

# CH5、Tree

樹與二元樹

## 目錄：

定義

表示法(4 種)

Linked List、二元樹、Child Sibling、括號法

二元樹

比較表

三定理

特殊二元樹

Skewed BT、Full BT、Complete BT、Strict BT

BT 表示法

BT 的應用

Traversal

反向

Traversal 演算法

應用演算法(7 個)

Count、Height、Leaf Node、Copy、Equal、SWAP BT、Expression

Binary Search Tree

Insert/Consturction、Search、Delete

Thread BT

Tree/Forest 與 BT 的轉換

Forest Traversal

[補充]n Node 可形成幾種 BT

Heap

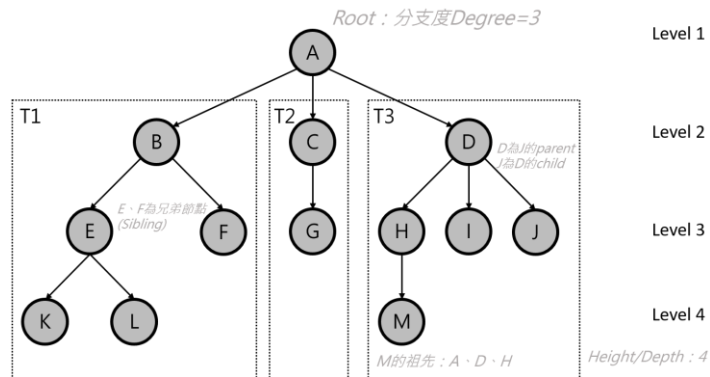
Max-Heap、Min-Heap

Top-Down、Bottom-Up

## Tree

Def：由一個或多個 Node 組成集合，具有：

1. 有一特定 Node 為 Root(樹根)，此 Node 沒有 Parent.
2. 其餘成為( $n \geq 0$ )個互斥集合  $T_1, T_2, \dots, T_n$  皆為一棵樹為 Root 之 Subtree(子樹)



Leaf Node(Terminal Node) : F, G, I, J, K, L, M( $Degree=0$ )

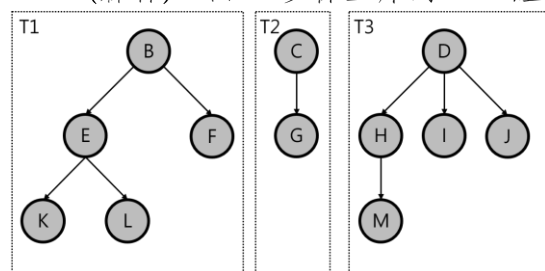
Non-terminal : A, B, C, D, E, H

Height/Depth : 4(若有註明從第 0 層開始，則為 3，要注意)

把 Root(A) 拔掉，可變成 3 個獨立的子樹：T1, T2, T3(虛線框內)

Tree Degree=Max(各 Node Degree)

Forest(森林)：由 0~多棵互斥的 Tree 組成



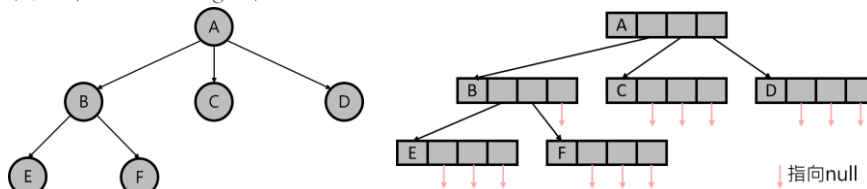
## Tree 的表示法

[法一]採 Linked List

Node Structure

Data	Link <sub>1</sub>	Link <sub>2</sub>	...	Link <sub>k</sub>
------	-------------------	-------------------	-----	-------------------

例：( $k$  為 Tree Degree)



問題：太浪費空間

說明：Tree Degree= $k$ ，Node 數= $n$  個，需準備  $n*k$  個 Links，又實際上有用到的只有  $n-1$  個，因此浪費率  $n*k-(n-1) / nk \approx (k-1)/k \Rightarrow$  可知  $k$  越小越好

$\Rightarrow$  [法二]化成二元樹再存

[法三]採 Child-Sibling 方法

Def：

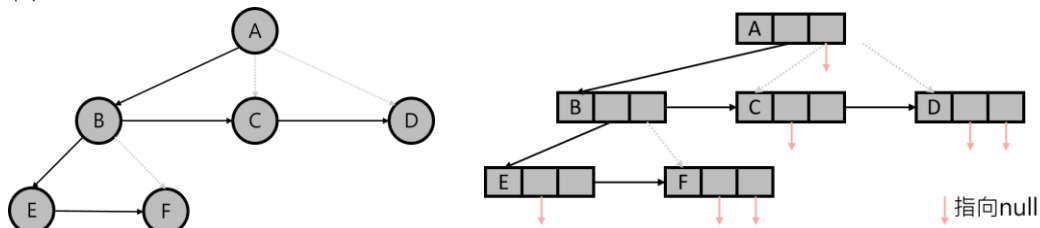
Node Structure：

Data	Child	Sibling
------	-------	---------

Child：Link 指向”Leftmost Child”

Sibling：Link 指向”Next Right Sibling Node”

例：

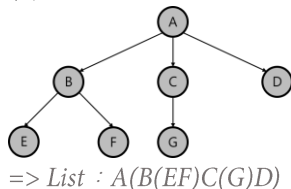


[法四]括號法(Lists)

作法：

1. 父(子...子)
2. 表示父子間的形成關係
3. 可以用巢狀(Nested)表示(括號內有括號)

例：

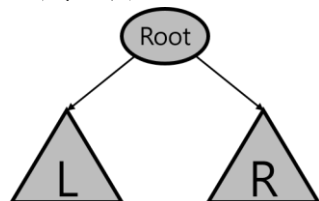


二元樹(Binary Tree, BT，也稱 Ordered Tree 有序樹)

Def：由 Node 組成之有限集合，由：

1. Root(D)
2. 左子樹(L)
3. 右子樹(R)

組成，例：

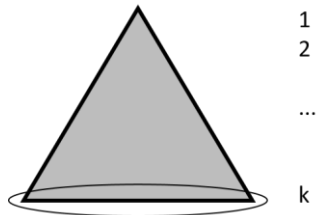


## 比較表

Tree	Binary Tree
不可為空	可為空
Degree $\geq 0$	Degree 介於 0~2 之間
“沒有”順序之分	“有”左右之分

## BT 三定理

1. 第  $i$  level 之 Node 數最多為  $2^{i-1}$  個



第  $k$  層最多有  $2^{k-1}$  個

[證明]：歸納法

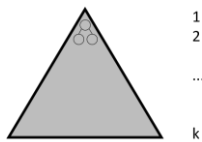
- (1) 恆真
- (2) 令  $i=k-1$  時成立
- (3) 當  $i=k$  時亦成立

(1) 當  $i=1$  時，只有 Root  $\Rightarrow 2^{1-1}=2^0=1$ ，成立

(2) 令  $i=k-1$  時皆成立

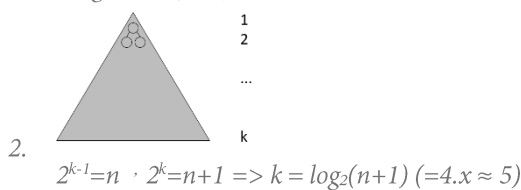
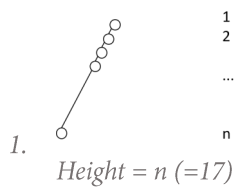
(3) 當  $i=k$  時，最多 Node 為： $k-1$  層之最多 Node 數  $\times 2 = 2^{k-1-1} \times 2 = 2^{k-1}$ ，成立。由(1)、(2)、(3)可得證

2. 高度為  $k$  之 BT，總 Node 數為  $2^k-1$



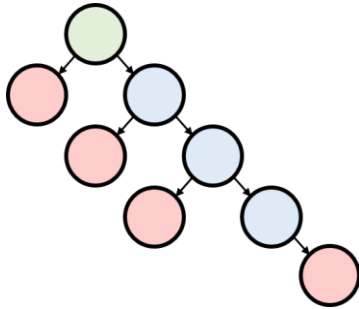
總 Node 數  $= 2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$

例 1：一 BT 有  $n(17)$  個 Node，問最高為何？最低為何？



1. BT 有用的 Link 數=99，則總 Node 數為何？
2. BT 有 200 個 Leaf Node，問最小高度為何？
3. BT 有 200 個 Node，問最小高度為何？

3. 一非空的 BT 為  $n_0$  為 Leaf Node 為 0 之個數， $n_2$  為 Degree=2 之 Node，則  $n_0=n_2+1$



$$2. \quad 56 = n_0 + 1 + 2 + 3 + 4 + 5 \Rightarrow n_0 = 41$$

例 3：承上，若 Degree=k，則  $n_0$  為何？

$$N=B+1 = n_1*1 + n_2*2 + n_3*3 + \dots + n_k*k + 1 = 1*1+2*2+\dots+k*k+1$$

$$N = n_0 + n_1 + \dots + n_k = n_0 + 1+2+\dots+k$$

$$n_0 = 12 + 22 + \dots + k2 + 1 - (1+2+\dots+k) = k(k+1)(2k+1)/6 + 1 - k(k+1)/2$$

$$= [k(k+1)(2k+1) - 3k(k+1) + 6] / 6$$

例 4：— Quard Tree，Tree Degree=4，且 Non-leaf Node 之 Degree 皆為 4，若其 Leaf Node 數量=x，問總 Node 數為何？

$$\text{已知 } N=B+1 = n_4*4 + 1$$

$$\text{已知 } n_0=x, N = n_0 + n_4$$

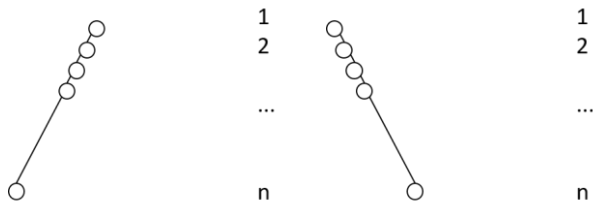
$$\text{故 } 4n_4 + 1 = x + n_4$$

$$\Rightarrow n_4 = (x-1)/3, \text{ 故 } N = x + (x-1)/3$$

## 特殊二元樹

### 一、Skewed BT(斜曲二元樹)

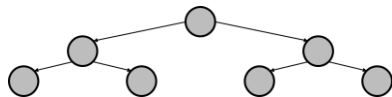
分為左斜曲 BT(Left Skewed BT)與右斜區 BT(Right Skewed BT)



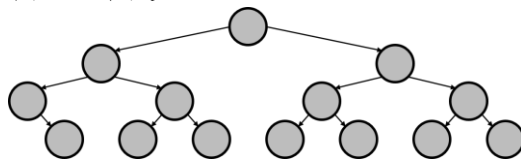
### 二、Full BT(完滿二元樹)

Def：高度為 k，具  $2^k-1$  個 Node，謂之

例 1：是



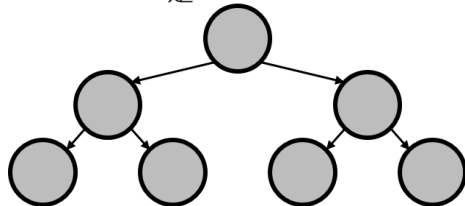
例 2：不是



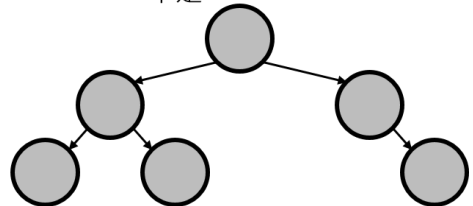
### 三、Complete BT(完整二元樹)

Def：高度為 k，具 n 個 Node，其依序編號後會跟高度為 k 之 Full BT 的前 n 個 Node 一一對應

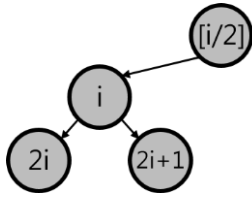
是



不是



定理：



if  $[i/2] < 1$ ，沒有父點  $\Rightarrow$  Root

if  $(2i > N)$ ，左兒子不存在

if  $(2i+1 > N)$ ，右兒子不存在

[證明] 先證 1. 成立，則 2. 成立，1. 與 2. 成立，則 3. 成立

1. 當 Node 編號=1，Root。而 Root 的左兒子=2(必成立)
2. 令編號  $i-1$  之前，上述皆成立
3. 則編號= $i$  時，其左兒子編號必為： $i-1$  之左兒子編號數+2  
 $= 2_{i-2} + 2 = 2_i$ ，可知 2. 亦成立。1. 與 2. 成立，則 3. 成立，故得證

例 1：一 Complete BT 有 1000 個 Node，則：

1. 最後一個 Parent 編號為何？
2. 編號 250, 251 是否為 Sibling？
3. 250 的祖父編號為何？
4. 371 的左子點、右子點、父點編號為何？
5. 編號 512 之左兒子為何？

1.  $1000/2 = 500$
2.  $250/2 = 125 = 251/2$ ，故此 2 數是兄弟
3.  $250/2 = 125$
4.  $371*2=742$ 、 $742+1=743$ 、 $374/2=185$
5.  $512*2=1024 > 1000$ ，故無左兒子不存在

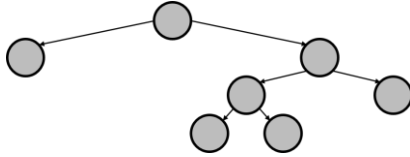
[延伸題]

6. 第 7 level 的第 1 個 Node 編號為何？
7. Depth 為何？
8. 375 位於 level 幾？
9. Leaf Node 數為何？
10. Degree = 1 的個數為何？
11.  $n_1$  之數量永為 0~1 之間(True/False)？

6.  $2^6 - 1 + 1 = 2^6 = 64$
7.  $\text{Log}(1000+1) \approx 10$
8.  $\text{Log}(375+1) = 9$
9. 500 個
10. 看最後父點：1 個
11. True

#### 四、Strict BT(嚴格/嚴謹二元樹)

Def：所有 Non-leaf Node Degree 皆為 2，即  $n_1=0$



$$N=n_0+n_2$$

例 1：

1. Full BT 一定是 Strict BT ?
2. Complete BT 一定是 Strict BT ?
3. Strict 必為 Full BT ?
4. Strict 必為 Complete BT ?

1. *True*
2. *False*
3. *False*
4. *False*

例 2：

- a.  $n_0=n_2+1$
- b.  $N=2n_0-1$
- c.  $\text{Height}=\log_2(n+1)$
- d.  $n_0+n_2 \leq N \leq n_0+n_2+1$

問：

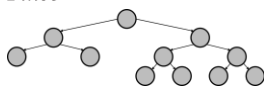
1. BT 符合：
2. Full BT 符合：
3. Complete BT 符合：
4. Strict BT 符合：

1. *a*
2. *a, b, c, d*
3. *a, c, d*
4. *a, b, d*

例 3：下列何者為非？

1. BT 中，each Node 皆有 Parent ?
2. 空 Link 必比非空 Links 多 1 個 ?
3. Root 之左、右為 Full BT，則整棵樹為 Full ?
4. Root 整棵樹為 Complete，則左、右子樹亦是 Complete ?

1. *False*
2. *False*
3. *False*



4. *True*

Note：通常 Top-down 會成立；Bottom-up 不見得



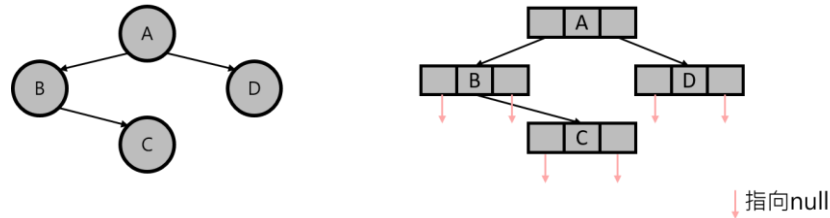
## BT 表示法

[法一] Linked List

Node Structure

Lchild	Data	Rchild
指向左兒子		指向右兒子

例：



分析：若 BT 有  $n$  個 Node，需  $2n$  個 Link  $\Rightarrow$  浪費  $2n - (n-1) = n+1$

優點：Insert、Delete 方便

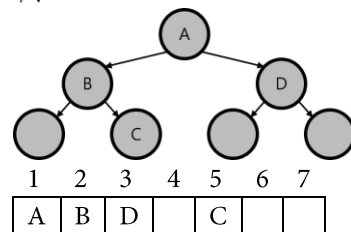
缺點：

1. 浪費 Link(約一半 Link 數沒用)
2. 對父點不得而知

[法二] Array(Heap 適用)

作法：將 BT 想成 Full BT，故需準備一個一維陣列  $A[1: 2^k-1]$ ， $k$  為高度

例：

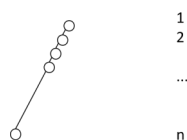


優點：

1. 不需 Link Space
2. 可以輕易找到父點：Full，所以可用 Complete 的定理

缺點：

1. Insert、Delete 不便
2. 對斜曲樹不適用



Full 需  $2^{10}-1 = 1023$ ，浪費： $1023-10 = 1013$

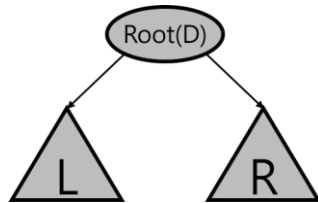
## Traversal(追蹤/走訪)

目的：拜訪 BT 中所有的 Node 一次

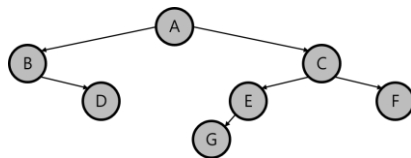
規定：L 要在 R 之前拜訪

前序 DLR、中序 LDR、後序 LRD(也有書將中間的 D 稱之為 N)

方法：



例 1：

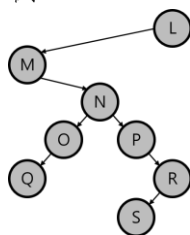


前：ABDCEGF

中：BDAGECF

後：DBGEFCA

例 2：



前：LMNOQPRS

中：MQONPSRL

後：QOSRPNML

## 反向

考法：給前序+中序 or 後序+中序，決定一棵唯一的 BT

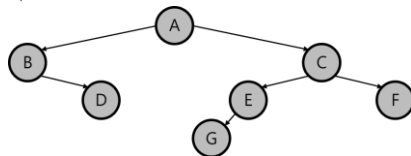
前序：從左而右找 Root

後序：從右而左找 Root

例 1：

前：ABDCEGF

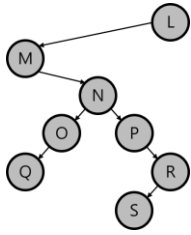
中：BDAGECF



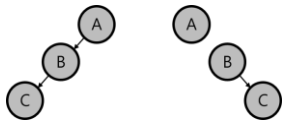
例 2：

前：QOSRPNML

中：MQONPSRL



例 3：試說明為何給前+後序，無法決定一棵唯一的 BT？



前序：ABC、後序：CBA

由上例可知，給前序+後序，無法決定一棵唯一的 BT

例 4：

1. 此 BT 的中序和前序相等？

2. 此 BT 的中序和後序相等？

3. 此 BT 的前序和後序相等？

1. 中序=前序(LDR=DLR)

(1) 當空或只有 Root

(2) 沒有 L(左子樹) => 即右斜曲

2. 中序=後序(LDR=LRD)

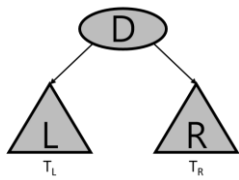
(1) 當空或只有 Root

(2) 當沒有 R(右子樹) => 即左斜曲

3. 前序=後序(DLR=LRD)

(1) 當空或只有 Root 時才成立

例 5：試證後序+前序無法決定唯一 BT



依  $T_L$ 、 $T_R$  為空與否，分成下最 4 種：

1.  $T_L=T_R$ =空

則：

(1) 前序：D

(2) 後序：D

2.  $T_L \neq$ 空， $T_R$ =空

則：

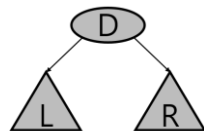
(1) 前序：DT<sub>L</sub>'(T<sub>L</sub>'為 T<sub>L</sub> 前序)

(2) 後序：T<sub>L</sub>\*D(T<sub>L</sub>\*為 T<sub>L</sub> 後序)

3.  $T_L = \text{空}, T_R \neq \text{空}$   
則：  
(1) 前序： $DT_R'(T_R' \text{ 為 } T_R \text{ 的前序})$   
(2) 後序： $T_R^*D(T_R^* \text{ 為 } T_R \text{ 的後序})$
4.  $T_L \neq \text{空}, T_R \neq \text{空}$   
(1) 前序： $DT_L'T_R'()$   
(2) 後序： $T_L^*T_R^*D()$

結論：由上可知，當  $T_L' = T_R'$ ， $T_L^* = T_R^*$ ，則無法判別 2. 與 3.，故結果不唯一

Note：注意題目會 Redefine。例：



$RLD \Rightarrow CAB$

$RDL \Rightarrow CAB$

$DRL \Rightarrow ACB$

例 6：試證前序+中序，決定唯一的 BT(採數學歸結法)

證明：

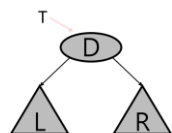
1. 當 Node 數=0(空)，此時前序=中序。故可決定一唯一的 BT
2. 令 Node 數 $\leq n-1$  時，此定理成立
3. 當 Node 數= $n$  時，自前序取第一個 Node 為  $D(\text{Root})$ ，依  $D$  將中序切割成  $D$  以及  $TL'$ (於  $D$  的左邊)、 $TR'$ (於  $D$  的右邊)。令  $TL'$  的長度= $|TL'|$ 、 $TR'$  的長度= $|TR'|$   
於前序中自第 2 個  
開始取長度 $|TL'|$ 個元素成為  $T_L^*$   
之後再取 $|TR'|$ 個長度成為  $T_R^*$   

0	$T_L^*$	$T_R^*$
---	---------	---------
4. 又  $TL'$  為左子樹的中序、 $T_L^*$  為左子樹的前序，且 $|T_L^*| \leq n-1$ 。故可決定唯一的左子樹  
同上，可決定唯一的右子樹。因為左、右子樹都唯一，故整個 BT 也唯一，得證。

## Treaversal 演算法

Node Structure：

Lchild	Data	Rchild
--------	------	--------



程式：

```

class Node
{
    Node *Lchild;
    int data;
    Node *Rchild;
};
  
```

前序：

```
void preorder(Node *T)
{
    if(T!=null)
    {
        print(T->data);    //D
        preorder(T->Lchild); //L
        preorder(T->Rchild); //R
    }
}
```

中序：

```
void preorder(Node *T)
{
    if(T!=null)
    {
        preorder(T->Lchild); //L
        print(T->data);    //D
        preorder(T->Rchild); //R
    }
}
```

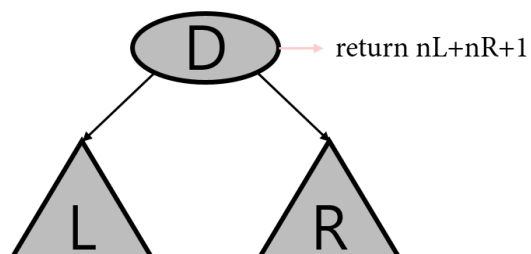
後序：

```
void preorder(Node *T)
{
    if(T!=null)
    {
        preorder(T->Lchild); //L
        preorder(T->Rchild); //R
        print(T->data);    //D
    }
}
```

### 應用演算法

#### 1. Count：計算總 Node 數

概念：

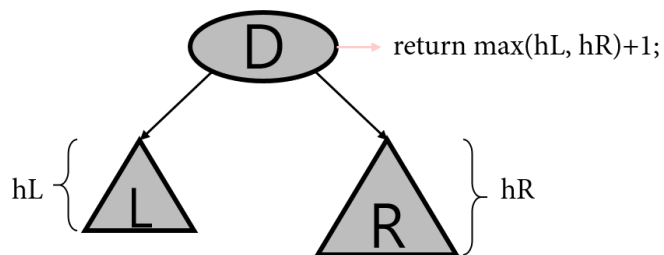


程式：

```
int count(Node *T)
{
    if(T==null) return 0;
    else
    {
        int nL=count(T->Lchild);
        int nR=count(T->Rchild);
        return nL+nR+1;
    }
}
```

## 2. Height：計算 Tree 的高度

概念：

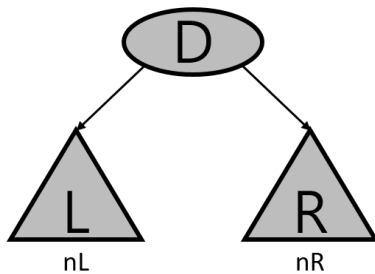


程式：

```
int height(Node *T)
{
    if(T==null) return 0;
    else
    {
        int hL=height(T->Lchild);
        int hR=height(T->Rchild);
        return max(hL, hR)+1;
    }
}
```

## 3. Leaf Node：計算 Leaf 數量

概念：

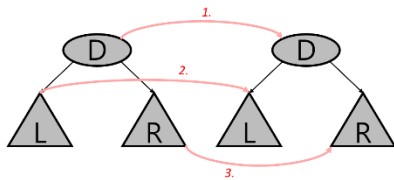


程式：

```
int countLeaf(Node *T)
{
    if(T==null) return 0;
    else
    {
        int nL=countLeaf(T->Lchild);
        int nR=countLeaf(T->Rchild);
        if(nL+nR==0)
            return 1;
        else
            return nL+nR;
    }
}
```

#### 4. BT 的 Copy

概念：



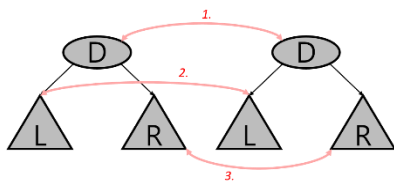
程式：

```
Node *copy(Node *orig)
{
    Node *t=new Node();
    if(orig!=null)
    {
        t->data = orig->data;
        t->Lchild = copy(orig->Lchild);
        t->Rchild = copy(orig->Rchild);
    }
    else t=null;
    return t;
}
```

#### 5. Equal：判別 s, t 2 個 BT 是否相等

概念：

- (1)  $s=t=\text{空} \Rightarrow \text{true}$
- (2)  $s \neq \text{空}, t \neq \text{空} \Rightarrow \text{check}$
- (3) otherwise (一空一非空)  $\Rightarrow \text{false}$

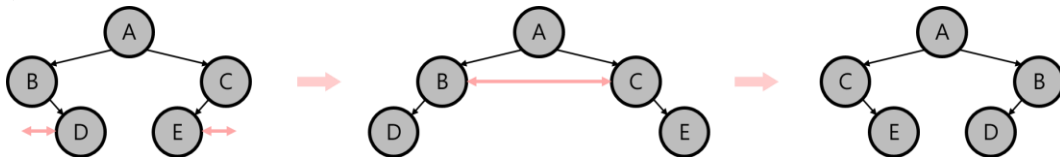


程式：

```
bool Equal(Node *s, Node *t)
{
    bool flag=false;
    if(s==null && t==null)
        return true;
    else if(s!=null && t!=null)
    {
        if(Equal(s->data, t->data)
            if(Equal(s->Lchild, t->Lchild)
                flag=Equal(s->Rchild, t->Rchild)
            return flag;
        }
    }
    else return false;
}
```

## 6. SWAP BT：將左、右子點互換

概念：



由下而上，左、右對調

程式：

```
void swap(Node *T)
{
    if(T!=null)
    {
        swap(T->Lchild);
        swap(T->Rchild);
        Node *tmp=T->Lchild;
        T->Lchild=T->Rchild;
        T->Rchild=tmp;
    }
}
```

## 7. Expression Tree

概念：利用 BT 表示運算式

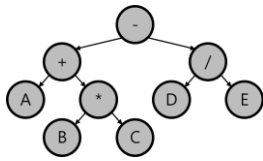
(驗證：Proder 為 Prefix；Inorder 為 Infix；Postorder 為 Postfix)

原則：

- (1) Non-leaf 表示”operator”運算子
- (2) Leaf 表示”operand”運算元
- (3) Priority 愈高的運算子在愈下層(先執行)

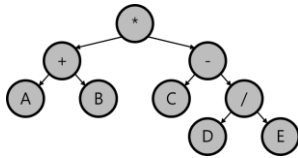


例 1：A+(B\*C)-D/E，建 Expression Tree



Preorder：-+A\*BC/DE

例 2：(A+B)\*(C-D/E)，建 Expression Tree



Recursive 演算法：計算 Expression BT 之結果：以 Linked List 表示

Node Structure

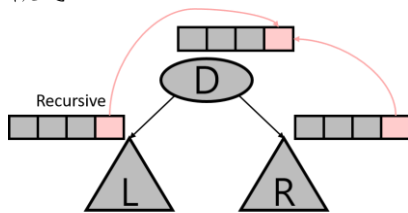
Lchild	Data	Rchild	Result
--------	------	--------	--------

Note：

Data：存 operator 或 operand

Result：運算結果 or 變數 or 常數值

概念：



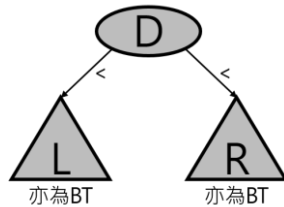
程式：

```
int Eval(Node *T)
{
    if(T->Lchild);
    {
        Eval(T->Lchild);
        Eval(T->Rchild);
        switch(T->data)
        {
            case"變數或常數":
                T->Result=T->data;
            case"二元運算子":
                T->Result=exec((T->Lchild)->Result,T->data, (T->Rchild)->Result);
            case"單元運算子":
                T->Result=exec(T->data, (T->Rchild)->Result);
        }
        return T->Result;
    }
}
```

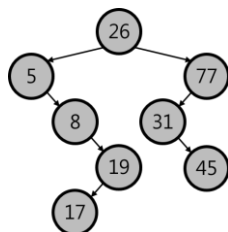
## Binary Search Tree(二元搜尋樹)

Def：為一 BT，可為空，若不為空，則：

1. 左子樹鍵值小於 Root
2. 右子數鍵值大於 Root
3. 左、右子樹亦為 BST



例 1：依下列 data 建一 BST：26, 5, 77, 8, 19, 31, 17, 15

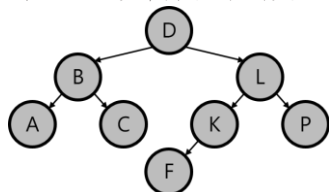


利用中序追蹤可得：由小到大排列(Sorting 效果)：5, 8, 17, 19, 26, 31, 45, 77

例 2：BST 之 postorder 為：ACBFKPLD，問此 BT 之前序？

反向：後序+中序可得一唯一 BT

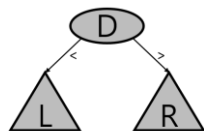
為 BST，故中序由小到大：ABCDFKLP，故可得圖：



前序：DBACKLFP

## Search in BST

概念：



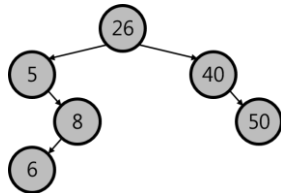
欲 Search key(x)於 BST 中，則：

1.  $x = \text{Root} \Rightarrow \text{Found}$
2.  $x < \text{Root} \Rightarrow$  找左子樹
3.  $x > \text{Root} \Rightarrow$  找右子樹

程式：

```
bool search(Node *T, int x)
{
    if(T==null) return false;
    else
    {
        if(T->data==x) return true;
        else if(x<T->data)
            return search(T->Lchild, x);
        else
            return search(T->Rchild, x);
    }
}
```

例 1：採 BST 之 Search，平均比較次數為何？



$$(1+2+2+3+3+4)/6 = 2.5$$

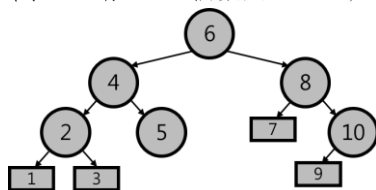
例 2：Worst case 的平均比較次數(Skewed BT)

$$(1+2+\dots+n)/n = [n(n+1)/2] / n = (n+1)/2 \Rightarrow O(n)$$

例 3：Best case 的平均比較次數(Full BT)

$$O(\log n)$$

例 4：有 1~10 個數於 BST，如下圖，求平均比較次數(含失敗值)



失敗本身不算  
 $25/10 = 2.5(\text{次})$

比較表

	Best Case	Worst Case
Search(x)	$O(\log n)$	$O(n)$
Insert(x)	(Full/Complete)	(Skewed)
Delete(x)		

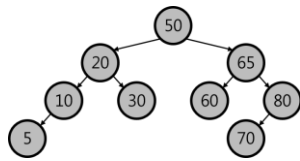
## Delete a node in BST :

分析：

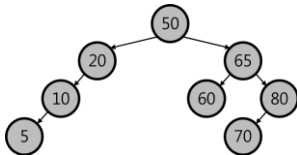
1. x 為 Leaf Node => Delete
2. x 為 Non-leaf
  - (1) 有一個 child：將 Parent Link 指向 x 的 child
  - (2) 有 2 個 child
    - a. 拿左子樹最大值取代之
    - b. 拿右子樹最小值取代之

例：問：

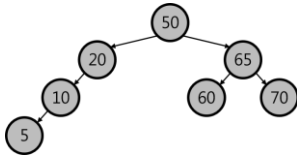
1. Delete “30”
2. Delete “80”
3. Delete “50”



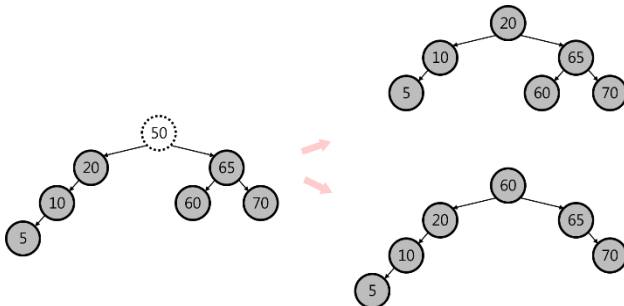
1.



2.



3.



## Thread BT(引線二元樹)

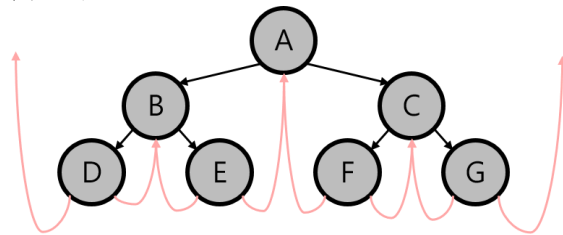
緣由：n 個 Node 之 BT，以 Link 表示會有 n+1 條空 Links，為充分利用這些 Links，故將之改為『引線(Thread)』

常見為『中序引線 BT』

Def：若 Node x 的

1. Lchild 為空，視為左引線=>將 Link 指向 x 的中序之前一個 Node
2. Rchild 為空，視為右引線=>將 Link 指向 x 的中序之後一個 Node

例：中序：DBEAFCG



Node Structure：

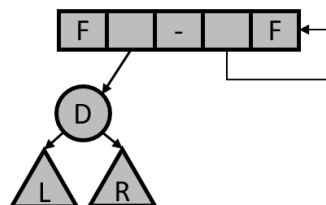
LeftThread	Lchild	Data	Rchild	RightThread
False 表已 Link				False 表已 Link
True 表引線				True 表引線

Note：在 Thread BT 中尚需引入一個 Head Node(串列首)，其格式如下：

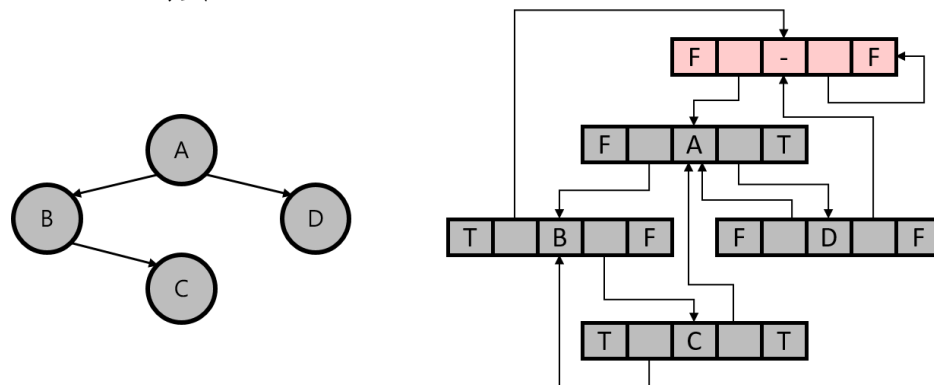
1. 空樹



2. 非空



Thread BT 表示：



Why? 優點：

1. 充分利用 **Links**
2. 容易存取 **x** 的中序前/後繼者
3. 簡化中序追蹤 => 不需 **Recursive**

演算法：

**Insuc(x)** => 找 **x** 的中序後繼者

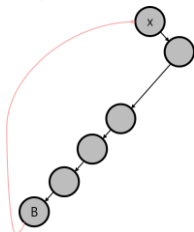
**Pre(x)** => 找 **x** 的中序前繼者

**Insuc(x)**

1. **x** 無右兒子，當 **RightThread=True**，**x**→**Rchild** 即是



2. **x** 有右兒子，將 **temp** 一直找到最後一個左兒子，此 **Node** 即是 **x** 的後繼者



程式：

```
Node Insuc(x)
{
    temp=x->Rchild;
    if(x->RightThread==false)    //有右子點
    {
        while(temp->LeftThread==false)
            temp=temp->Lchild;
        return temp;
    }
}
```

**Pre(x)**

程式：

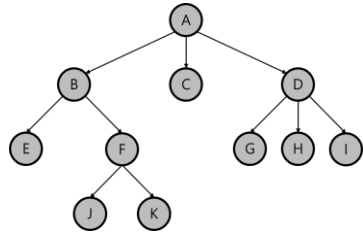
```
Node Pre(x)
{
    temp=x->Lchild;
    if(x->LeftThread==false)    //有左子點
    {
        while(temp->RightThread==false)
            temp=temp->Rchild;
        return temp;
    }
}
```

## Tree 轉成 BT

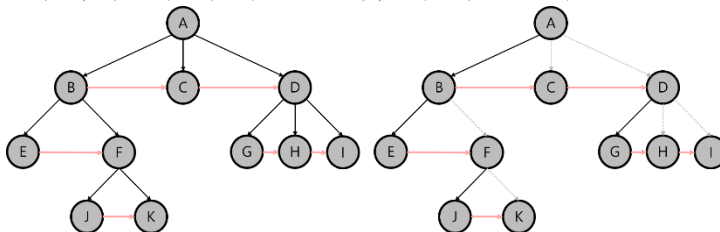
採用 Leftmost child next right sibling 之方式，步驟：

1. 將 Sibling 間的 Link 接起
2. 只留最左兒子的 Link，其餘父子 Link 皆 Delete
3. 以最左兒子為中心，順時針轉 45 度

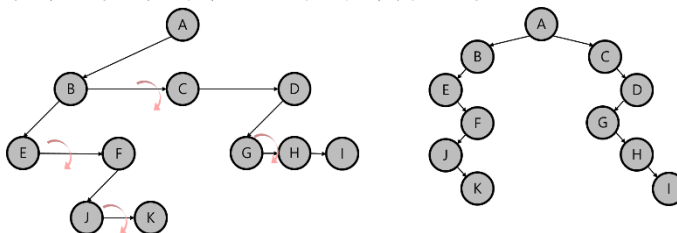
例 1：轉之轉成 BT



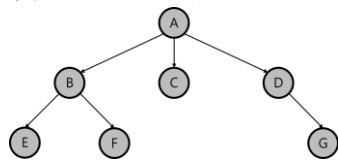
1. 將 Sibling 間的 Link 接起
2. 只留下最左兒子的 Link，其餘父子 Link 皆 Delete



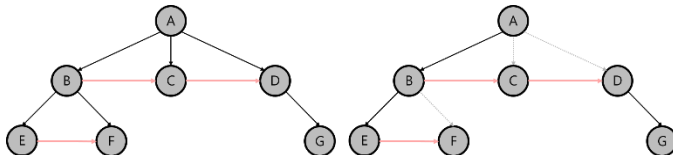
3. 以最左兒子為中心，順時針轉 45 度



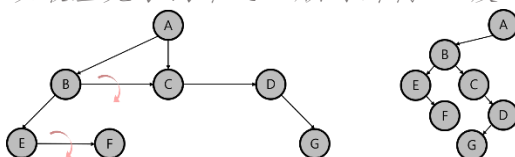
例 2：轉之轉成 BT



1. 將 Sibling 間的 Link 接起
2. 只留下最左兒子的 Link，其餘父子 Link 皆 Delete



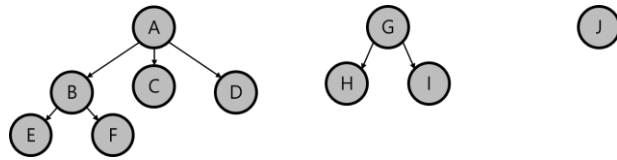
3. 以最左兒子為中心，順時針轉 45 度



## Forest 轉成 BT

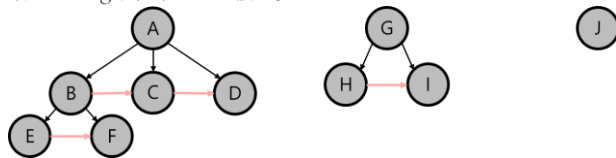
1. 將各 Tree 轉成 BT
2. 將各 BT 的 Root 連起
3. 以最左 Root 為主，順時針將右邊兄弟轉 45 度

例 1：

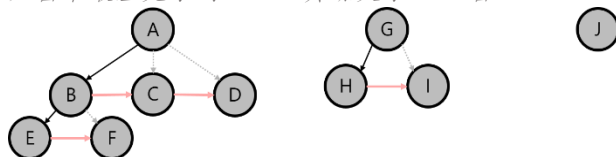


1. 將各 Tree 轉成 BT

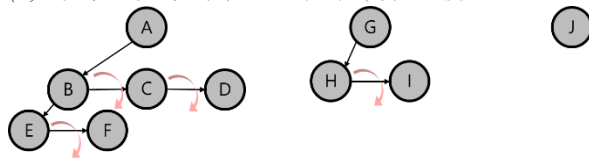
(1) 將 Sibling 間的 Link 接起



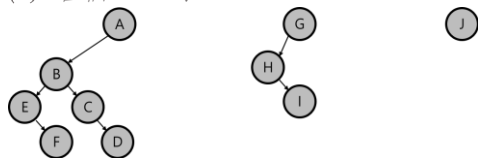
(2) 只留下最左兒子的 Link，其餘父子 Link 皆 Delete



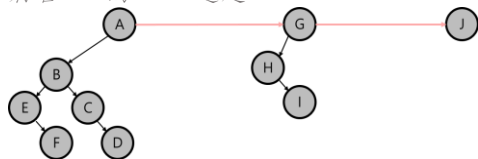
(3) 以最左兒子為中心，順時針轉 45 度



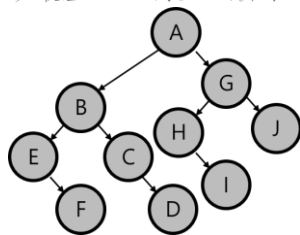
(4) 各個 Tree 的 BT



2. 將各 BT 的 Root 連起



3. 以最左 Root 為主，順時針將右邊兄弟轉 45 度



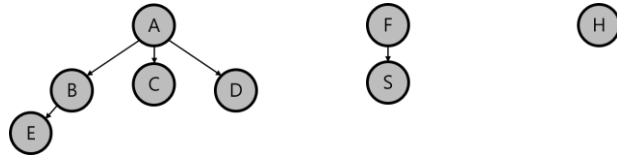


## Forest Traversal

森林的前序、中序追蹤 = 將之化為 BT 的前序、中序  
但是：

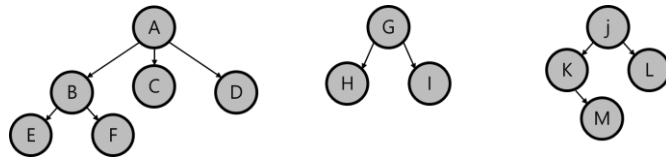
Forest 的後序  $\neq$  化成 BT 的後序

例 1：



*EBCDSHFA*

例 2：



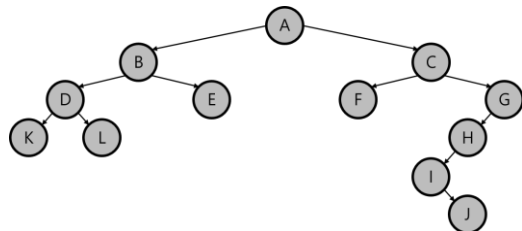
後序：*EFBCDHIMKLJGA*

## BT 轉成 Tree/Forest

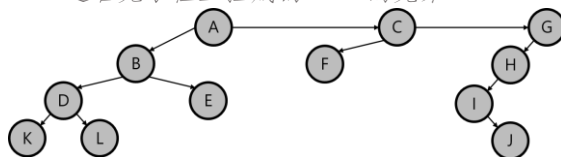
步驟：

1. Root 之右兒子往上拉成為 Root 的兄弟
2. Delete Root 之間的 Sibling Link
3. 將每棵 BT 化成 Tree
  - (1) 將右兒子逆時針轉 45 度
  - (2) 將父點與往上拉的 Node 加上 Link
  - (3) 將各 Sibling 之間的 Link Delete

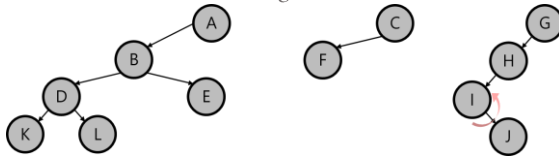
例 1：轉成 Forest



1. Root 之右兒子往上拉成為 Root 的兄弟

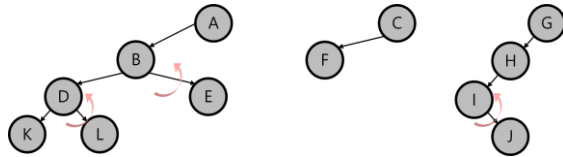


## 2. Delete Root 之間的 Sibling Link

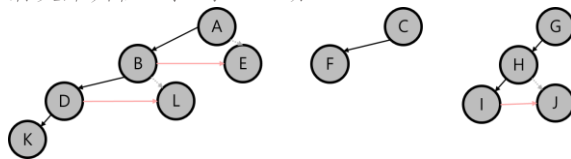


## 3. 將每棵 BT 化成 Tree

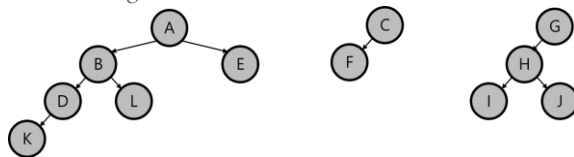
(1) 將右兒子逆時針轉 45 度



(2) 將父點與往上拉的 Node 加上 Link



(3) 將各 Sibling 之間的 Link Delete



[補充]n 個 Node 能形成幾種 BT ?

(等於同 n 筆 Data , 依序 push 、 pop 的合法 output 組合)

$$\frac{1}{(n+1)} \times C_n^{2n}$$

n=3 => 5 種

n=4 => 14 種

n=5 => 42 種

## Heap(堆積)

### 一、Max-Heap(最大堆積)

Def :

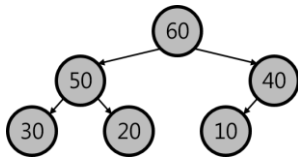
1. 為一"Complete BT"
2. 父點必大於子點鍵值
3. Root 為最大值

### 二、Min-Heap(最小堆積)

Def :

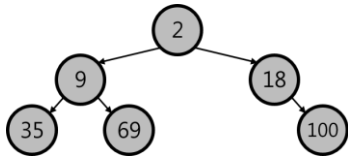
1. 為一"Complete BT"
2. 父點必小於子點鍵值
3. Root 為最小值

例 1 :



Max-Heap

例 2 :



不成立

## Heap Construction(建構)

分為 :

一、Top-Down( $O(n \log n)$ )

⇒ Data size 不知道

二、Bottom-Up( $O(n)$ )

⇒ Data 事先準備好適用

一、Top-Down 方式 : (為製作 Priority Queue 的 DS)

Max-Heap(Min-Heap)

1. 將欲加入的 Node(x)置於 last Node 的下一個位置

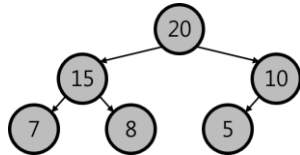
2. 由下往上比較

(1) 若  $x > (<) \text{Parent Node}$  , swap(x, Parent) , goto (2)

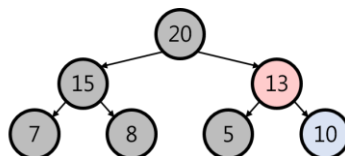
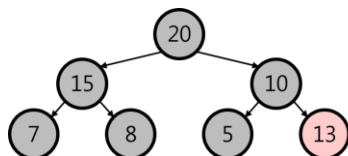
(2) 若  $x \leq (\geq) \text{Parent Node}$  , 停止 , x 插入於此

例 1 : 有一 Max-Heap 如下 :

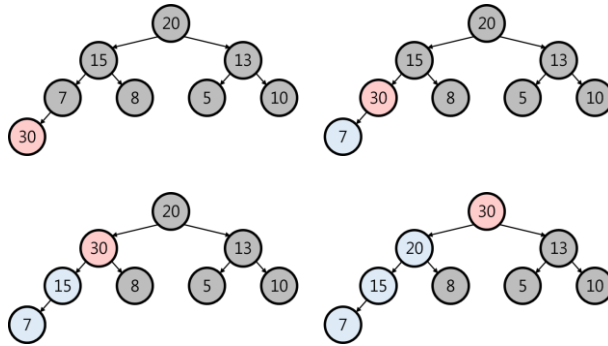
1. Insert "13"
2. Insesrt "30"



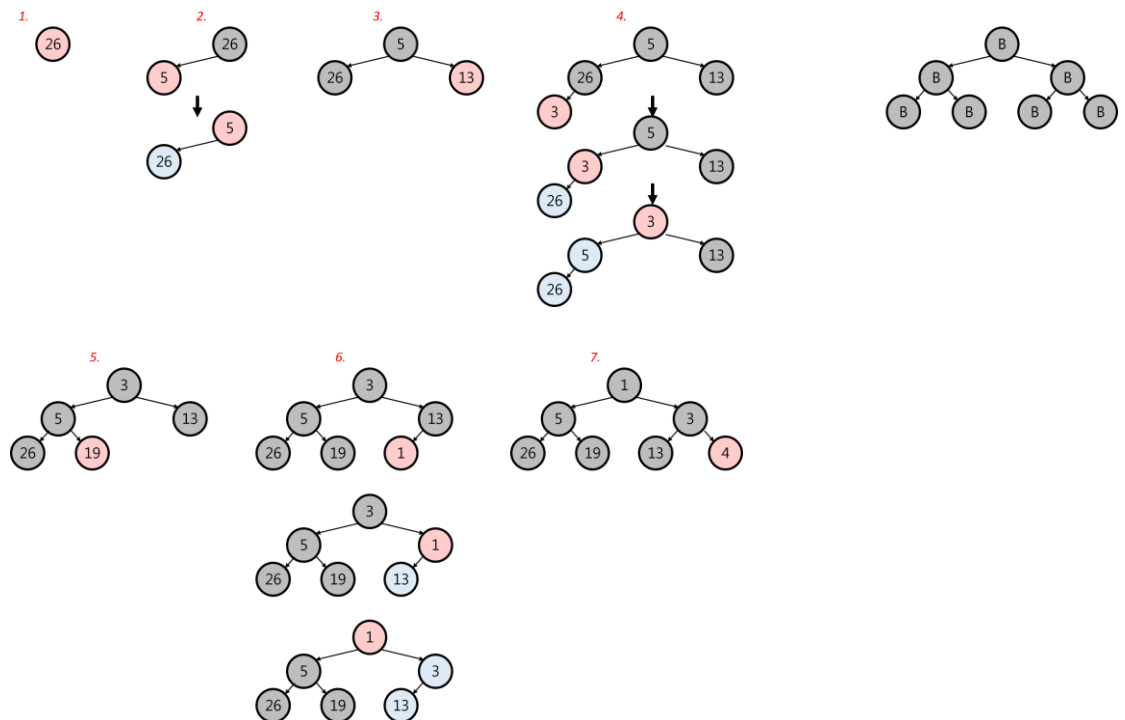
1.



2.



例 2 : Data : 26, 5, 13, 2, 19, 1, 4 , 建一 Min-Heap(採 Top-Down)



1	5	2	26	19	13	4	(多的一格當做 Swap 的 temp)
---	---	---	----	----	----	---	----------------------

Why Heap using array structure ?

1. 因為 Complete 適用
2. 在做 Heap 調整時，才能方便得知父點

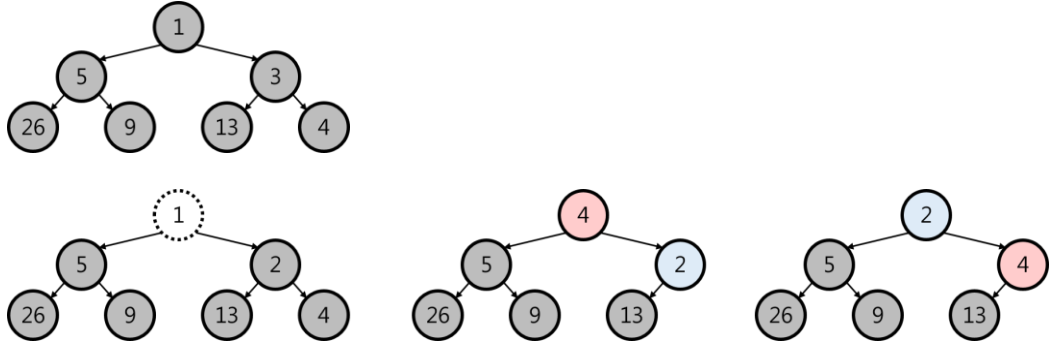
Heap 特性：

1. Heap 適用 Array 存放
2. Heap 的 Insert =>  $O(\log n)$ ，跟高度有關
3. 刪除最大或最小元素 =>  $O(\log n)$
4. 於 Max-Heap(Min-Heap)找最大(小)值，在 Root( $O(1)$ )

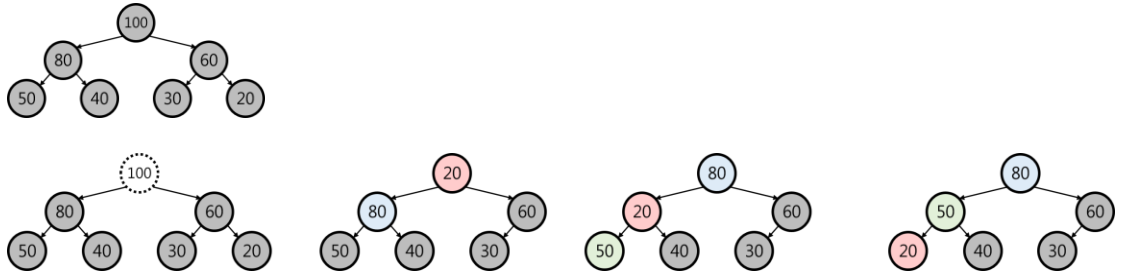
## Heap Delete : Max-Heap(Min-Heap)

1. 移走 Root
2. 將 last Node 取代 Root
3. 調整成 Max-Heap
  - (1) 找出 x 左、右子點最大(小)值
  - (2) Compare(x, k) : if (x < (>) k) : Swap(x, k), goto(1) ; if (x ≥ (≤) k) : 停止

例 1：有一 Max-Heap 如下，求 Delete Min 後為何？



例 2：有一 Max-Heap 如下，求 Delete 100 後為何？



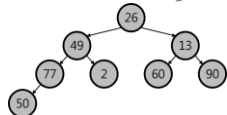
## 二、Bottom-Up 方式：

### Max-Heap(Min-Heap)

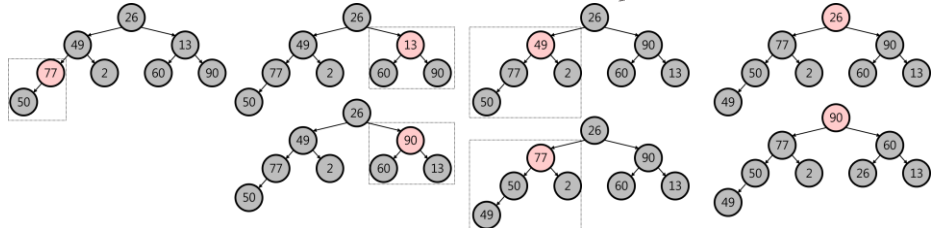
1. Data 先以 Complete BT 呈現(先建，暫不調整)
2. 從最後一個父點，往回到 Root，一一調整子樹成 Heap

例 1：Data : 26, 49, 13, 77, 2, 60, 90, 50。以 Bottom-Up 建立 Max-Heap 之結果為何？

1. Data 先以 Complete BT 呈現(先建，暫不調整)

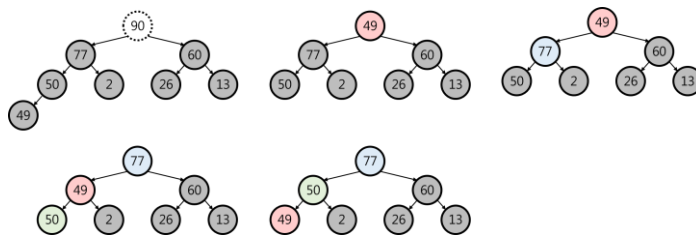


2. 從最後一個父點，往回到 Root，一一調整子樹成 Heap



Note：會跟採 Top-Down 建立的 Heap 不同

例 2：承上，Delete “90”



Heap Sort 就是將 Heap 做  $n-1$  次的 Delete，即可有 Sorting 之效果

演算法：

1. (tree, i, n) => 調整 => 副程式
2. adjust create(tree, n) => 主程式：由下而上、一一呼叫 adjust()  
(反覆繞上來)

PASCAL like: “Max-Heap”

```

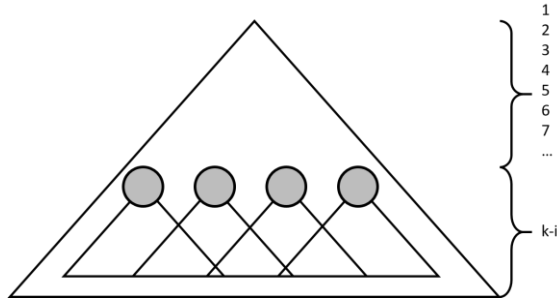
Procedure create(tree, n)
begin
    for i=[n/2] to 1 do
        adjust(tree, i, n);
    end

    Procedure adjust(tree, i, n)
    {
        int j, k;
        Node r;
        bool done;
        done=false; r=tree[i];  k=tree[i].key;
        j=2*i;
        while(j<=n && !done)
        {
            if(j<n)           //成立有右兒子
                if(tree[j].key<tree[j+1].key)
                    j=j+1;
            if(k>=tree[j].key)
                done=true;
            else
            {
                tree[j/2]=tree[j];
                j=2*j;
            }
        }
        tree[j/2]=r;
    }
}
    
```

考題：

1. Heap：Construction、Insert、Delete
2. 演算法：adjust
3. Bottom-Up 為何是  $O(n)$

分析：Heap 以 Bottom-Up 建立之 Time Complexity： $O(n)$



由上圖可知，調整的最大成本為“ $k-i$ ”，Root 在第  $i$  level 之子樹最多有  $(2^{i-1})$  棵，而每棵最大調整成本為  $k-i$ 。

因此，整個調整成本為  $\sum 2^{i-1}(k-i)$ ，最少為 1 即可

可知，令  $s$  為總成本，則

$$s = 2^0(k-1) + 2^1(k-2) + \dots + 2^{k-1-1}(1)$$

$$2s = 2^1(k-1) + 2^2(k-2) + \dots + 2^{k-1}(1)$$

兩式相減可得：

$$s = 2^0(k-1) + 2^1 + 2^2 + 2^3 + \dots + 2^{k-1} = 2^k - k - 1, \text{ 又 Heap 是 Complete BT}$$

$$\text{則 } k = \log(n+1) \Rightarrow O(\log n)$$

$$2^k - k - 1 = 2^{\log n} - \log n - 1 = n - \log n - 1 \Rightarrow O(n)$$