資料結構

C 演算法

二 元 樹

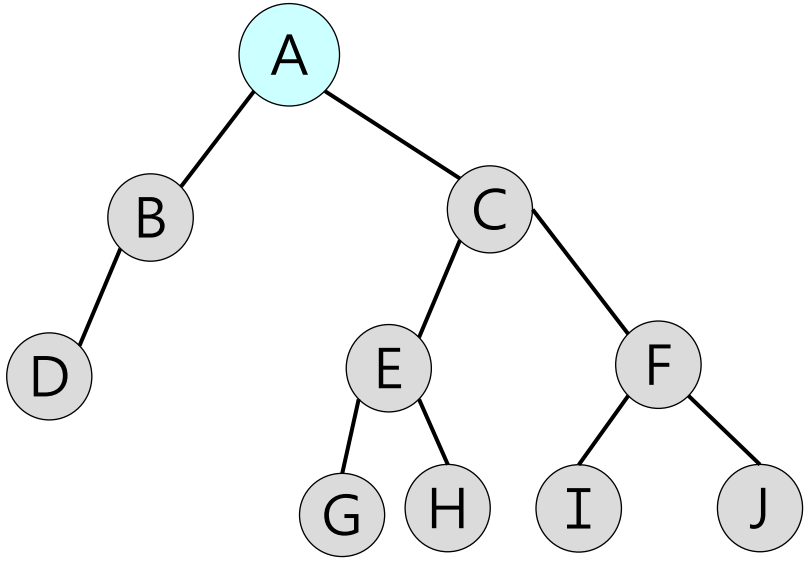
binary Tree

二元樹 Binary Tree

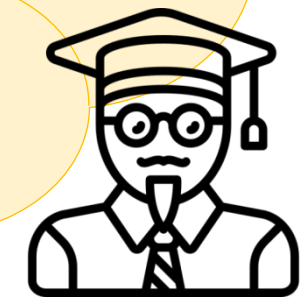
十八豆教育科技

二元樹 binary tree

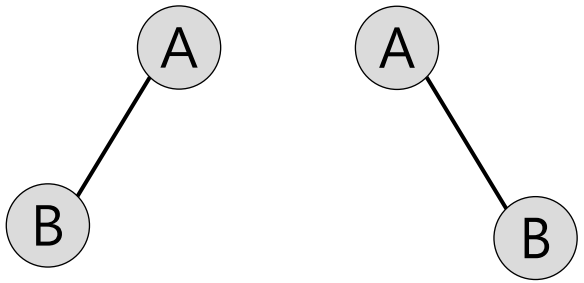
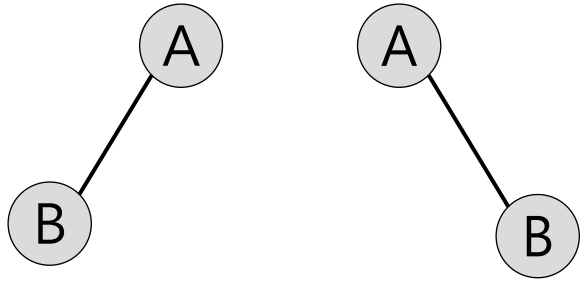
- 樹的任一個節點最多只有 2 個子節點



二元樹是樹型結構內最常出現而且應用最廣泛的一種子集，因此在瞭解基礎的樹概念後，就會將重點放在二元樹！



二元樹 V.S. 樹

	二元樹 Binary Tree	樹 Tree
節點數	可以為空集合	不可為空集合，至少要有 1 個 Root
分支度	分支度 <= 2	沒有限制
方向	子樹有左、右方向的分別	無方向之分
	 不相同的樹	 相同的樹


<二元樹 Binary Tree>
專有名詞

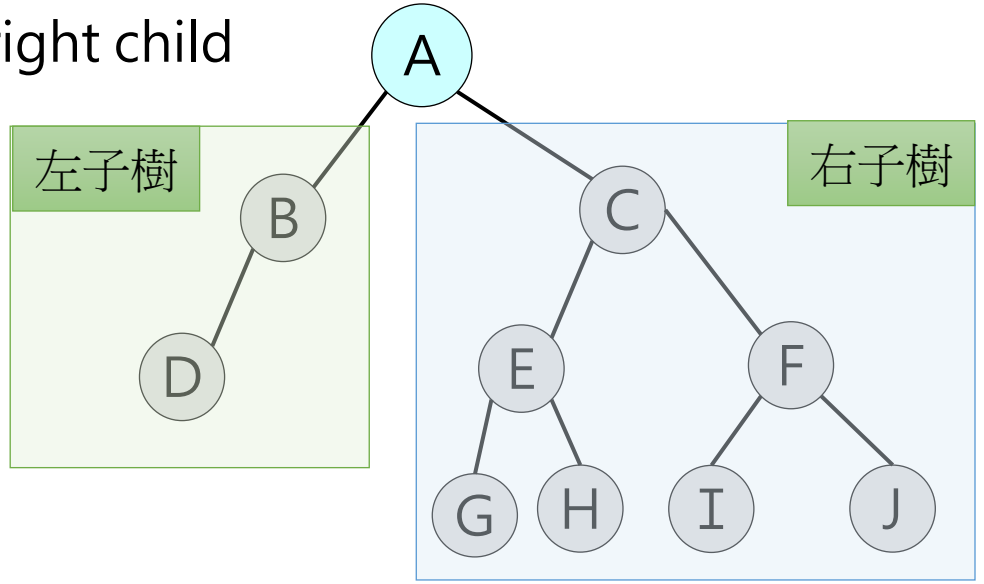
二元樹的專有名詞

左子樹 Left Sub-tree	完滿二元樹 Full Binary Tree
右子樹 Right Sub-tree	完整二元樹 Complete Binary Tree
左歪斜樹 Left-skewed binary tree	嚴格二元樹 strictly binary tree
右歪斜樹 Right-skewed binary tree	

是的，即使二元樹只是數型結構的一個子集合，但二元數仍有一些特殊的專有名詞。

左子樹與右子樹

- 非空集合的二元樹由三個元素組合：
  - 樹根(root)
  - 樹根左邊的子樹，稱為**左子樹 Left Sub-tree**
    - 每個節點的左邊子節點稱為此節點的左兒子 left child
  - 樹根右邊的子樹，稱為**右子樹 Right Sub-tree**
    - 每個節點的右邊子節點稱為此節點的右兒子 right child

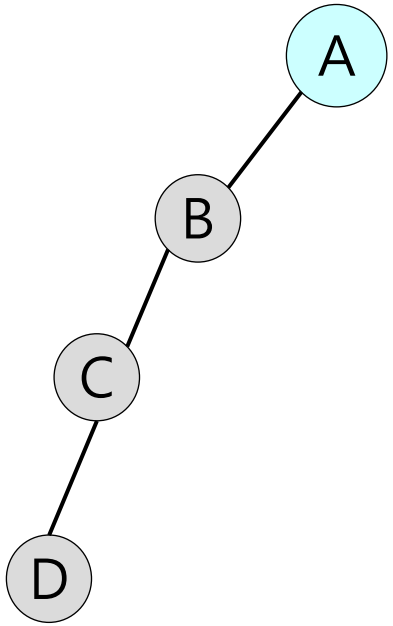


二元樹的專有名詞

歪斜樹

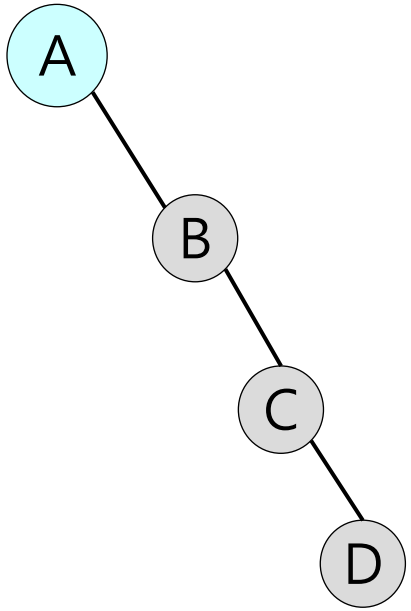
左歪斜樹 Left-skewed binary tree

- 完全沒有右節點的二元樹



右歪斜樹 Right-skewed binary tree

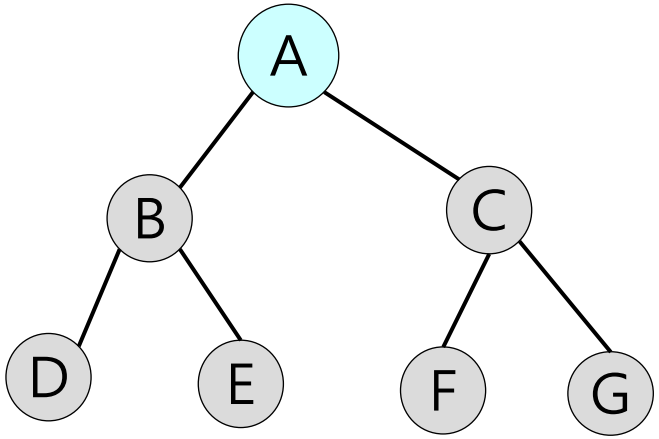
- 完全沒有左節點的二元樹



二元樹的專有名詞

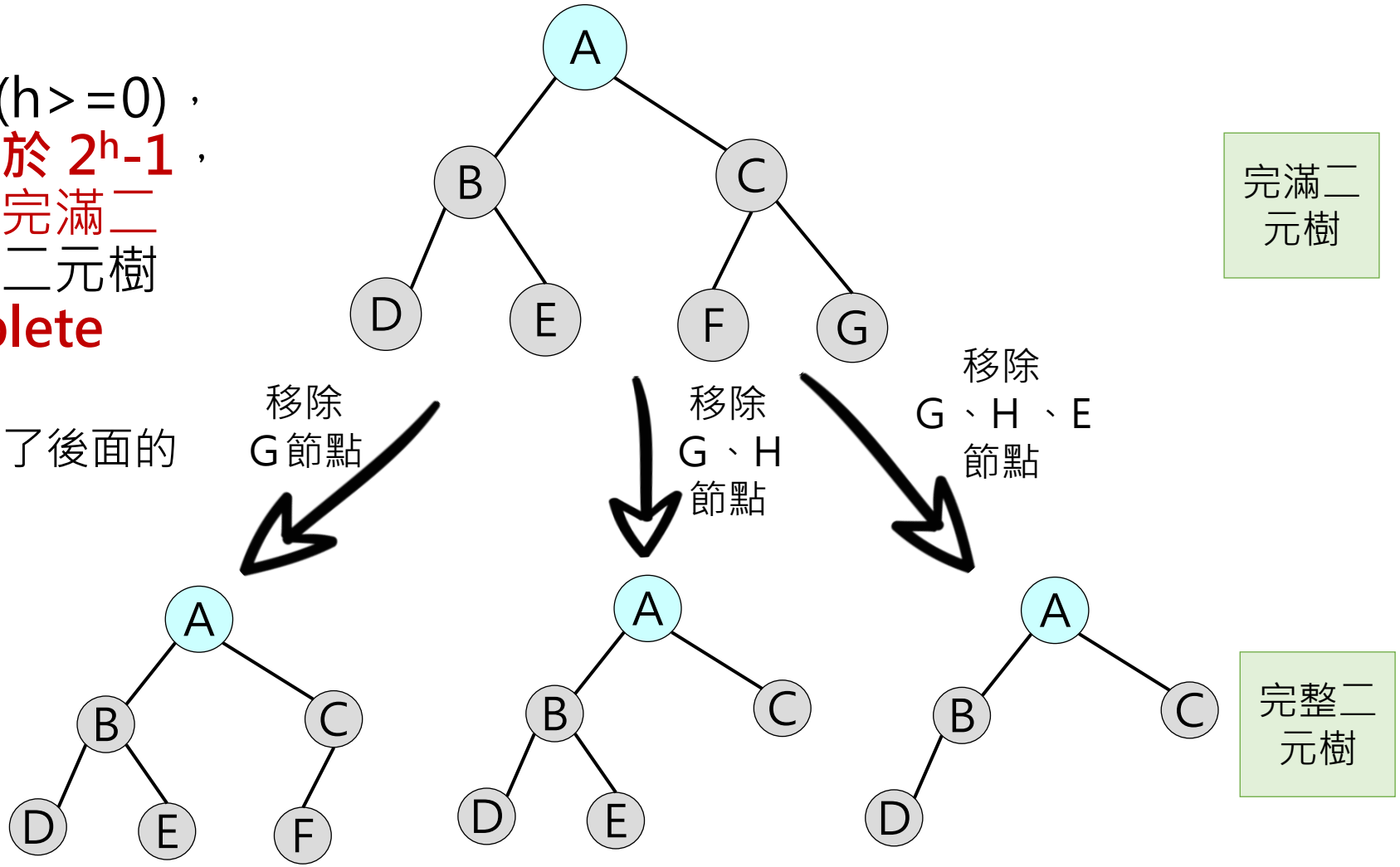
完滿二元樹 Full Binary Tree

- 若二元樹的高度為 h(h>=0)，且二元樹內有 **2^h-1**的節點數時，就稱此二元樹為**完滿二元樹 Full Binary Tree**
  - 也就是除了第 h 層以外的每個節點的分支度都是 2
  - 也可稱為 **完全二元樹**



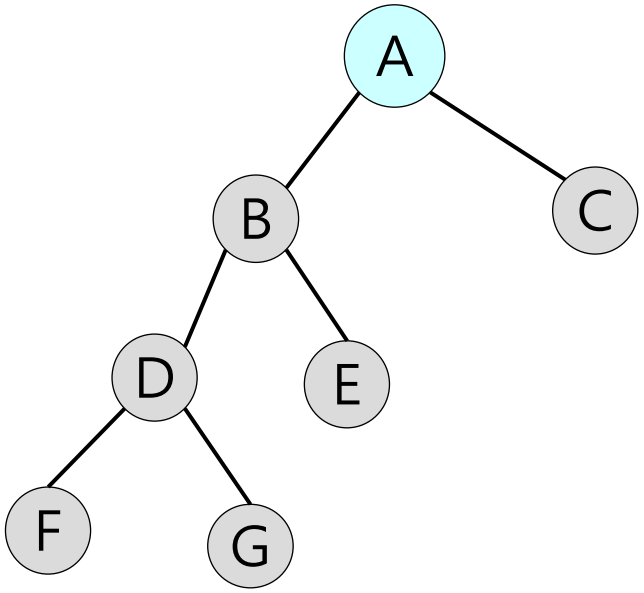
完整二元樹 Complete Binary Tree

- 若二元樹的高度為 h(h>=0)，且二元樹內節點數**少於 2^h-1**，但節點的編號順序與完滿二元樹一樣時，就稱此二元樹為**完整二元樹 Complete Binary Tree**
  - 相當於完整二元樹少了後面的幾個節點



嚴格二元樹 Strictly Binary Tree

- 若二元樹的每個非終端節點都有非空的左右子樹，就稱此二元樹為**嚴格二元樹 Strictly Binary Tree**
  - 也就是說除了葉子節點以外的每個節點的分支度都是 2




<二元樹 Binary Tree>
資料結構表示法

二元樹的表示法

- 陣列表示法
  - 使用一維陣列儲存
  - 二元樹的節點依序編號，每個編號根據公式放在對應的陣列索引內

- 鏈節串列表示法

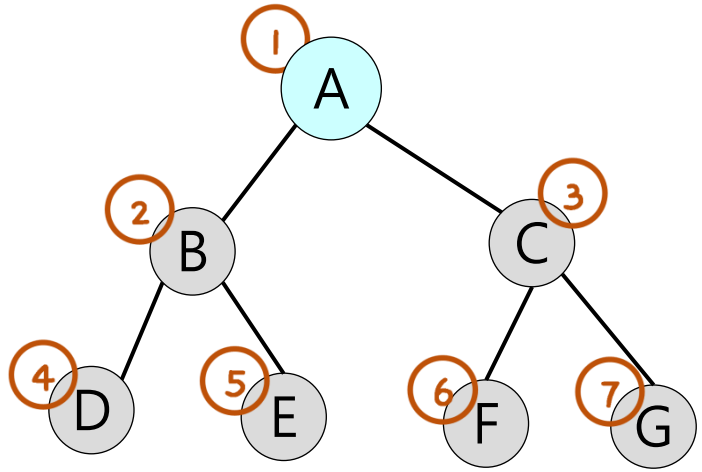
陣列表示法

- 使用**一維陣列**儲存
- 二元樹的節點依序編號，每個編號根據公式放在對應的陣列索引內
  - root 編號為 1，放在陣列索引位置 [0]
  - 若節點編號為 i
    - 節點的左兒子 left child，放在陣列索引位置 **[2*i-1]**
    - 節點的右兒子 left child，放在陣列索引位置 **[2*i]**
    - 若節點 i 不是 root，節點 i 的父節點放在陣列索引位置 **[(i/2)-1]**
      - i / 2 必須取整數商

完整二元樹的陣列表示法

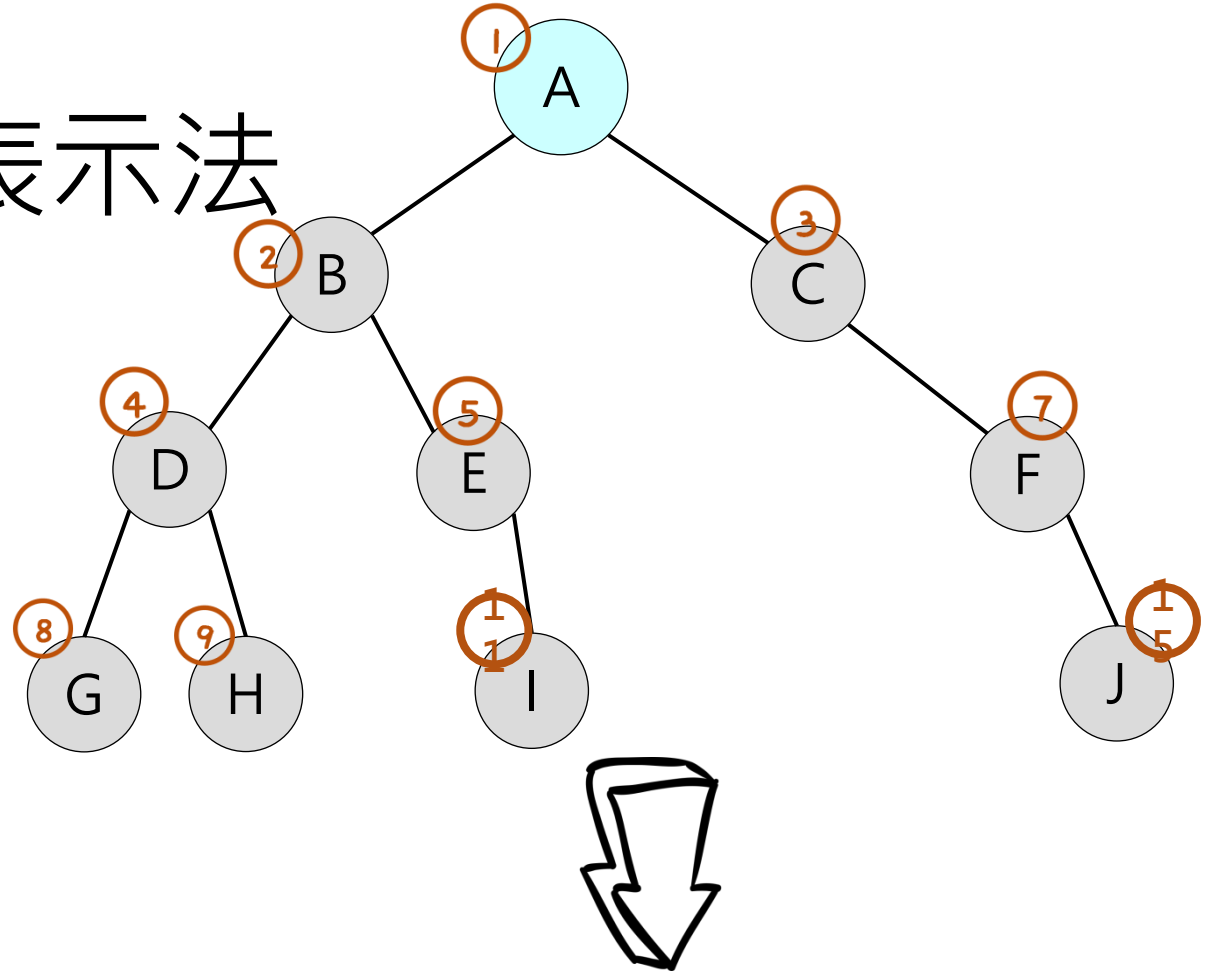
- B 節點編號為 2
  - 左兒子 left child 的索引位置為 [2*2-1]
  - 右兒子 left child 的索引位置為 [2*2]
  - 父節點的索引位置為 [2/2]

- C 節點編號為 3
  - 左兒子 left child 的索引位置為 [3*2-1]
  - 右兒子 left child 的索引位置為 [3*2]
  - 父節點的索引位置為 [(3/2)-1]



陣列索引	[0]	[1]	[2]	[3]	[4]	[5]	[6]
節點	A	B	C	D	E	F	G
節點編號	1	2	3	4	5	6	7
階層	第 1 層	第 2 層		第 3 層			

非完整二元樹的陣列表示法

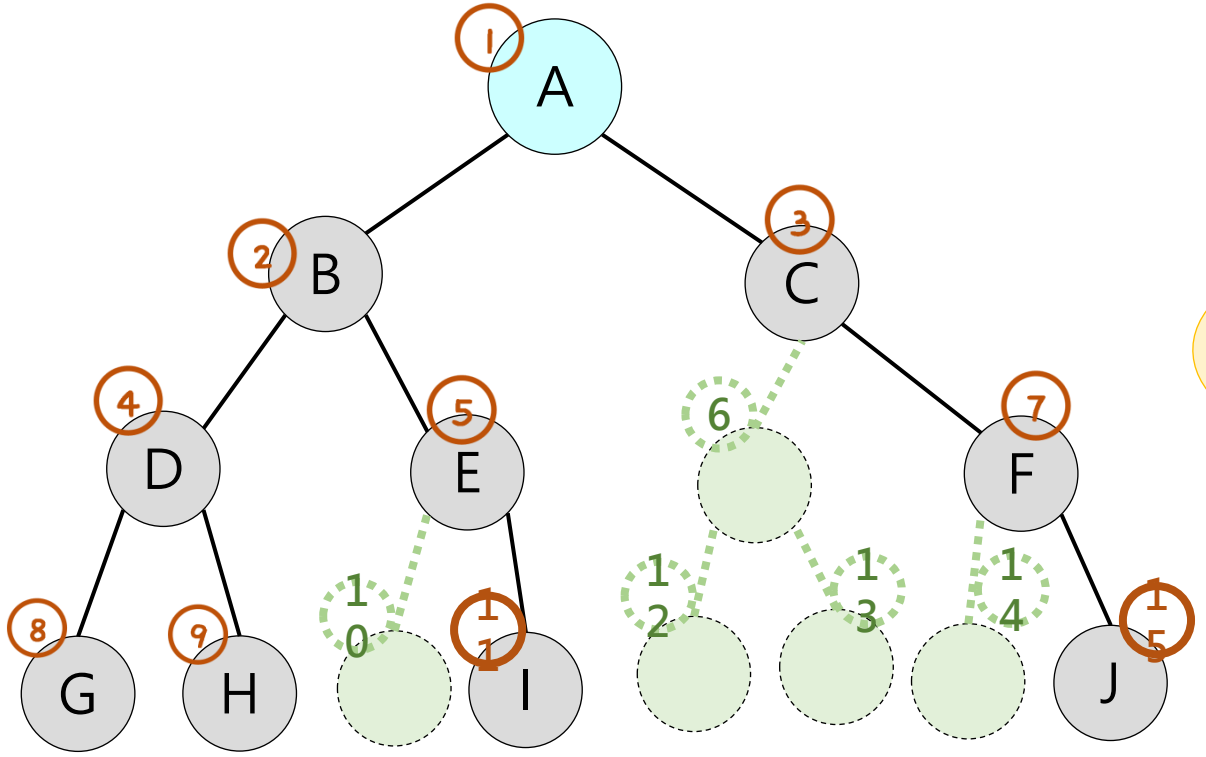


- D 節點編號為 4
  - 左兒子 left child 的索引位置為 [2*4-1]
  - 右兒子 left child 的索引位置為 [2*4]
  - 父節點的索引位置為 [2/2]

- F 節點編號為 7
  - 左兒子 left child 的索引位置為 [2*7-1]
  - 右兒子 left child 的索引位置為 [2*7]
  - 父節點的索引位置為 [(7/2)-1]

陣列索引	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
節點	A	B	C	D	E		F	G	H		I				J
節點編號	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
階層	第 1 層	第 2 層		第 3 層				第 4 層							

非完整二元樹的陣列表示法



當遇到非完整二元樹時，即使節點不存在，仍然會保留對應的陣列索引位置，也就會造成閒置的空間配置。

陣列索引	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
節點	A	B	C	D	E		F	G	H		I				J
節點編號	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
階層	第 1 層	第 2 層		第 3 層				第 4 層							

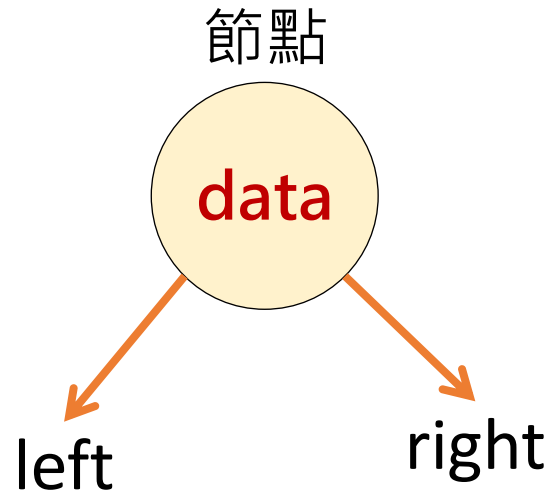
陣列表示法

- 優點：
  - 陣列表示法最適合用來儲存完整二元樹，不會浪費空間，存取也最為方便。
  - 可快速找尋左右子節點與父節點。

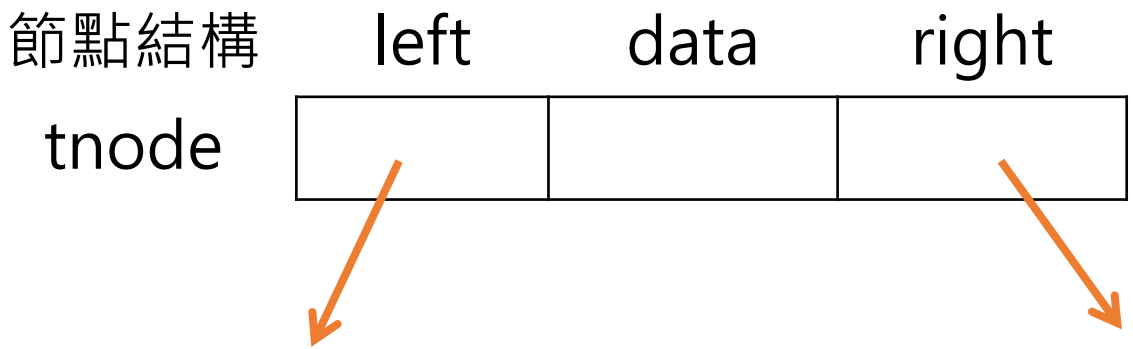
- 缺點：
  - 若是非完整二元樹，尤其是歪斜樹時，就會浪費大量空間。
  - 若節點有新增或刪除時也需要大量移動資料。

鏈節串列表示法

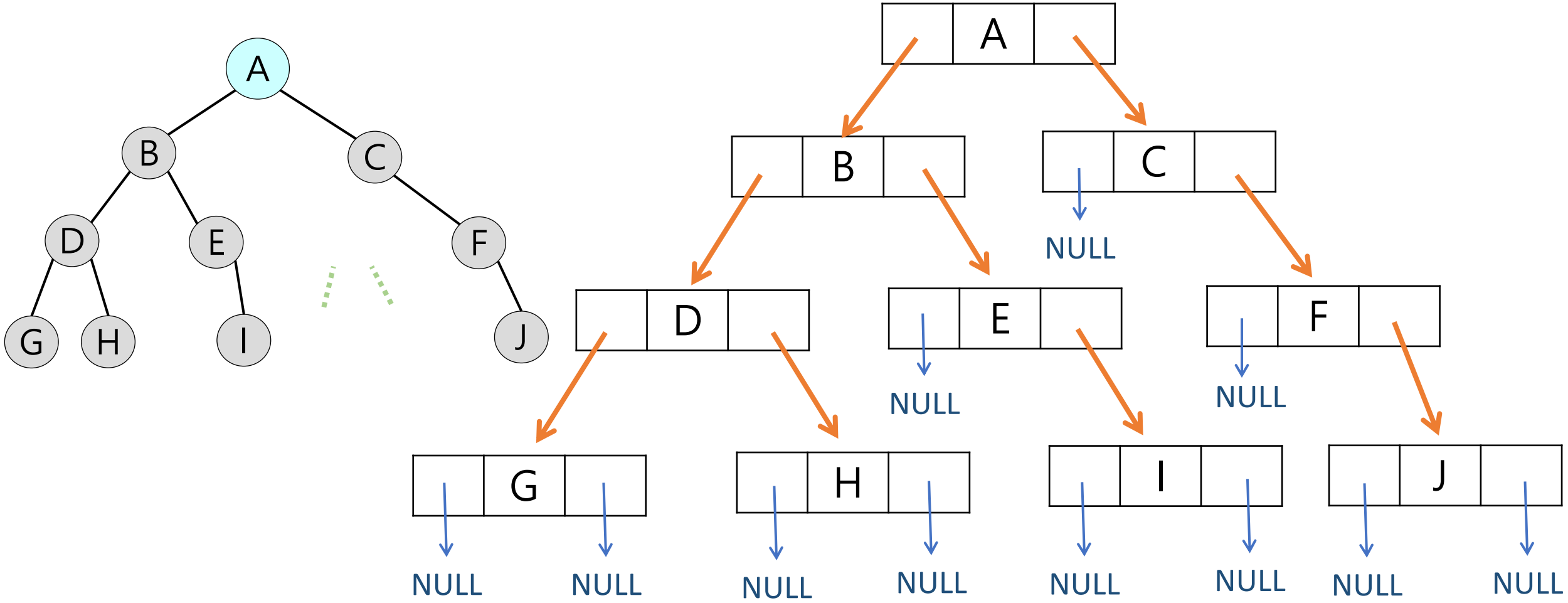
- 鏈節串列是用一個節點結構表示樹上的每個節點，結構內除了儲存該節點本身的值，另外有**兩個指標 left、right**，分別指向節點的左兒子 left child 與右兒子 right child。




```
typedef struct node {
  int data;
  struct node *left;
  struct node *right;
} tnode;
```



鏈節串列表示法



鏈節串列表示法

- 優點
  - 節點新增或刪除容易操作

- 缺點：
  - 很難找父節點
  - 容易浪費空間：
    - 非葉子節點，若缺少左兒子或右兒子，就會浪費一個 link 空間
    - 葉子節點會浪費兩個指向 NULL 的 left 與 right 指標


<二元樹 Binary Tree>
二元樹的走訪

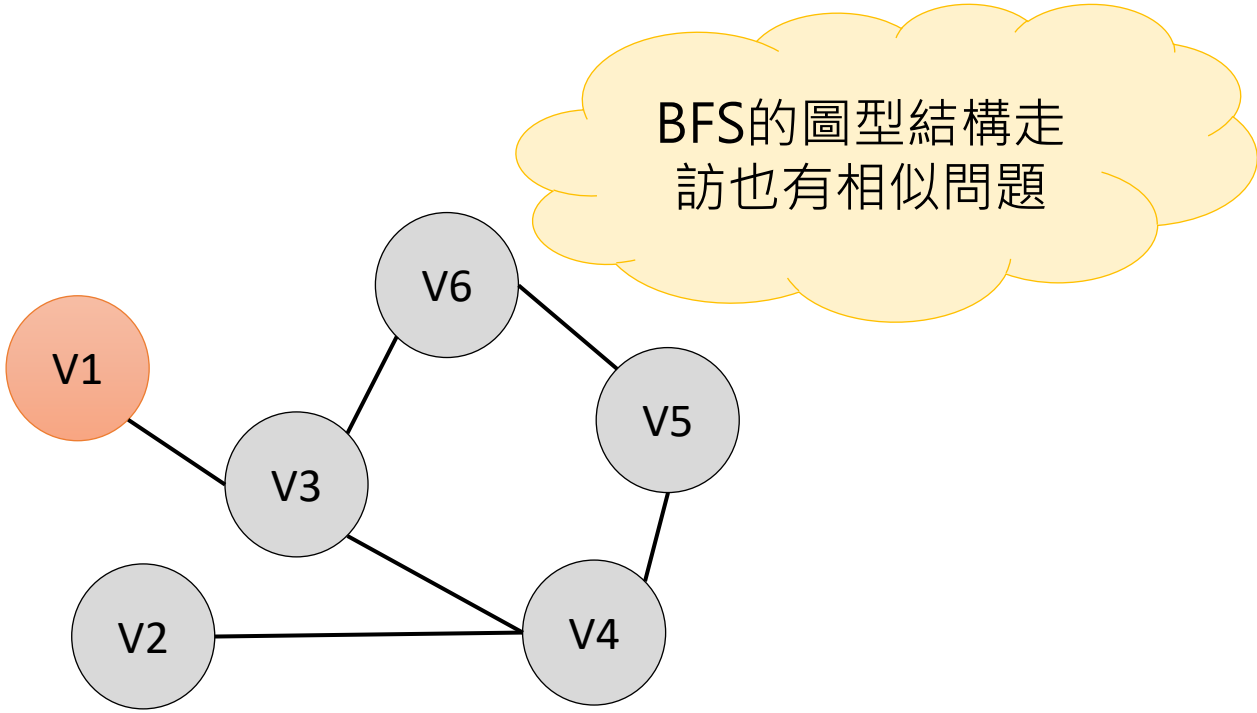
回想圖型結構的走訪 - DFS

- 圖型結構使用 DFS 走訪時，依照走訪順序不同，可能有多種答案
  - 每次都選編號比較小的先拜訪

V1 → V3 → V4 → V2 → V5 → V6

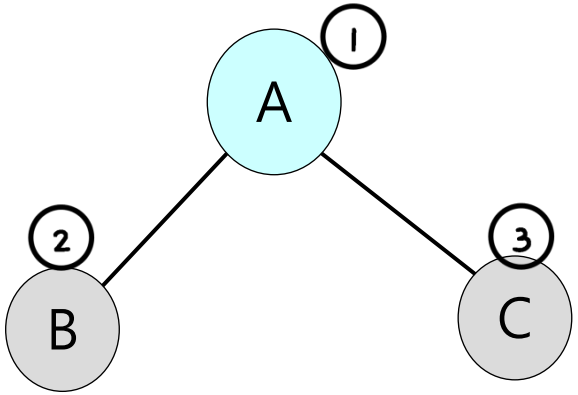
  - 每次都選編號比較大的先拜訪

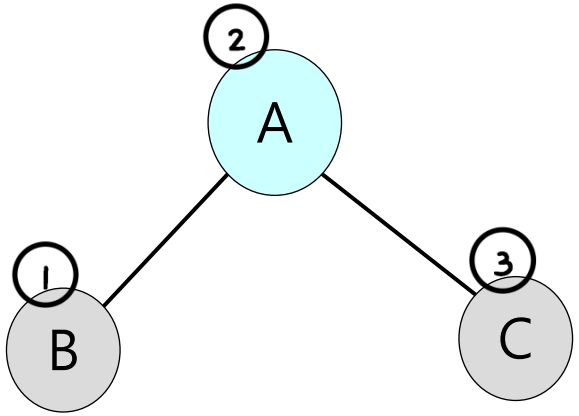
V1 → V3 → V6 → V5 → V4 → V2

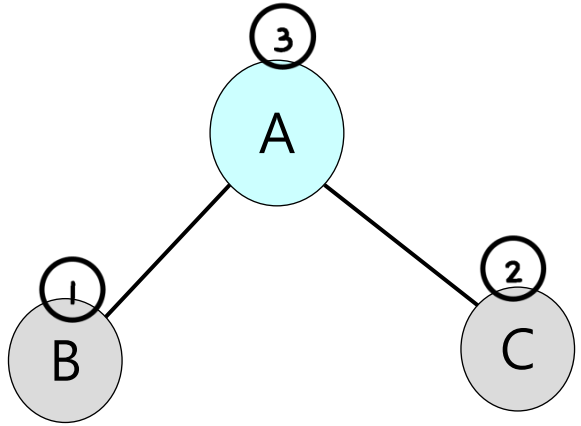


樹狀結構的走訪也有同樣問題

- 應該先拜訪樹根？
- 還是先拜訪左子樹？
- 還是先拜訪右子樹？







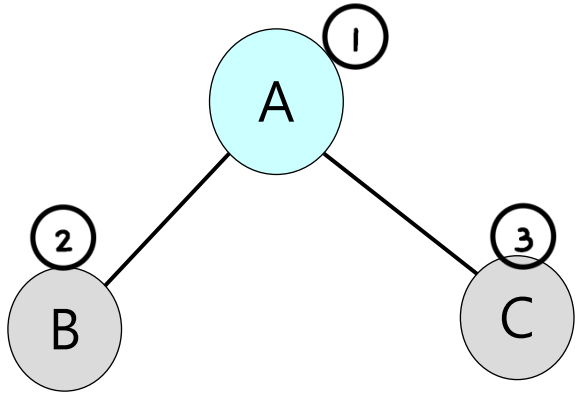
根據順序，樹狀結構有三種走訪方式

- 前序走訪 Preorder Traverse
- 中序走訪 Inorder Traverse
- 後序走訪 Postorder Traverse

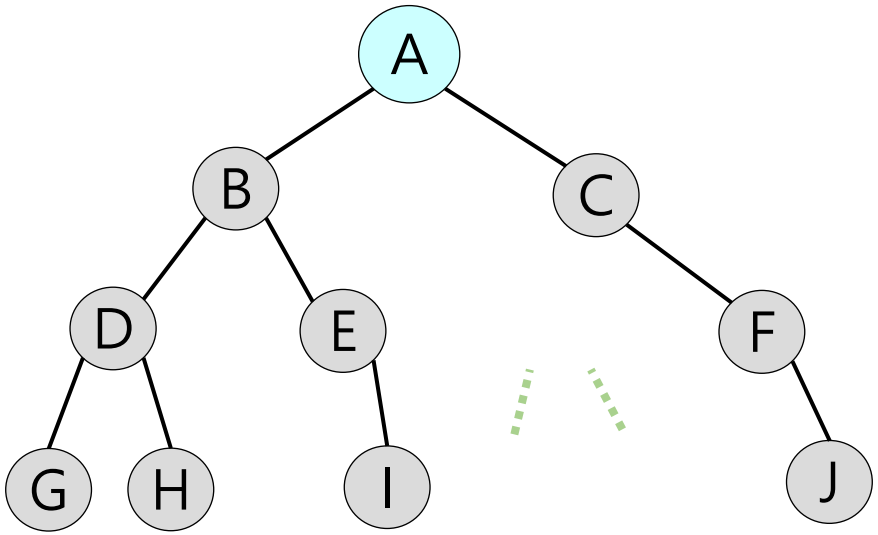
前序走訪 Preorder Traverse

前序走訪 Preorder Traverse

走訪順序：A B C



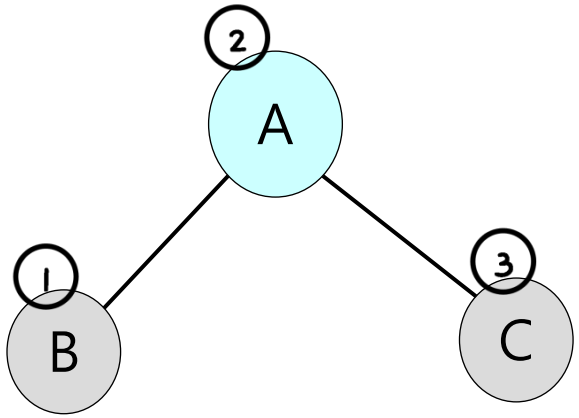
走訪順序：A B D G H E I C F J



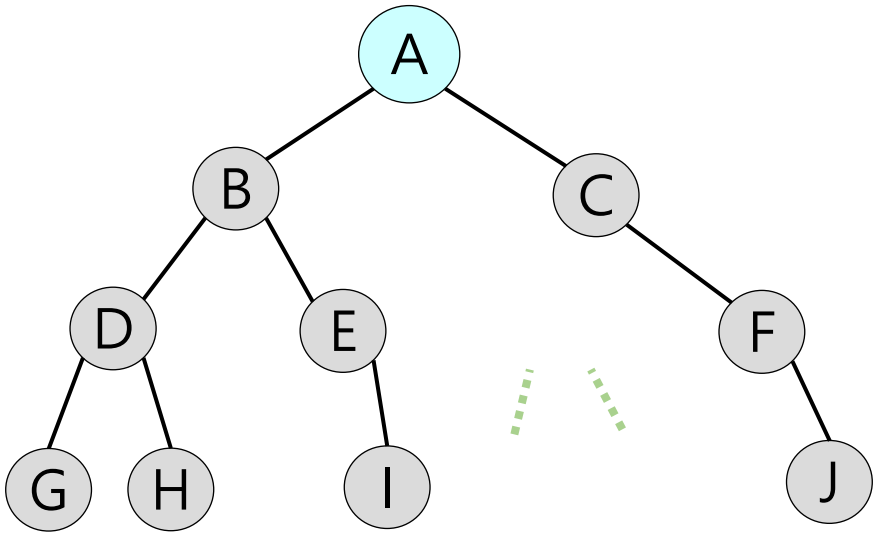
中序走訪 Inorder Traverse

中序走訪 Inorder Traverse

走訪順序：B A C



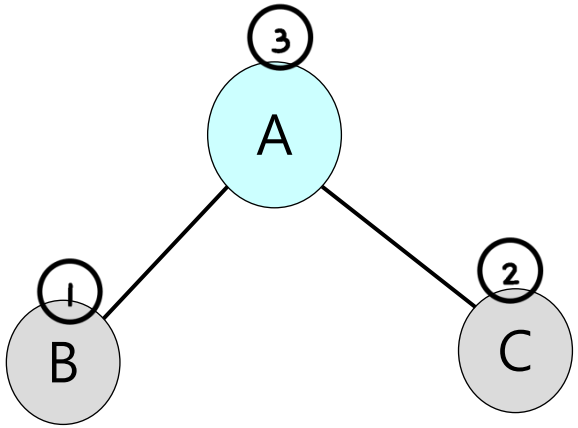
走訪順序：G D H B E I A C F J



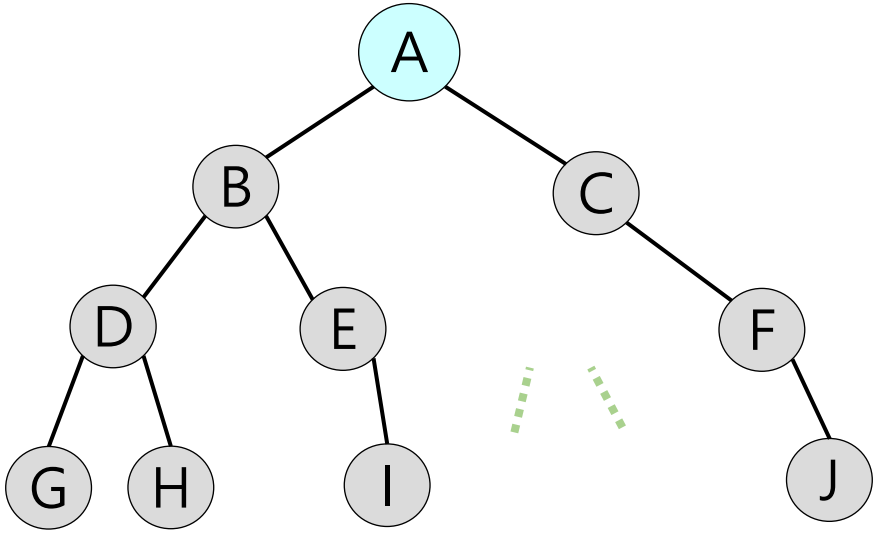
後序走訪 Postorder Traverse

後序走訪 Postorder Traverse

走訪順序：B C A



走訪順序：G H D I E B J F C A

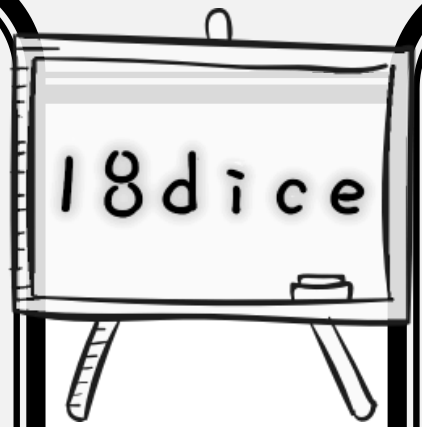


二元樹的走訪

三種走訪方式都是使用**遞迴**實作的喔！

```
// 中序走訪: 左子樹, 目前節點, 右子樹
void inorder(struct node root)
{
  if (root){
    postorder(root->left);
    printf("%d", root->data);
    postorder(root->right);
  }
}
```

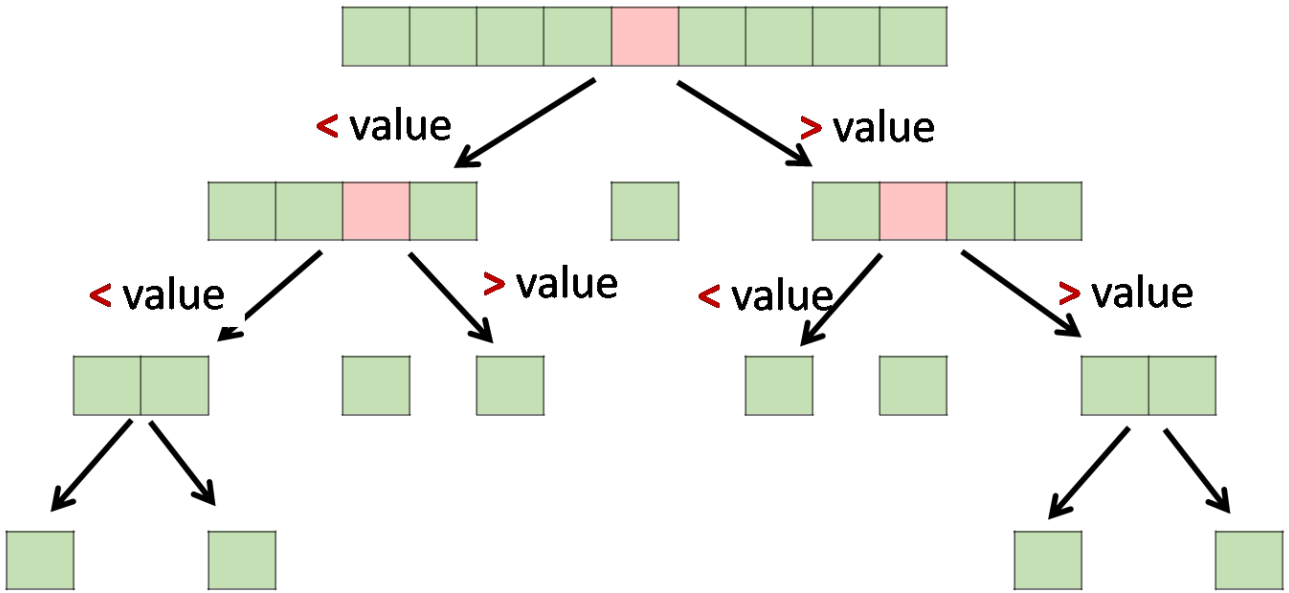
```
// 後序走訪: 左子樹, 右子樹, 目前節點
void postorder(struct node root)
{
  if (root){
    postorder(root->left);
    postorder(root->right);
    printf("%d", root->data);
  }
}
```

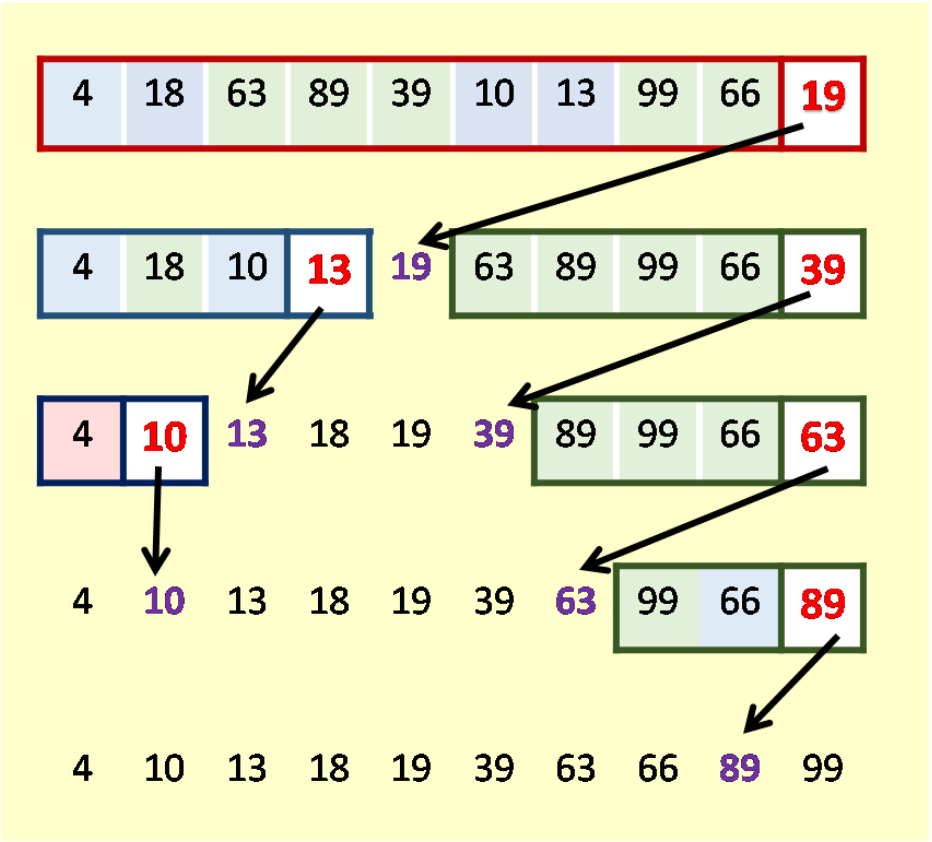



```
// 前序走訪: 目前節點, 左子樹, 右子樹
void preorder(struct node root)
{
  if (root){
    printf("%d", root->data);
    postorder(root->left);
    postorder(root->right);
  }
}
```


<二元樹 Binary Tree>
二元搜尋樹

知道這兩張圖是代表什麼演算法嗎？



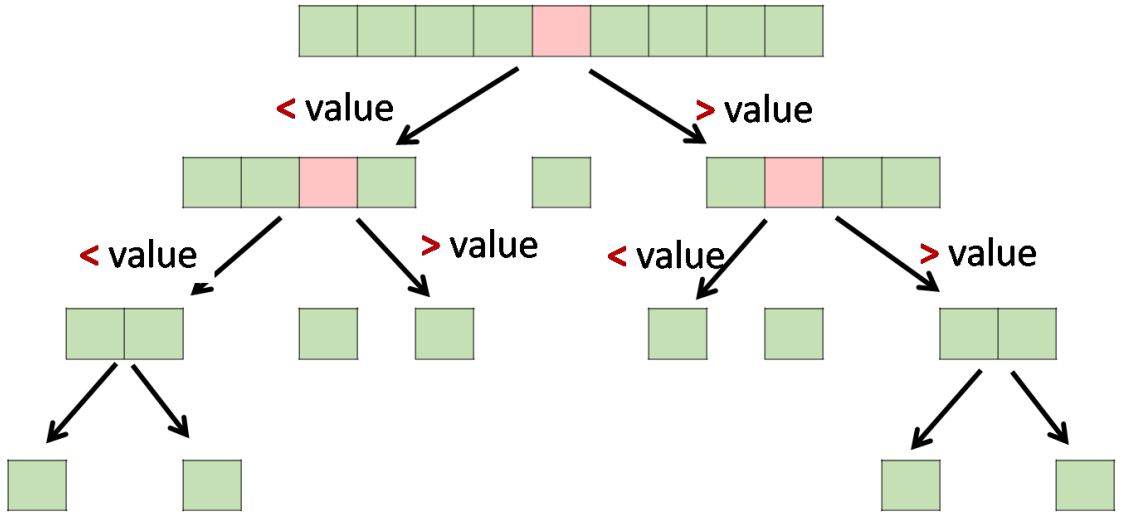


快速排序法 quick sort

- 快速排序法透過找尋中間值的方式，每經過一回合就會將資料分成兩群，左邊群內的值衡小於右邊群內的值。

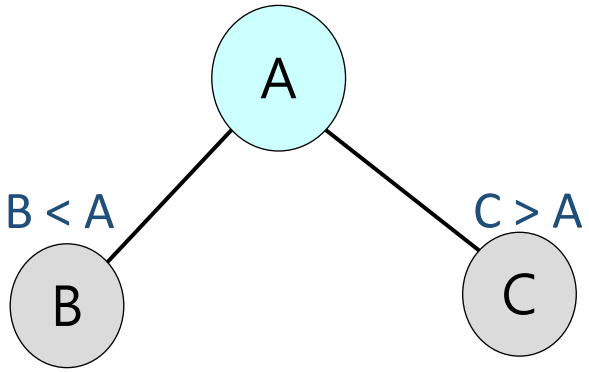
- 如果將中間值視為樹根節點，那 quick sort 每回合的變化圖是否就很像樹狀結構呢？

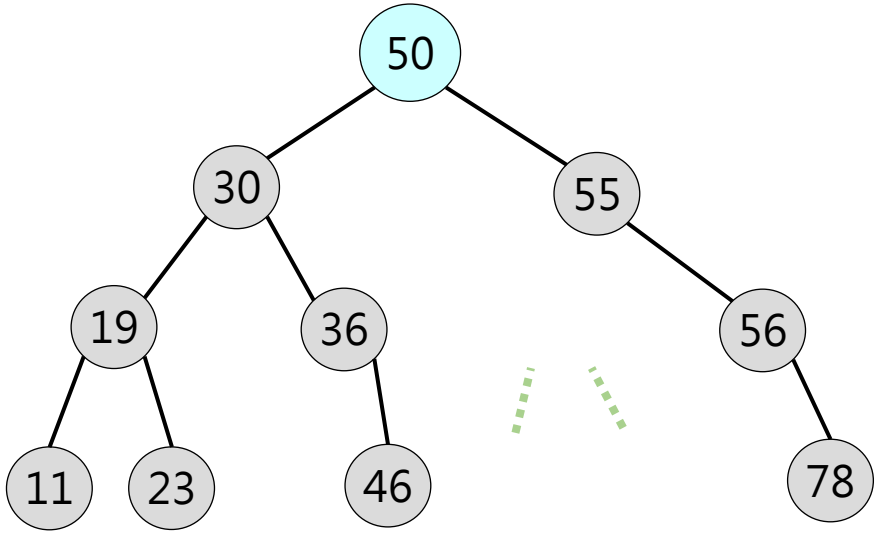
- 只要加點限制，二元樹就會是處理排序與搜尋演算法的重要資料結構。



二元搜尋樹 Binary Search Tree

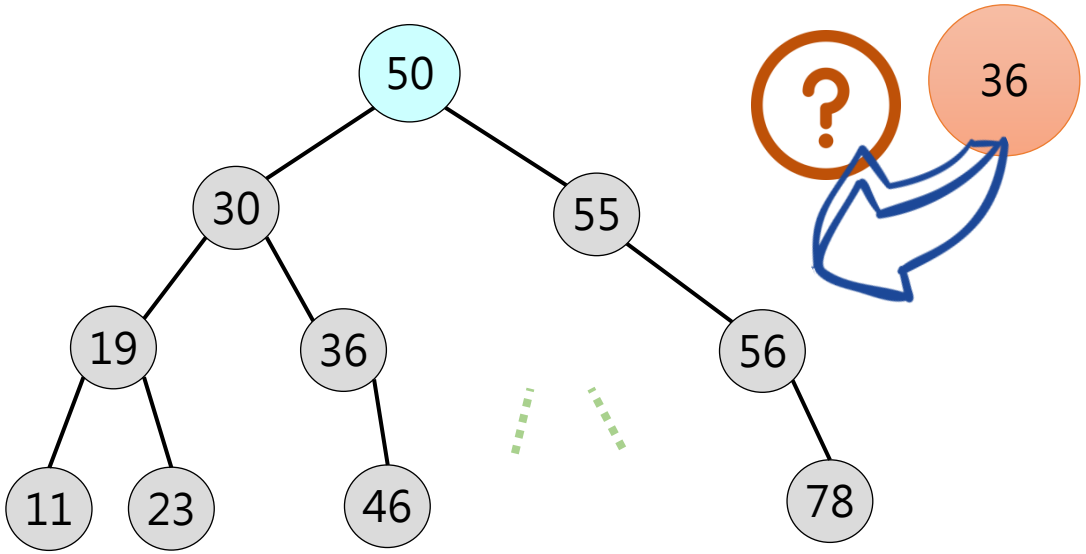
- 二元搜尋樹(BST)是符合以下特性的二元樹：
  - 每個節點都有一個鍵值，而且每一個節點的鍵值都不相同。
  - 非空的左子樹內的每個節點的鍵值都要小於 root 的鍵值。
  - 非空的右子樹內的每個節點的鍵值都要大於 root 的鍵值。
  - 左子樹和右子樹也都是二元搜尋樹。



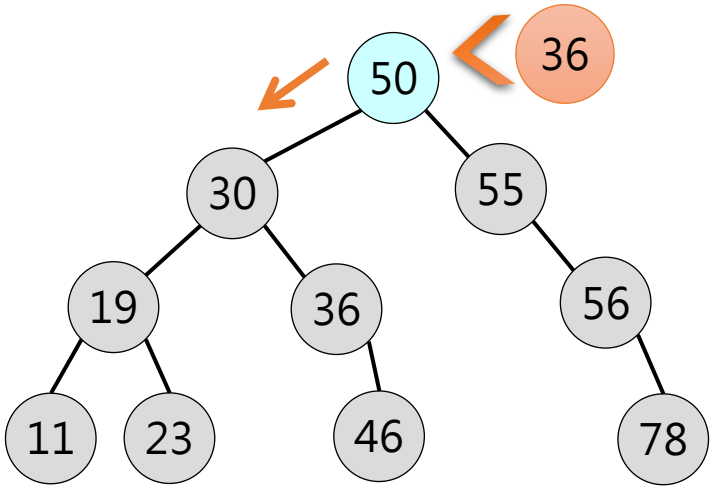


節點搜尋 (1)

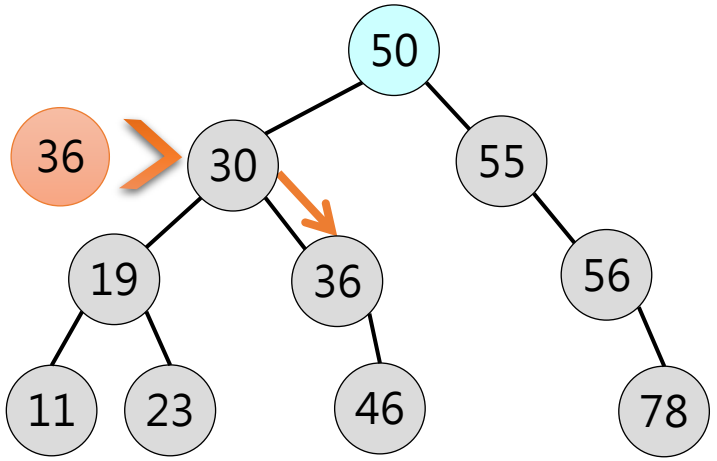
- BST 的節點搜尋相當簡單，只要從樹根節點開始，若搜尋值比樹根節點的值大就往右子樹找，若比樹根節點的值小就往左子樹找，持續的往子節點搜尋，直到符合的搜尋值或是已經沒有子節點為止。



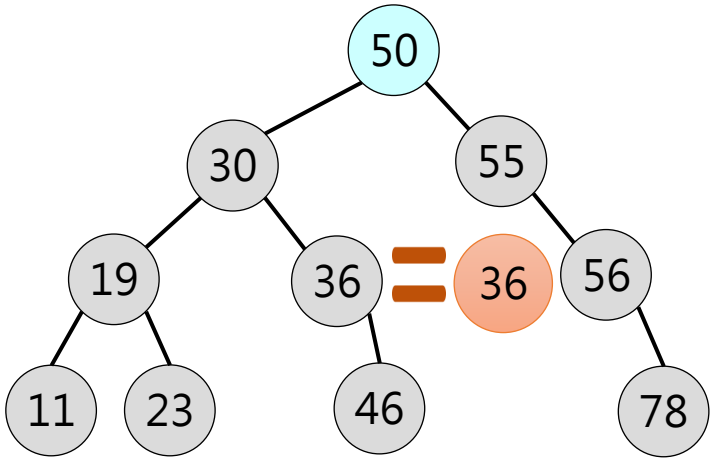
節點搜尋 (2)



36 < 50，往左子樹找



36 > 30，往右子樹找



36 =36，找到了！

節點搜尋 (3)

節點的搜尋也是使用**遞迴**實作的喔！

```
struct node* search(struct node *root, int x)
{
  if (root == NULL) {
    return NULL;
  } else if (root->data==x) {
    return root;
  } else if(x > root->data) {
    return search(root->right, x);
  } else {
    return search(root->left, x);
  }
}
```

節點是空的，代表搜尋失敗，就不再繼續搜尋。

節點的值與搜尋的值相等，就回傳這個符合搜尋值的節點。

當下這個節點的值與搜尋的值不相等，且搜尋的值大於此節點的值，就往**右子樹**繼續尋找（右子樹內的所有節點值都會小於目前這個節點的值）。

當下這個節點的值與搜尋的值不相等，且搜尋的值小於此節點的值，就往**左子樹**繼續尋找（左子樹內的所有節點值都會大於目前這個節點的值）。

新增節點 (1)

- 由於二元搜尋樹的特色就是左子樹的值會小於樹根節點值，右子樹的值會大於樹根節點值，因此新增節點時，只要判斷新增的值與樹根節點值的大小比較，就可找出要新增的位置：
  - 新增節點值大於樹根節點值，就加到右子樹，
  - 新增節點值小於樹根節點值，就加到左子樹。
- 若要加入右子樹且右兒子已經有節點時，就用**遞迴**的方式，往柚子樹尋找，再度判斷新增的節點應該加入右子樹的哪個位置。
- 同樣的，若是要加入左子樹，也是使用相同的遞迴方式處理。

新增節點 (2)

新增節點在找尋要放置的位置時也是使用**遞迴**實作的喔！

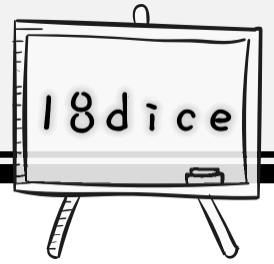
找到可以新增的空位置，就配置一個新節點儲存，並將新節點回傳給上一層呼叫者，讓上一層節點的左(或右)節點可以指向新節點。

```
struct node* insert(struct node *root, int x)
{
  if (root==NULL)
    return new_node(x);
  else if(x > root->data)
    root->right = insert(root->right, x);
  else
    root->left = insert(root->left, x);
  return root;
}
```

```
struct node* new_node(int x)
{
  struct node *p;
  p = malloc(sizeof(struct node));
  p->data = x;
  p->left = NULL;
  p->right = NULL;
  return p;
}
```

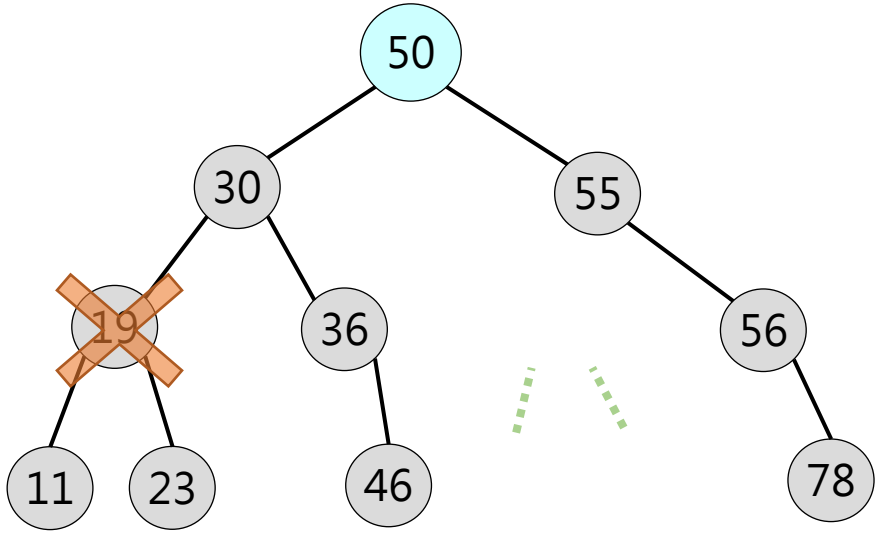
若已存在的值大於要新增的值，就往左子樹找出符合條件的空位置。

要新增的位置已經有節點時，就判斷已存在的值是否是小於要新增的值，若是就繼續往右子樹找出符合條件的空位置。



刪除節點

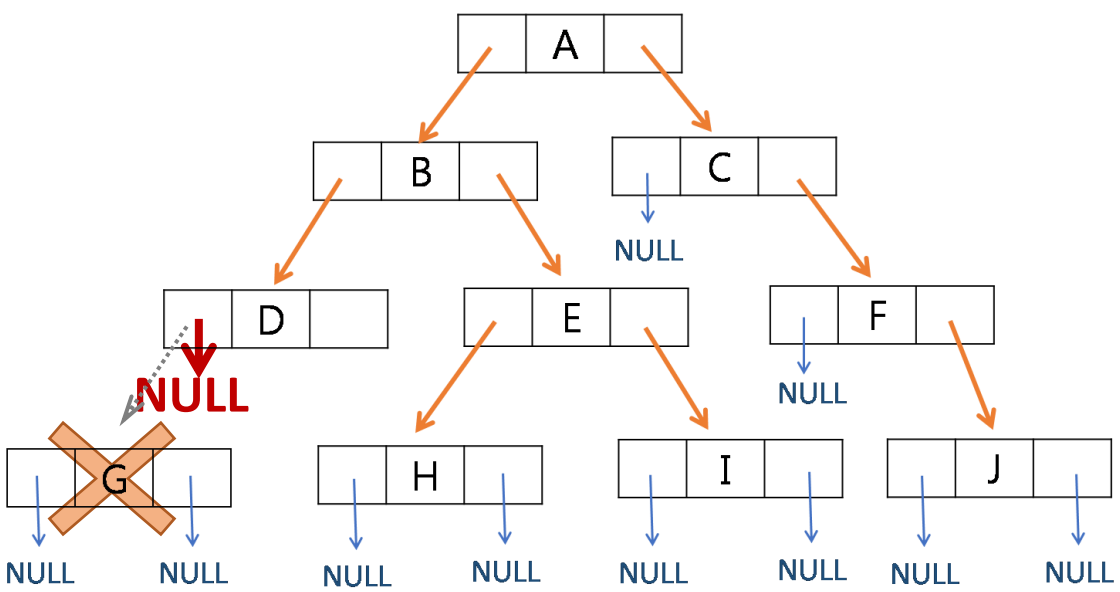
- 刪除節點時，若刪除的不是葉子節點時，將節點移除會造成該節點的左子樹與右子樹的節點也被迫從原樹中消失，因此刪除節點時，需要先考慮刪除的節點位置，再進行對應的處理。



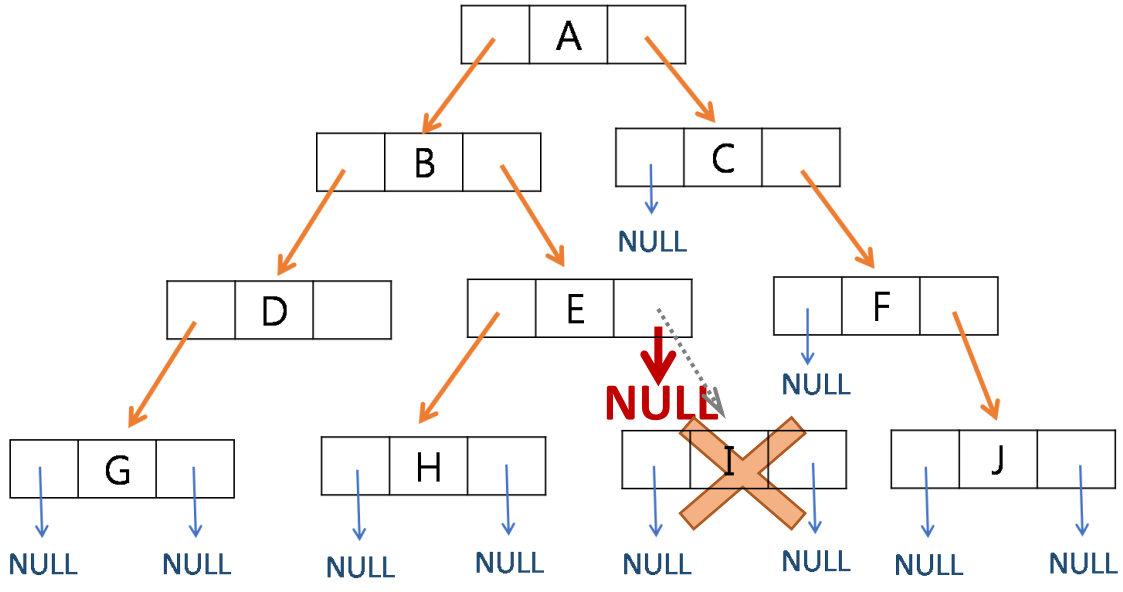
若直接將19移除，那左子樹的11與右子樹的23也會被迫消失（走訪原樹時，沒有可通行的路徑到達）

刪除節點 – 刪除葉子節點

- 如果刪除的是葉子節點，可以直接將節點刪除，父節點原本指向此節點的鏈結指向 NULL 就可以了。



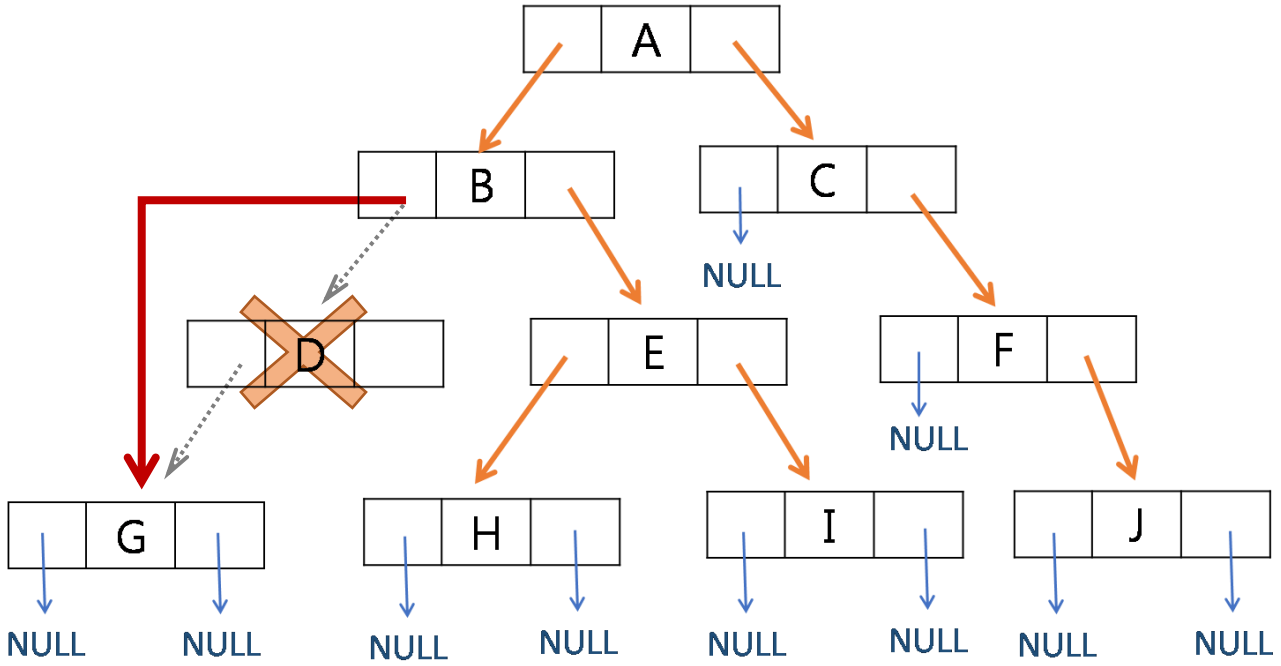
- 刪除 G 節點
直接刪除，再將 D 節點的 left link 指向 NULL



- 刪除 I 節點
直接刪除，再將 E 節點的 right link 指向 NULL

刪除節點 – 刪除有左子樹的節點

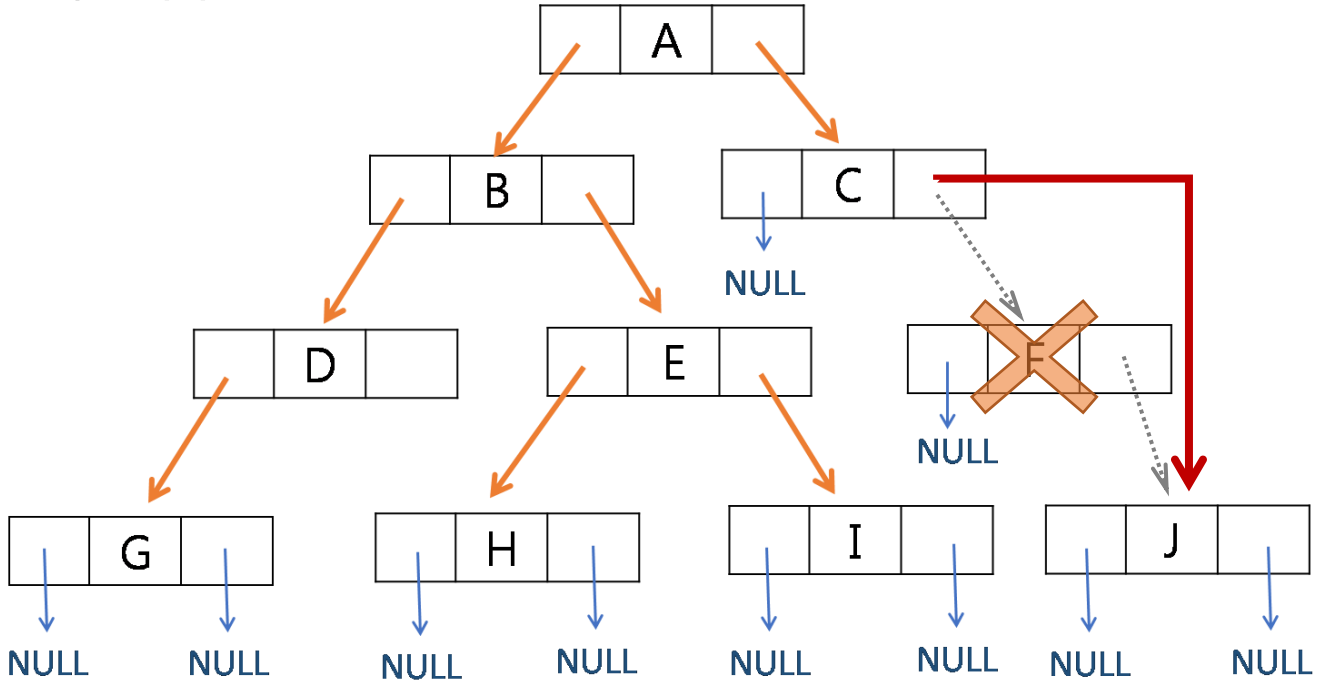
- 如果刪除的節點還有左子樹的時後，就將父節點原本指向此節點的鏈節改成指向左子樹。



• 刪除 D 節點
先將 B 節點的 left link 指向 G 節點(D節點的 left link)，
才能將 D 節點刪除

刪除節點 – 刪除有右子樹的節點

- 如果刪除的節點還有右子樹的時後，就將父節點原本指向此節點的鏈節改成指向右子樹。



- 刪除 F 節點

先將 C 節點的 right link 指向 J 節點(F節點的 right link)，
才能將 F 節點刪除

刪除節點 – 刪除有左右子樹的節點 (1)

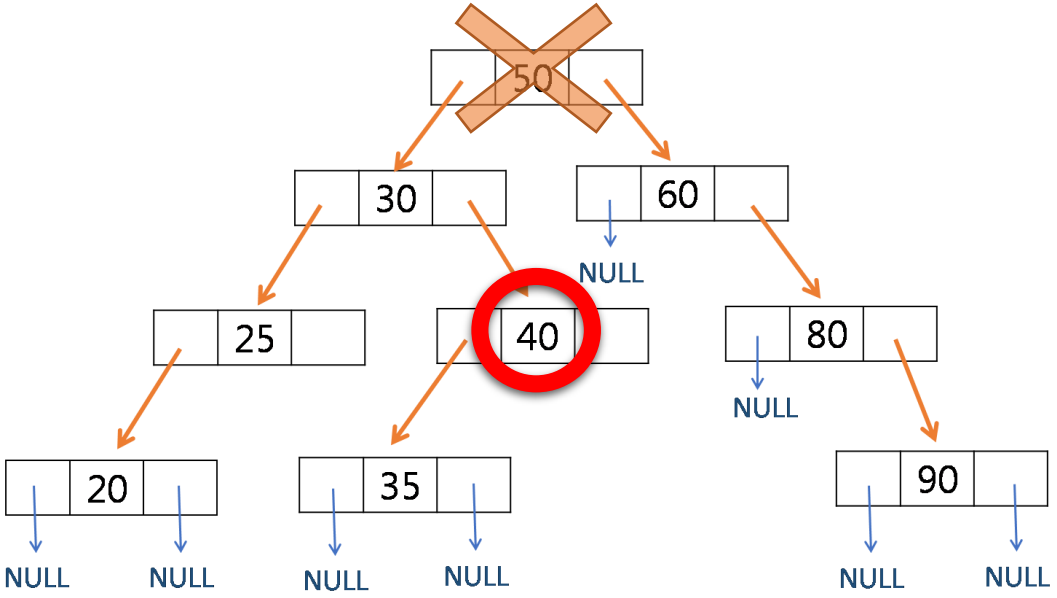
- 還有左右子樹的節點一旦刪除，會同時影響兩邊子樹的所有節點，會讓影響範圍太大，為了降低影響度，會改找**取代點**進行刪除。

- 取代點：用交換的方式，改用比較容易刪除的節點做為取代點進行刪除，取代點內的值放到原本要被刪除的節點內。
  - 第一種取代點：在左子樹找出最大值
  - 第二種取代點：在右子樹找出最小值

刪除節點 – 刪除有左右子樹的節點 (2)

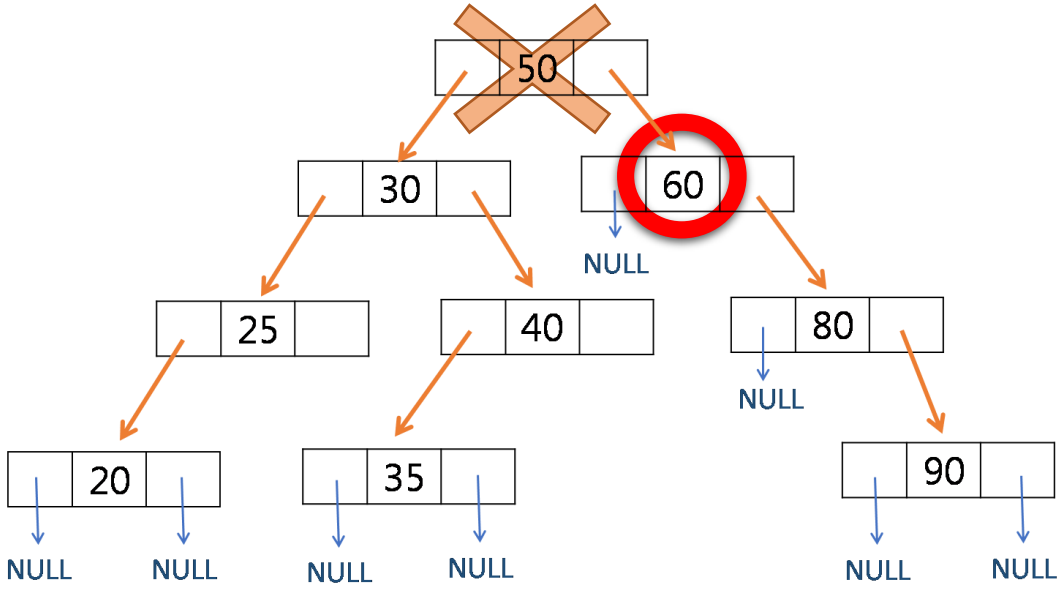
第一種取代點:在左子樹找出最大值

使用 40 做為取代點

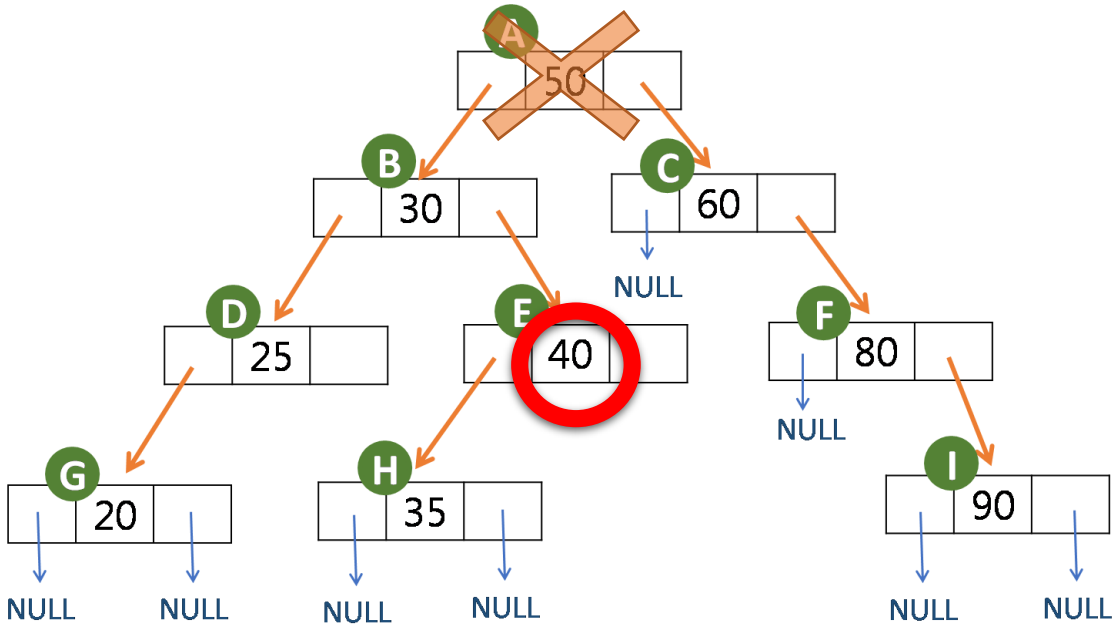


第二種取代點:在右子樹找出最大值

使用 60 做為取代點

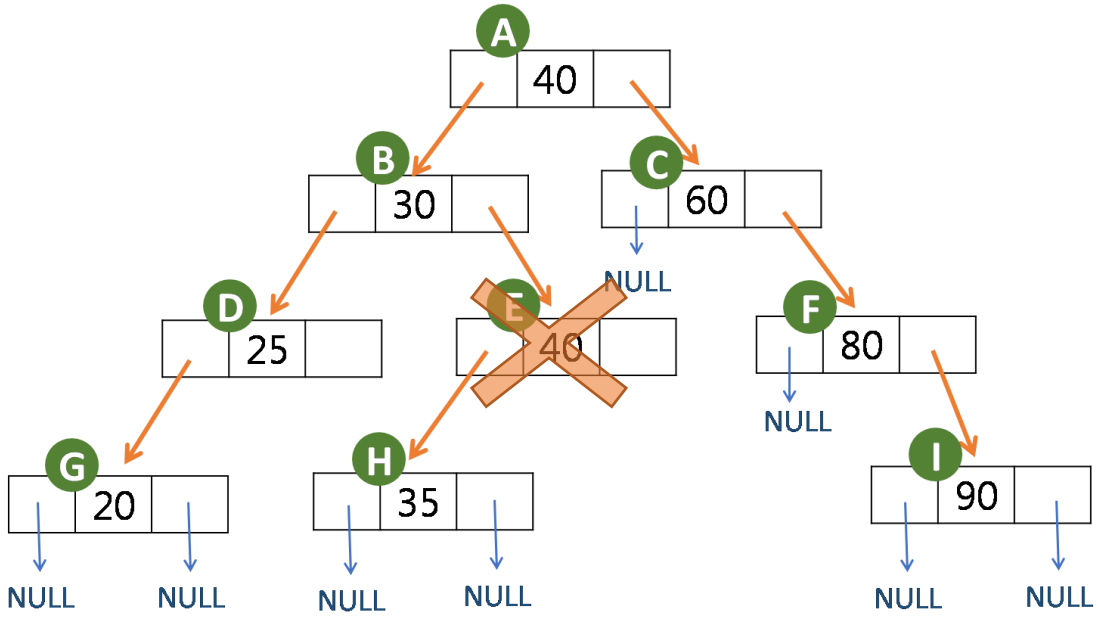


刪除節點 – 刪除有左右子樹的節點 (3)



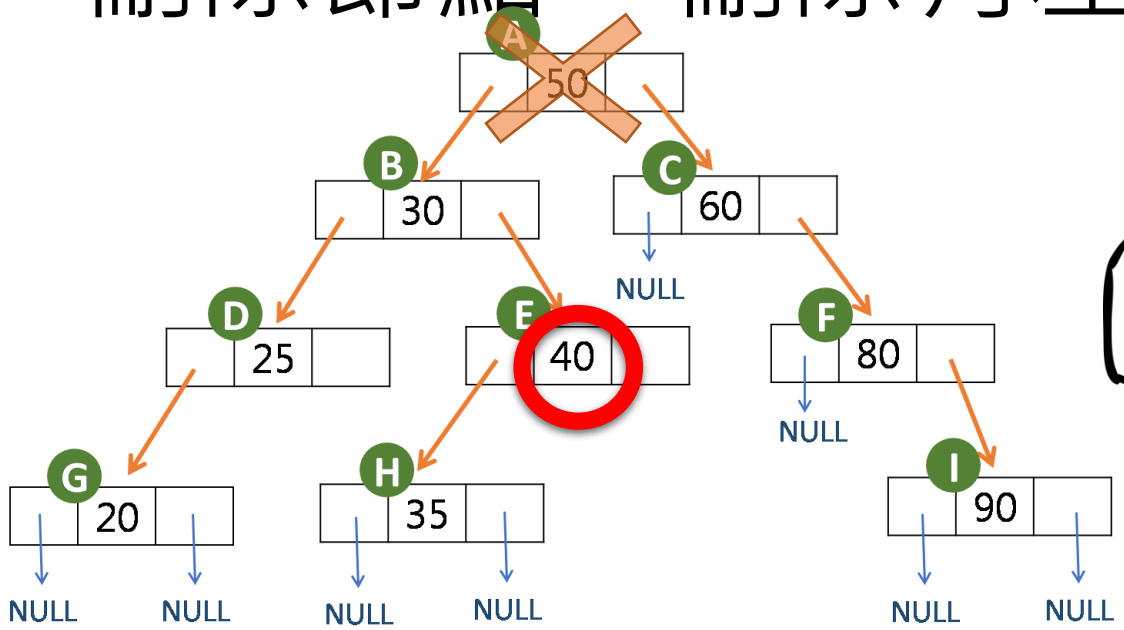
- 步驟 1:
  - 目標: 刪除節點 A
  - 找出節點 A 位置
- 步驟 2:
  - 目標: 刪除節點 A
  - 在節點A的左子樹找取代點最大值
- 步驟 3:
  - 目標: 刪除節點 A
  - 找到節點E(40)做為取代點
- 步驟 4:
  - 目標: 刪除節點 A
  - 將節點A內的值改成節點E的值 40

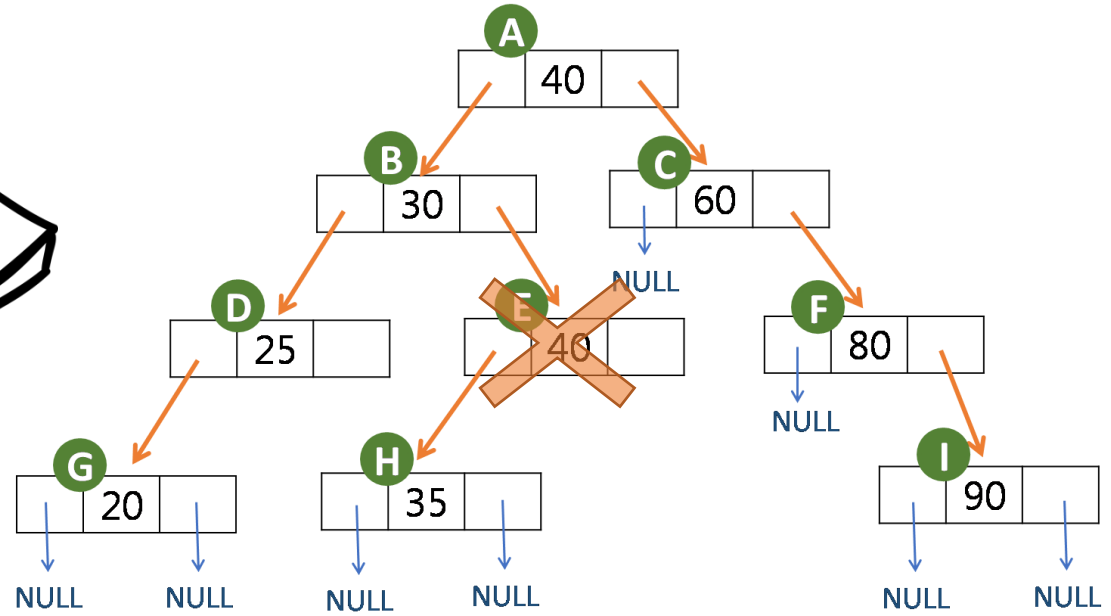
刪除節點 – 刪除有左右子樹的節點 (4)



- 步驟 5:
  - 目標: 刪除節點 E
  - 移到節點 E 位置
- 步驟 6:
  - 目標: 刪除節點 E
  - 將節點 B（節點E的父節點）的 right link 指向節點 H
- 步驟 7:
  - 目標: 刪除節點 A
  - 移除節點 E

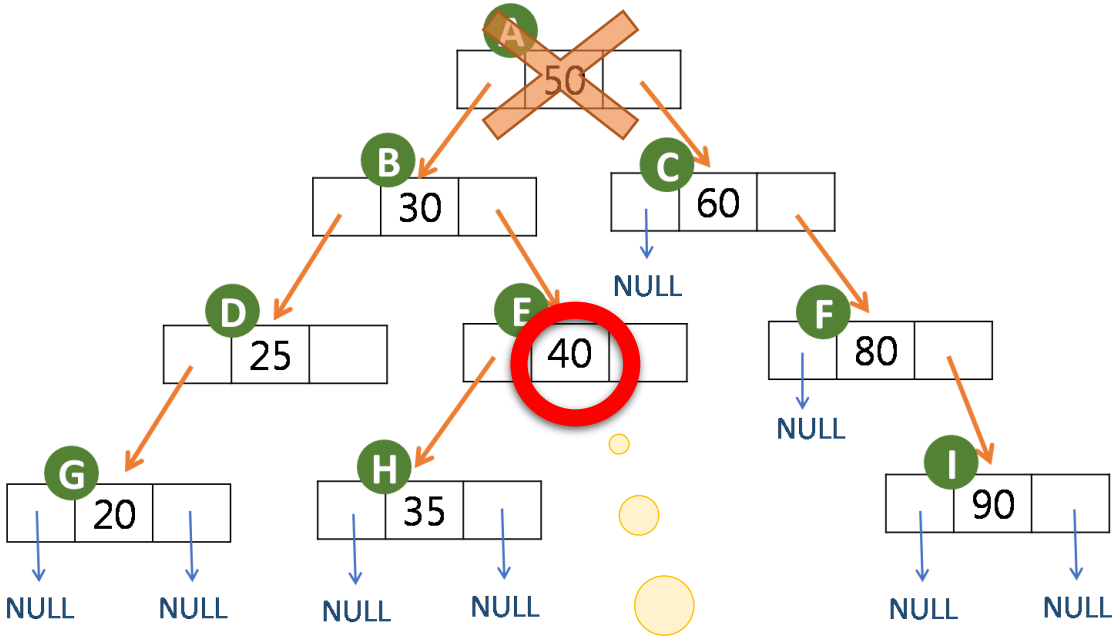
刪除節點 – 刪除有左右子樹的節點 (5)





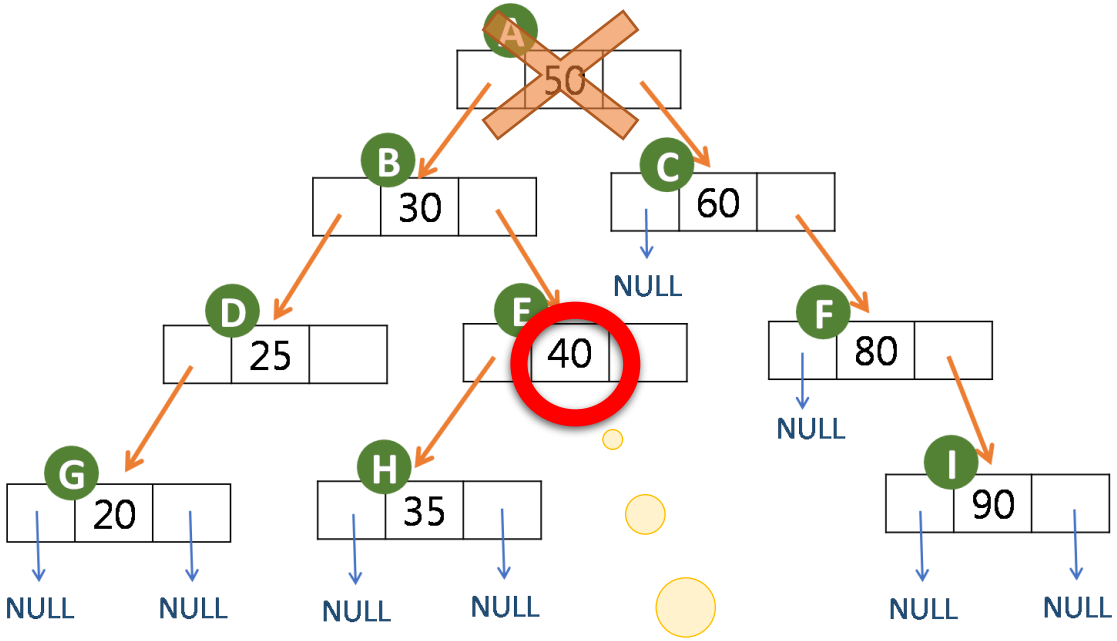
- 步驟 1:
  - 目標: 刪除節點 A
  - 找出節點 A 位置
- 步驟 2:
  - 目標: 刪除節點 A
  - 在節點A的左子樹找取代點最大值
- 步驟 3:
  - 目標: 刪除節點 A
  - 找到節點E(40)做為取代點
- 步驟 4:
  - 目標: 刪除節點 A
  - 將節點A內的值改成節點E的值 40
- 步驟 5:
  - 目標: 刪除節點 E
  - 移到節點 E 位置
- 步驟 6:
  - 目標: 刪除節點 E
  - 將節點B(節點E的父節點)的right link指向節點H
- 步驟 7:
  - 目標: 刪除節點 A
  - 移除節點 E，完成此次的刪除

刪除節點 – 刪除有左右子樹的節點 (6)



想想看，為什麼在左子樹找到的取代點**一定**不會有右兒子呢？

刪除節點 – 刪除有左右子樹的節點 (7)



因為取代點是左子樹的**最大值**，
如果取代點有右兒子的話，那
左子樹的最大值應該會是那個
右兒子！

要刪除的節點不存在時，就直接回傳NULL

要刪除的節點比目前所在節點的值還小時，就往左子樹找尋要刪除的節點位置。

要刪除的節點比目前所在節點的值還大時，就往右子樹找尋要刪除的節點位置。

節點刪除也是會使用到**遞迴**喔！

```
struct node* delete(struct node *root, int x)
{
 if(root==NULL)
  return NULL;
 if (x > root->data)
  root->right = delete(root->right, x);
 else if(x < root->data)
  root->left = delete(root->left, x);
 else
 {
   if(root->left==NULL && root->right==NULL) {
     free(root);
     return NULL;
   }
   else if(root->left==NULL || root->right==NULL) {
     struct node *temp;
     if(root->left==NULL)
       temp = root->right;
     else
       temp = root->left;
     free(root);
     return temp;
   }
   else {
     struct node *temp = find_minimum(root->right);
     root->data = temp->data;
     root->right = delete(root->right, temp->data);
   }
 }
 return root;
}
```


```
struct node* new_node(int x)
{
  struct node *p;
  p = malloc(sizeof(struct node));
  p->data = x;
  p->left = NULL;
  p->right = NULL;
  return p;
}
```

```c
struct node* delete(struct node *root, int x)
{
  if(root==NULL)
   return NULL;

  if (x > root->data)
   root->right = delete(root->right, x);

  else if(x < root->data)
   root->left = delete(root->left, x);

  else
  {
    if(root->left==NULL && root->right==NULL) {
      free(root);
      return NULL;
    }

     else if(root->left==NULL || root->right==NULL) {
      struct node *temp;
      if(root->left==NULL)
        temp = root->right;
      else
        temp = root->left;
      free(root);
      return temp;
     }
    else {
      struct node *temp = find_minimum(root->right);
      root->data = temp->data;
      root->right = delete(root->right, temp->data);
    }
  }
  return root;
}
```

```c
struct node* find_minimum(struct node *root)
{
  if(root == NULL)
    return NULL;
  else if(root->left != NULL)
    return find_minimum(root->left);
  return root;
}
```

要刪除的是葉子節點，就直接移除節點就可以了！

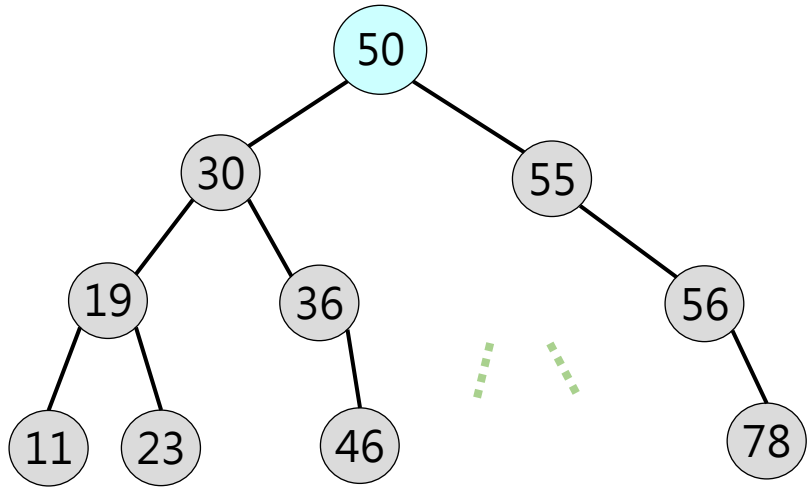
要刪除的只有右子樹，就直接將父節點的 link改成指向右子樹。

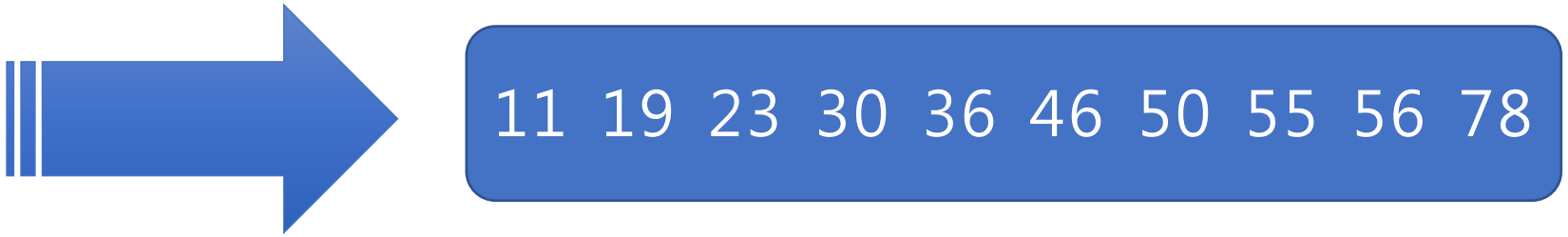
要刪除的只有左子樹，就直接將父節點的 link改成指向左子樹。

要刪除的節點同時有左右子樹時，就在右子樹找最小值做為取代點。

二元排序樹

- 使用二元搜尋樹記錄的資料，只要使用**中序走訪**就會自然產生資料排序結果。



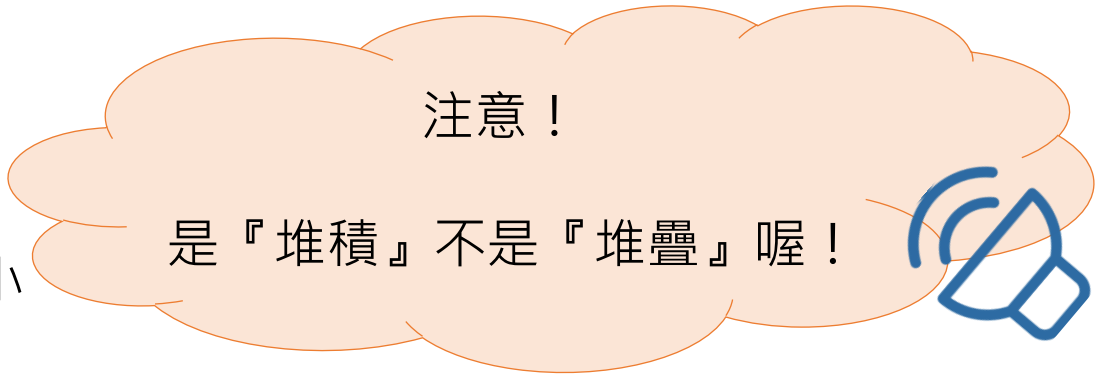


延伸的概念

概念 1: 堆積 Heap (1)

- 堆積 (Heap)，是一種特殊的**完全二元樹**

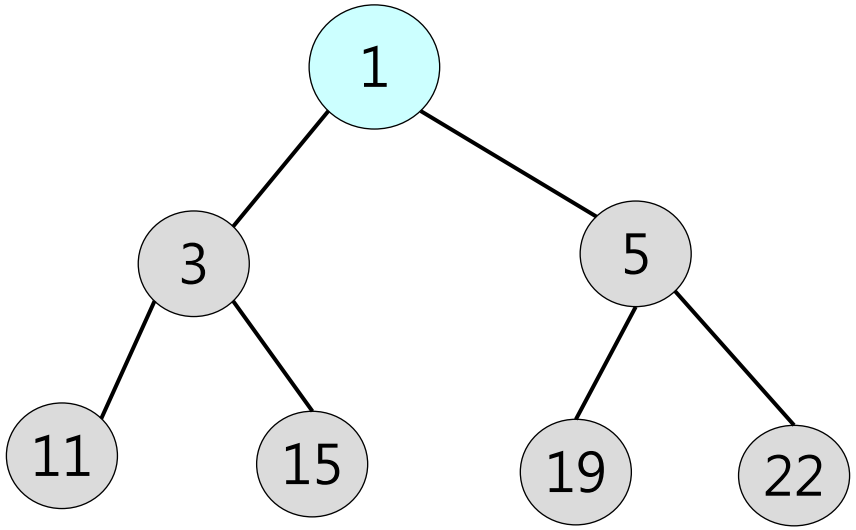
- 堆積有兩種：
  - **最小堆積**
    - 完全二元樹內所有的父節點都比子節點要小
  - **最大堆積**
    - 完全二元樹內所有的父節點都比子節點要大



概念 1: 堆積 Heap (2)

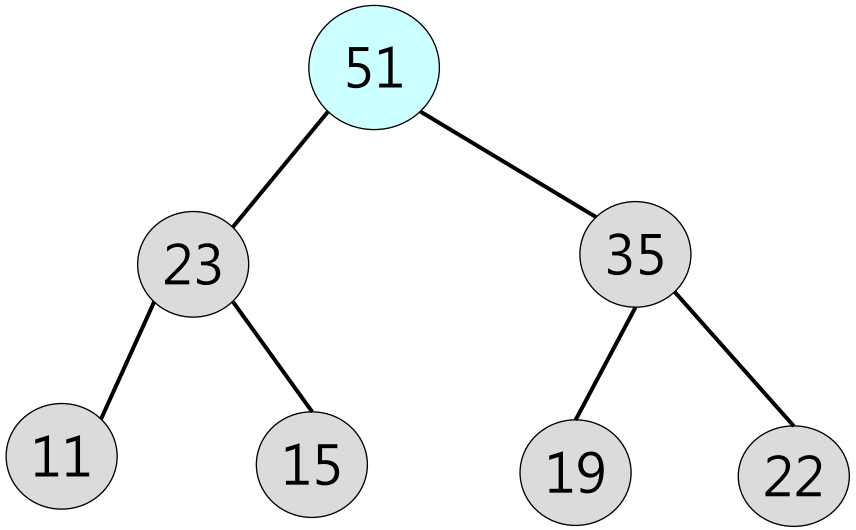
最小堆積

- 所有的父節點都比子節點要小



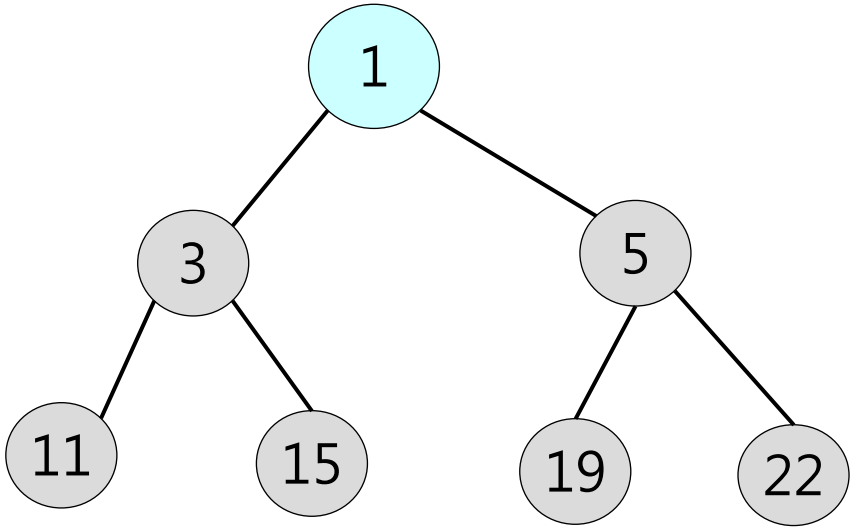
最大堆積

- 所有的父節點都比子節點要大



概念 1: 堆積 Heap (3)

- 堆積適用於取最大或最小值
- 二元搜尋樹適用於排序、搜尋



堆積樹的新增與刪除等功能可以參考此教學網頁

概念 2: 二元樹還有很多議題與應用

- 將一般樹轉二元樹
- 引線二元樹
- 二元運算樹
- 霍夫曼樹 (Huffman’s Tree)
- Level-order traversal