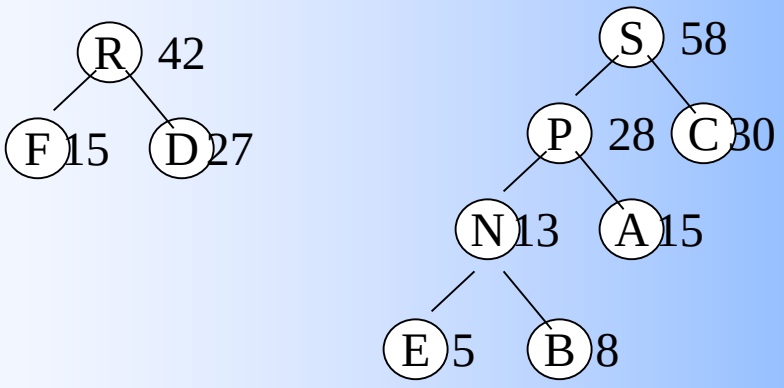
6. 取最左邊兩個節點 F 和 D，合成新節點 R



7. 將節點 R 放入有序串列的適當位置，使串列保持次序。

P / 28,　C / 30,　R / 42

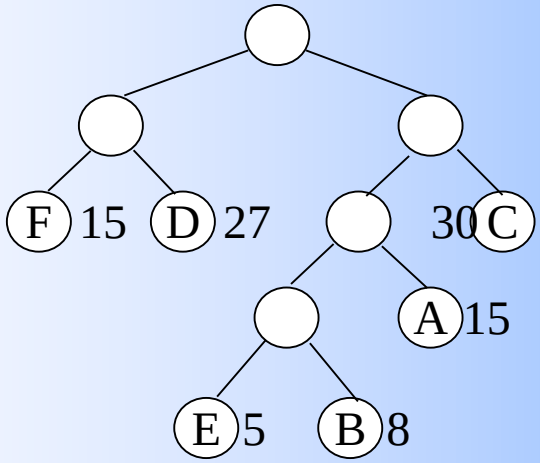
8. 取最左邊兩個節點 P 和 C，合成新節點 S



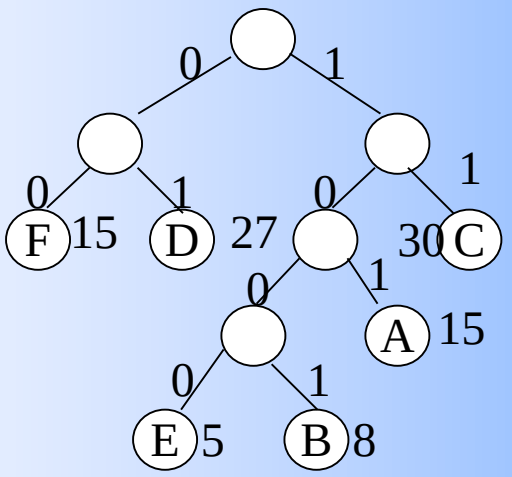
9. 將節點 S 放入有序串列的適當位置，使串列保持次序。

R / 42,　S / 58

10. 取最左邊兩個節點 R 和 S，合成新節點 T，只剩一個節點，停止



Huffman樹經常用在處理資料的壓縮編碼。假設共有 6 個字母需要編碼，分別是A/15, B/8, C/30, D/27, E/5, F/15，合起來剛好100 %。我們依照前述 Huffman 演算法建構 Huffman樹：



A的編碼 = 右左右 = 101
B的編碼 = 右左左右 = 1001
C的編碼 = 右右 = 11
D的編碼 = 左右 = 01
E的編碼 = 右左左左 = 1000
F的編碼 = 左左 = 00

編碼一篇1000個字母的文章。根據字母出現的頻率，它們的出現次數分別是：

A 出現的次數 = 1000 * 15 % = 150

B 出現的次數 = 1000 * 8 % = 80

C 出現的次數 = 1000 * 30 % = 300

D 出現的次數 = 1000 * 27 % = 270

E 出現的次數 = 1000 * 5 % = 50

F 出現的次數 = 1000 * 15 % = 150

因此總共需要的bit數

150 * 3 + 80 * 4 + 300 * 2 +270 * 2 + 50 * 4 + 150 * 2 = **2410**

未經編碼，每個字母用3個位元表示，(8 種組合)，共需**3000**個位元

$$\text{壓縮率} = \left(1 - \frac{2410}{3000}\right) * 100\ \% \approx 20\ \%$$