**Ch12. 二元樹 Binary Tree 的基礎**

接下來就要正式進入二元樹的章節。在電腦科學的領域裡，二元樹有相當廣泛的應用，所以要學好資料結構，一定要非常清楚二元樹的理論和實作。

課程大綱

A. 樹的簡介及定義

B. 二元樹

a. 簡介及定義

b. 建立二元樹

C. 二元樹的操作

a. 新增 Insert 與搜尋 Search

b. 刪除 Delete

c. 尋訪 Traversal

一開始，一樣會簡介「樹」是什麼，並給出樹的定義。在各種的「樹」中，最常用到的是「二元樹」，因此會另外講解二元樹的定義，接下來，會進入二元樹的實作。

二元樹當中會有一些基本的操作，包含新增、搜尋、刪除與尋訪。

**第一節：樹的簡介與定義**

上面的圖就是一顆典型的「樹」。

1. 樹的特性

A. 從一個樹根（Root）出發

B. 節點間具有階層結構

a. 具有方向 / 階層關係

b. 在尋訪時可以返回

C. 每個節點都可以連接到零個或多個節點

a. 「不會」形成環（Cycle）：從任一點出發後不會回到原地

b. 當每個節點向外連結的個數 <= 2 時，即為二元樹。

來看一顆樹有怎麼樣的特徵，首先，它由單一的樹根出發，這個樹根本身也是一個節點。節點間具有階層結構，譬如樹的箭頭都是由上到下，具有方向或階層關係，由某筆資料指向另外一筆資料，但是，這不代表在尋訪時不能向上返回，方向和階層只代表了頂點和頂點間的從屬關係。

另外，每個頂點都可以連接到零個或多個頂點，但是無法形成環，比如每個箭頭方向都是從上到下，因此從任一個頂點出發，都不會在若干不後回到原地，也就不會形成頭尾相接的「環」。

樹的每個頂點可以向外連接到若干個頂點，當整棵樹的每個頂點都指向外連接到不多於兩個頂點時，就被叫做「二元樹」。上圖中的樹是二元樹嗎？不是，因為根節點向外連到了 A、B、C 三個頂點，不符合二元樹的限制。要記得二元樹的要求比樹更加嚴格。

2. 樹的應用

A. 列舉所有情況

B. 地圖探索 BFS

C. 資料儲存

D. 搜尋資料

E. 分類器

F. 編碼

什麼時候會用到樹呢？首先，需要「列舉所有情況」的時候就會用到樹。

比如上圖就是一棵樹，這棵樹的用途其實是列舉出「A」、「B」、「C」三個字元不重複所能構成的所有排列。比如開頭是 A，後面可以接 B 或 C 變成「AB」和「AC」，最後再接剩下的一個字元變成「ABC」與「ACB」，所以樹可以用來窮舉所有情況。

下一章會提到編碼和分類是怎麼透過樹來達成的。

3. 樹的定義

A. 由多個節點以及邊所構成

* 但只能存在一個根節點 Root

B. 其餘節點為互不重複（互斥）的集合

a. 稱為根節點的子樹 subtree

b. 每個節點可以指向 0 或多個節點

c. 每個節點只能被一個節點指到

C. 無法形成環 Cycle

若圖中有多個根節點，看作是複數棵樹，稱為樹林。

上面是關於「樹」的比較精確的定義，樹由兩個部分構成，第一個是「節點」，上圖中所有英文字母的部分各是一個節點，第二個是「邊」，且邊是有方向的。

當一個節點指向別的節點，而不被其他節點指到時，就是「根節點」，從 A 出發，有兩條路可以走：向左下走向 AB，或者向右下走向 AC，從 AB 和 AC 往下走都是根節點的子樹，因為去掉 A 之後，由 AB 和 AC 作為根節點時向下出發各自都符合樹的定義。

要注意的是每個頂點雖然可以向外指到多個頂點，但是只能被一個節點指到，比如 ABC 如果同時被 AB 和 AC 指到，就會導致不符合樹的定義。

上圖中，A、B、C 都是不被任何節點指到的根節點，因為各自都是根節點，所以是分開的三棵樹，這三棵樹可以合稱一個樹林 Forest。

4. 節點 Node 的分類

節點大致而言可以分為三種：父節點、子節點和根節點。根節點的定義剛才已經說過，父節點與子節點之間則以誰指向誰來區分，任一個邊都是由「父節點」指向「子節點」，這也代表除了根節點以外的節點都會有對應的父節點。

把除了根節點外的任何節點當成根節點時（不看在他上方的部分），都應該仍然符合樹的定義，比如 A 節點指向「父」和「C」，那麼父和 C 就分別是兩顆子樹的根節點，這兩顆樹分別被叫做「左子樹」和「右子樹」。

另外，從根節點出發要走到任意頂點，都只會有唯一一條路徑。

觀察 A 節點，它的父節點不存在，因為 A 本身是根節點，它的子節點則是 B 和 C，因為從 A 出發有兩條邊分別指向 B 和 C；觀察 B 節點，它的父節點是 A，子節點是 D 和 E；C 節點的父節點一樣是 A，它沒有子節點。

5. 樹相關的名詞定義

(1) 分歧度 Degree：一個頂點指到幾個頂點，也代表一個節點有幾個子節點

A：2

B：3

C：0

Degree of TREE = max{Degree of all nodes}

一棵樹的分歧度定義為其中每個節點分歧度的最大值：二元樹的分歧度最多是 2，每個節點向外不會指到超過 2 個頂點。

(2) 葉節點 Leaf node

葉節點是「沒有」子節點的 node，又稱為外部節點external node：圖中的 C、D、F、G 為葉節點。

(3) 非葉節點 Non-Leaf node

「非葉節點」是指有子節點的 node，又稱為內部節點 internal node，圖中 A、B、E 為非葉節點。

(4) 手足節點 Sibling nodes

手足節點間會有相同的父節點 Parent node。圖中 B 與 C 為手足節點（因為都是 A 的子節點）、D 與 E 和 F 也都是手足節點。

(5) 祖先節點 Ancestor nodes

一個節點一直往父節點的方向走，一直到根節點為止經過的所有節點，都是這個節點的祖先節點，比如從 D 往上走會經過 B 跟 A，因此 B 和 A 都是 D 的祖先節點；G 的祖先節點則有 A、B、E。

(6) 後代節點 / 子孫節點 Descendant nodes

一個節點一路往下方走，能夠達到的所有節點，比如 A 能夠走到其他所有節點，因此除了 A 以外的節點都是 A 的子節點；由 B 出發則可以到達 D、E、F、G，子節點的子節點也算是子孫節點。

(7) 階層 Level

定義根節點的階層為 1，其餘節點的階層是其父節點的階層 +1，也就是說，從根節點出發，每經過一條邊階層就加一。

譬如 A 是根節點，根據定義階層為 1，B、C 的階層為 2，D、E、F 根節點為 3，G 為 4。

(8) 高度 Height

高度的定義是某個節點與「最遠的後代節點」間的邊的個數。

比如距離 B 最遠的後代節點是 G，而 B 與 G 之間有兩條邊，因此 B 的高度為 2。如果一個節點是葉節點（沒有子節點），它的高度就是 0。

樹的高度有兩個實際意義相等的定義，第一個定義是「根節點的高度」，第二個是「所有頂點中階層最高者，其階層 - 1」。

要注意以上是原文書裡的定義，有些台灣考試中對階層與高度的定義不同，比如把高度定義為「頂點中階層最高者，其階層」，在上圖中，這個定義會使得樹高為 4 而非 3。

(9) 深度 Depth

深度的定義是某一節點與根節點間的邊數。可以把根節點 A 看作是地平面，B、C 節點是地下一層，D、E、F 是地下二層、G 是地下三層，因此 B、C 的深度為 1，其他以此類推。

(10) 路徑 Path

路徑指的是祖先節點到後代節點連成的邊，比如由 A 到 B 到 D，可以記做 A-B-D，A 到 G 的路徑是 A-B-E-G。

6. 樹的記錄法

(1) 括號法

以括號表示每個節點有哪些子節點，每個節點後方以一個小括號內含其所有子節點，如果子節點又有子節點，同樣後接小括號與其所有的子節點，形式可如

父（子1（孫）子2（孫））

上圖中的樹以括號法表示為：A(B(DE(G)F)C)，因為 A 的子節點為 B 與 C，先寫為 A(BC)，B 的子節點為 DEF，可以寫為 B(DEF)，E 的子節點為 G，寫為 E(G)，結合起來即為整棵樹的括號表示法。

(2) 鏈結串列

另一個方法是利用之前提過的鏈結串列 Linked List，透過指標把每個頂點串連起來，A 頂點會內含兩個指標分別指到 B 跟 C，B 頂點有三個指標分別指到 D、E、F，E 頂點有一個指標指到 G，就像這樣透過指標把頂點和頂點之間串連起來。

在使用鏈結串列表示一棵樹時，會希望每個頂點都是一樣的結構（成員相同，意即每個頂點內含的指標數相同），這代表每個頂點都要有 Degree of Tree 樹的分歧度（每個頂點中分歧度最高者）個指標才夠用，比如圖中分歧度最大的頂點 B 有三個子節點，分歧度為 3，因此要讓每個頂點的結構相同，就要各都含有 3 個指標才夠用。

[樹的圖]

7. 練習一下： 下面的六張圖中，哪些是樹？

A. 樹的條件

a. 有且僅有一個根節點

b. 每個節點可以有多個子節點，但只能有一個父節點

（只有根節點沒有父節點）

c. 節點間不形成環

圖 A：滿足定義

圖 B：不滿足「每個頂點只能有一個父節點」的要求（E 頂點）

圖 C：滿足定義

圖 D：不滿足「只有一個根節點」的要求

圖 E：不滿足「只有一個根節點」的要求

圖 F：不滿足「不形成環」的要求（由 B 到 C 後，可以再回到 B）

因此根據定義，只有 A 和 C 是樹。

8. 試著寫出下圖中樹的各種性質

A. 樹高 Height of tree

B. 哪些頂點為葉節點 Leaf nodes (external nodes)

C. 哪些頂點不是葉節點 Non-leaf nodes (internal nodes)

D. 節點 E 的深度 Depth of node#E

E. 節點 B 的高度 Height of node#B

F. G 有哪些手足節點 Sibling nodes of node#G

A：4（根節點 A 離最遠的後代節點的邊數）

B：DFGIJ（沒有子節點）

C：ABCEH（有子節點）

D：2（距離根節點 A 的邊數）

E：1（B 離最遠的後代節點 D 的邊數）

F：H（G 跟 H 有同一個父節點）

9. 試著用括號法描述上圖的樹

A(BC)

B(D)

C(EF)

E(GH)

H(IJK)

-> A(B(D)C(EF))

-> A(B(D)C(E(GH(IJK))F))

**第二節：二元樹簡介與定義**

接下來要來看「樹」這種結構裡面最常使用的「二元樹」。

首先，剛剛有提到過可以利用鏈結串列來表示樹，每個頂點透過「指標」來連接到下一個頂點。

1. 空間利用率與二元樹

但是在樹裡，一個頂點有時要連接到很多個頂點，每個結構下要有幾個指標，是由「樹的分歧度 Degree of Tree」來決定，亦即由「分歧度最大的頂點」來決定，才能確保每個節點需要的結構大小是相同的。

結果很多指標並沒有被用到，比如 A 結構中，只用到 p1 和 p2 兩個指標，C 結構中，三個指標都沒有用到。

可以看出這種做法會浪費非常多空間：

A. 假設 n 為節點數目，k 為 Degree of Tree

B. 總共能容納的指標數目：n x k

C. 實際用到的指標數目：n-1（除了根節點外，每個節點正好有一個父節點）

空間浪費率公式：

觀察上式，空間浪費率與節點間的連接方式（樹的形狀）無關，只由樹的分歧度 k 決定。

比如上面的圖中，如果覺得 C 節點的指標都沒有被使用到，太過浪費，所以決定把 G 節點改接到 C 節點後，那麼 C 節點就「少浪費了一個指標空間」，但是同時，E 節點不再往後接到 C 節點，因此 E 節點同時「多浪費了一個指標空間」，也就是說，無論如何調整這棵樹的形狀，浪費的空間比例是一定的。

由於空間浪費率可以表示為 ：

k = 1：0.0%

k = 2：50%

k = 3：67%

k = 4：75%

k = 1 時，整個結構相當於鏈結串列 linked list，如果要使一個節點可以連接到超過一個節點，那麼只能取 k = 2，使得空間浪費率最小，這就是「二元樹」最常被使用的原因。

2. 二元樹的定義

二元樹是在符合「樹」的要求的前提下，再加上額外的限制： Degree of tree 為 2。

也就是說，在二元樹中，每個父節點最多只能連接到兩個子節點，這兩個子節點分別被稱為「左節點 Left child」和「右節點 Right child」。

3. 二元樹的特性

在第 i 個階層上的節點個數： 。

第一層為根節點，只有一個節點，第二層最多有兩個節點，第三層最多有四個節點，以此類推。

最大階層 h 的二元樹，節點最多為 。

（等比級數公式）

另外，如果節點數量為 ，邊數 。

4. 二元樹的數學運算

(1) 葉節點個數的計算

葉節點個數 = 分岔度 2 的節點數 + 1

：分岔度 的節點個數， 者為葉節點。

是節點總數， 代表分岔度分別為 0、1、2 的節點數，前面說明過 會相當於邊數 。

同時，邊數 也可以表示成 ，因為 一定向後連接到另一個節點，需要一條邊， 則向後連接到兩個節點，需要兩條邊。

把 的兩種表示方式連接起來，並移項整理，就會發現 與 間正好相差 1。

(2) n 個節點共可以組成幾種二元樹？

能組成的二元樹的種類 可以用數學歸納法證明，但很麻煩，因此這裡只列出結論。

(3) 高度為 N 的二元樹（定義根節點高度為 1），最多能有幾個節點？

關於樹的基本觀念都有可能出現在考題當中，比如「台北市國中教師聯合甄選電腦專業科目」中出現過這題，答案為 D。

（A）

（B）

（C）

（D）

5. 二元樹的應用

A. 新增、刪除：二元搜尋樹 Binary Search Tree（BST）

* C++ STL 中的 map 和 set 就是由二元搜尋樹中的紅黑樹實作

B. 檔案壓縮：霍夫曼編碼 Huffman Coding

C. 機器學習：決策樹 Decision Tree

D. 路由表 Router-tables：二元樹

E. 作業系統中的處理佇列

F. 堆積：優先權佇列

下一章就會介紹二元樹的一些應用，因為應用繁多，二元樹在面試時基本上是必考題。

6. 二元樹與樹定義的差異

|  |  |
| --- | --- |
| 樹 | 二元樹 |
| 不可以是空集合 | 可以是空集合 |
| 分岔度 >= 0 | 0 <= 分岔度 <= 2 |
| 子樹無次序之分 | 左右子樹有次序之分 |
| 不區分左右節點 | 區分左右節點 |

7. 二元樹的各種分類方式

A. 斜曲二元樹 Skewed binary tree

a. 左斜曲 Left-skewed 二元樹

b. 右斜曲 Right-skewed 二元樹

B. 嚴格二元樹 Strictly binary tree

C. 完滿二元樹 Full Binary Tree

D. 完整二元樹 Complete Binary Tree

(1) 斜曲二元樹

完全沒有某一側的子節點，形狀和功能基本上與鏈結串列相同。

左斜曲：完全沒有右節點

右斜曲：完全沒有左節點

斜曲二元樹是應「避免」發生的情形，因為複雜度與鏈結串列相同，沒有展現出樹中較好的特性。

(2) 嚴格二元樹

每個非葉節點都有兩個子節點，也就是說，整棵樹中只有分歧度為 0 的葉節點，和分歧度為 2 的非葉節點。

，即分岔度 1 的節點個數為 0。

(3) 完滿二元樹

每個非葉節點都有兩個子節點，且每個葉節點的階層都相同。

最大階層 h 的完滿二元樹共有 個節點，也就是節點數量的上限。

最大階層 3 時，節點有 個。

(4) 完整二元樹

節點按照次序排列後，連續且沒有空缺，次序是由上而下、由左至右。

完整二元樹

非完整二元樹

上例中，因為編號為 3 的位置沒有節點，但編號為 4 與 5 處卻有節點，所以並非完整二元樹。

對於一個完整二元樹而言：

< 節點個數 <

最少的情況發生在最下方的階層只有一個節點時，此時節點個數為 。

8. 二元樹的資料表示法

A. 陣列

a. 適合完滿二元樹、完整二元樹

b. 不適合斜曲二元樹

c. 容易尋訪

B. 鏈結串列

a. 適合斜曲二元樹

b. 會有浪費指標空間的問題（浪費 ，50% 的空間）

(1) 陣列表示法

若二元樹的最大階層為 h，節點最多有 個，因此要準備這個長度的陣列，依序把所有資料放入。

如果二元樹不是完滿的，沒有節點的位置仍要在陣列中空出，比如下圖中的斜曲二元樹只有 A、B、D 三個節點，這三個節點都要放在對應的陣列位置當中，其他位置則空著，導致空間浪費率很大。

第 k 個節點

Left child：索引值

Right child：索引值

Parent：索引值 向下取整

上圖中，B 節點編號是 2，代入上方公式可以判斷其左節點 D 在陣列中的「索引值」是 3（注意不是編號），右節點 E 的「索引值」則是 4。

另外，也可以計算出 B 節點和 C 節點的共同父節點 A 的「索引值」是 0。

(2) 陣列表示法的優點與缺點

A. 優點

a. 同一階層內左右節點容易取得，可以透過公式直接算出索引值

b. 完滿二元樹時不浪費任何空間

B. 缺點

a. 新增與刪除資料較為困難：陣列大小不夠時，要搬移所有資料

b. 斜曲二元樹時浪費大量空間

c. 空間利用率：，例如高度為三的斜曲二元樹，利用率是

(3) 鏈結串列表示法

節點內容

A. 資料內容

B. 左節點 left child node 指標

C. 右節點 right child node 指標

D. 父節點指標，可以幫助我們往父節點方向移動（非必要）

(4) 鏈結串列表示法的優點與缺點

A. 優點

a. 新增與刪除資料容易

b. 斜曲二元樹時，比陣列節省空間

B. 缺點

a. 若節點中沒有父節點指標，難以向上方移動

b. 至少會浪費一半的節點空間，空間浪費率：，

9. 下面的左圖與右圖中，哪個是完整二元樹？請用陣列法表示這兩棵樹。

ABCDEF\_

ABCDE\_F

可以看出完整二元樹的陣列表示法中，所有資料都是連續的。

**第三節：建立二元樹**

1. 節點結構

A. 資料內容

a. 編號（方便檢索）

b. 資料內容

B. 左節點

C. 右節點

D. 父節點（方便返回，非必要）

每個頂點的結構中，至少會有兩個指標分別指向左節點與右節點，分別將其命名為 left 和 right，另外，也可以加入父節點的指標方便返回上個階層。

|  |  |
| --- | --- |
| 二元樹的節點 | |
| 1  2  3  4  5  6  7 | template<typename T>  struct Node{  int index;  T data;  Node<T> \*left; // 指向左節點  Node<T> \*right; // 指向右節點  }; |

2. 二元樹類別

A. 根節點

B. 印出特定節點（方便測試）：Print

C. 新增資料：Insert

D. 搜尋資料，回傳頂點指標：Search

E. 刪除資料：Delete

F. 尋訪

a. 前序

b. 中序

c. 後序

|  |  |
| --- | --- |
| 二元樹的類別宣告 | |
| 1  2  3  4  5  6  7  8  9  10 | template<typename T>  class Binary\_Tree{  public:  Node<T>\* root;  Binary\_Tree();  void Print(Node<T>);  bool Insert(int, T);  Node<T>\* Search(int);  bool Delete(int);  }; |

3. 二元搜尋樹的編號方式

為了檢索/搜尋的需求，必須將每個節點加以排序，也就是節點中的「編號 index」。

編號的規則設定為：如果一個節點比其父節點的編號小，代表其為左子樹；反之，比其父節點的編號大的話，代表其為右子樹。

這樣一來，假定每個節點的編號與其中的資料 data 相同，如果要搜尋一筆資料，可以先把它的值與 A 比較，比 A 小時往左子樹移動，比 A 大時，則往右子樹移動。假設往左子樹移動，同樣比較值和 B 何者較大，決定要往 D 還是 E 的方向移動。

實際來看一個例子，如果先插入一筆資料 58，58 成為樹的根節點。接下來插入 97，因為 97 比 58 來得大，因此成為 58 這個節點的右節點。

接下來插入 24，因為 24 比 58 還要小，因此插入後變成 58 的左節點（注意不是插在 97 的下方，一定要從根節點開始比較決定往左或往右移動）。

再插入 82，82 比 58 大，因此先往右子樹移動，遇到 97 時，因為比 97 小，所以 82 成為 97 的左節點。

插入 49，同樣先與根節點 58 比較，因為 49 較小，因此往左移動，遇到 24 後，因為比 24 大，因此 49 是 24 的右節點，後續以此類推。

對於子節點而言，結構中的兩個指標 left 和 right 都是空指標 nullptr。

根據上述規則建立的二元樹稱為「二元搜尋樹」，可以方便我們找到資料在二元樹中的位置。

4. 在二元搜尋樹中搜尋資料

在二元搜尋樹中進行搜尋時，可以依照下列方法進行

要找的編號與現處的節點一致：結束，回傳現處節點

要找的編號跟現處的節點不一致：

要找的編號比節點編號小：往左節點移動

要找的編號比節點編號大：往右節點移動

當現在的節點為 left node（但編號仍不符合）：結束，回傳空指標

要搜尋 38 時，依序上述步驟進行

A. 從根節點 58 開始移動

B. 38 < 58，往左節點移動

C. 38 > 24，往右節點移動

D. 38 < 48，往左節點移動

E. 找到相符資料，結束搜尋

5. 搜尋的複雜度

(1) 若是完滿二元樹

A. 每經過一次分岔，可以刪去一半的節點

B. 經過 n 次搜尋，可以找遍 筆資料

C. 搜尋次數約等於

D. 搜尋、新增、刪除都是

因為樹的結構非常常見，所以使得資料結構與演算法的學科中，常常看到以 2 為底的 log。

[圖]

在先前看過的複雜度圖表中，屬於相當優秀的算法表現。

(2) 各種樹的複雜度

二元樹的複雜度為

三元樹的複雜度為

四元樹的複雜度為

三、四元樹的時間複雜度比二元樹低，新增、刪除等操作所需的時間更短，然而它們的空間浪費率較高，如前面看過的，空間浪費率的值可以用樹的分歧度 k 表示。

k = 1：0.0%

k = 2：50%

k = 3：67%

k = 4：75%

這樣一來，究竟該使用較節省空間的二元樹，還是使用速度較快的三、四元樹呢？

(3) 選擇二元樹的原因

橫軸跨越的範圍小時，、、 的表現似乎有相當的差距。

但當橫軸跨度大時，、、 間的差距可以忽略。

從上圖中可以看到，事實上 、、 在 n 的值較大時相差無幾，只要複雜度可以用 的方式表示，都在可接受的範圍內。

因此一般來說，比起三元樹、四元樹、...，還是優先使用二元樹。

(4) 比較不同種的樹

[比較表]

上表中，每一列代表分歧度不同的樹搜尋所需的次數，每一行則代表資料個數。

即使資料多達 100,000 筆，二元樹所需的次數 為 17 次，四元樹則為 8 次，兩者只差 8 次運算（相對的， 和 所需的運算次數相差近 100,000 次），然而空間浪費率卻相差很大，代表多數時候二元樹是表現較為理想的結構。

使用二元樹時，還是要特別注意不要形成「斜曲二元樹」，因為斜曲二元樹中進行的搜尋、新增、刪除的複雜度都是 O(n)，與鏈結串列相同，但是比鏈結串列還更浪費空間（因為每個節點含有兩個指標，多於鏈結串列中的一個指標）。

6. 初始化一個二元樹的架構，並完成以下兩個函式：

A. 建構式 Constructor

B. Print

|  |  |
| --- | --- |
| struct Node 和 class Binary\_Tree 的宣告 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include <iostream>  using namespace std;  // 節點的結構  template<typename T>  struct Node{  int index;  T data;  Node<T> \*left;  Node<T> \*right;  };  // 二元樹類別  template<typename T>  class Binary\_Tree{  public:  Node<T>\* root;  Binary\_Tree();  void Print(Node<T>);  bool Insert(int, T);  Node<T>\* Search(int);  bool Delete(int);  };  ... // 函式定義寫在這裡  int main()  {  return 0;  } |

|  |  |
| --- | --- |
| Binary\_Tree 的建構式 | |
| 1  2  3  4  5  6  7 | template<typename T>  Binary\_Tree<T>::Binary\_Tree(){  // 根節點的指標設定為空指標  root = nullptr;  } |

|  |  |
| --- | --- |
| Print 印出特定節點 | |
| 1  2  3  4  5  6  7 | template<typename T>  void Binary\_Tree<T>::Print(Node<T> node){  cout << "Index: " << node.index << endl;  cout << "Data: " << node.data << endl;  } |

7. 完成以下兩個函式

A. Insert：插入一筆資料

* 插入成功時回傳 true
* 沒有成功（與現有節點重複）時回傳 false

B. Search：搜尋特定「編號」的資料（依據剛才介紹過的編號方式）

* 回傳搜尋到的指標，未搜尋到時回傳空指標

|  |  |
| --- | --- |
| Insert | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | template<typename T>  bool Binary\_Tree<T>::Insert(int target, T value){    // 例外處理：樹是空的  if (root == nullptr){  // 新增空節點，利用大括號指定 {index, data, \*left, \*right}  root = new Node<T>{target, value, nullptr, nullptr}  return true;  }  // 一般情況  // 要找到應插入的位置  Node<T>\* current = root;  while (true){  // 要插入的編號已經存在了，回傳 false  if (current->index == target){  return false;  // 要插入的編號比目前頂點編號小，往左邊走  } else if (target<current->index){    // 發現左節點是空的，直接插入成為左節點  if (current->left == nullptr){  current->left = root = new Node<T>{target, value, nullptr,  nullptr};  // 新增成功  return true;  }  // 若左節點有值，往左節點移動  current = current->left;  // 要插入的編號比目前頂點編號大  } else {  // 發現右節點是空的，直接插入成為右節點  if (current->right == nullptr){    current->right = root = new Node<T>{target, value, nullptr,  nullptr};  // 新增成功  return true;  }  // 若右節點有值，往右節點移動  current = current->right;  } //end of if  } // end of while  } // end of Insert |

|  |  |
| --- | --- |
| Search | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | template<typename T>  Node<T>\* Binary\_Tree<T>::Search(int target){  // 例外處理：樹是空的  if (root == nullptr)  return nullptr;  // 一般情況，與新增時進行方法類似  Node<T>\* current = root;  while(true){  if(current->index == target){  return current;  }  // 向左移動  else if (target < current->index){  // 要向左繼續搜尋，但遇到空指標  if (current->left == nullptr){  return nullptr;  }  current = current->left;  }  // 向右移動  else{  if (current->right == nullptr){  return nullptr;  }  current = current->right;  } // end of if  } //end of while    } // end of Search |

|  |  |
| --- | --- |
| 測試 Insert 和 Search | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int main()  {  // 宣告存放字串的二元樹 tree  Binary\_Tree<string> tree;  tree.Insert(1,"Mick");  tree.Insert(2,"Rallod");  tree.Insert(3,"Daphene");  // found 是指向節點的指標  Node<string>\* found = tree.Search(2);  // \*found 是節點  tree.Print(\*found);  } |
| 執行結果 | |
| Index: 2  Data: Rallod | |

**第三節：二元樹的尋訪**

尋訪 Traversal 指的是要向該節點連結的「所有」節點進行移動

A. 移動後可進行讀寫、新增、刪除等操作

B. 如下圖，A 可分別至 B、C 進行操作

C. 在樹的尋訪中，希望每個節點都能剛好被操作一次

比如圖中的二元樹中只有三個節點 A、B、C，要尋訪這個二元樹，就要把三個節點都各自進行一次處理（比如印出資料）。

1. 尋訪的順序

像上面這樣的單一子樹，可以一般化表示成具有 M、L、R 三個節點的二元樹，共有 L、M、R 三種操作

A. 操作 M 節點的資料（處理 M 的資料）

B. 往 L 方向探索（處理 L 的資料）

C. 往 R 方向探索（處理 R 的資料）

三個操作的排列數為 3!=6，也就是上圖中列出的所有可能順序，共有 6 種。為了避免左右對稱造成的重複，規定 L 一定要出現在 R 之前，因此剩下 3 種排列。

這三種排列 MLR、LMR、LRM 根據中間的頂點 M 被處理的次序，分別被叫做「前序」、「中序」與「後序」。

前序 Pre-order：中->左->右

中序 In-order：左->中->右

後續 Post-order：左->右->中

2. 前序尋訪

前序尋訪的步驟

A. 處理當前節點的資料

B. 往左探索（遞迴）

C. 往右探索（遞迴）

|  |  |
| --- | --- |
| 前序尋訪的程式碼 | |
| 1  2  3  4  5 | void Pre\_Order(Node\* n){  n->data;  Pre\_Order(n->left);  Pre\_Order(n->right);  } |

3. 中序尋訪

中序尋訪的步驟

A. 往左探索（遞迴）

B. 處理當前節點的資料

C. 往右探索（遞迴）

|  |  |
| --- | --- |
| 中序尋訪的程式碼 | |
| 1  2  3  4  5 | void In\_Order(Node\* n){  In\_Order(n->left);  n->data;  In\_Order(n->right);  } |

4. 後序尋訪

後序尋訪的步驟

A. 往左探索（遞迴）

B. 往右探索（遞迴）

C. 處理當前節點的資料

|  |  |
| --- | --- |
| 後序尋訪的程式碼 | |
| 1  2  3  4  5 | void Post\_Order(Node\* n){  Post\_Order(n->left);  Post\_Order(n->right);  n->data;  } |

5. 演練前序尋訪

先從根節點 A 出發，前序指的是「先處理」該節點的資料，因此要最先處理 A，接著往左邊走，走到 B。

B 一樣要先被處理，處理完往左邊走，處理 D。D 處理完後，試著往 D 的左節點走，但該處沒有節點，接著嘗試往右節點走，發現該處同樣沒有節點，因此要退回 B 處。

對於 B 來說，B 本身和左節點 D 都已經處理完了，下一步是處理 B 節點的右節點 E。

就像處理 D 節點的程序一樣，處理 E 節點後，因為發現沒有左右節點，因此往回退回 B，又因為 B 與其左右節點 DE 都已處理完成，再退回 A。

對 A 而言，A 本身和其左子樹已經完全處理完成，接下來往右處理 C。

同樣的道理，處理完 C 後，會依序處理 F 和 G。

上述過程表示該二元樹採取前序處理會依序處理 ABDECFG。

6. 演練中序尋訪

中序尋訪代表要把一個節點的「左子樹」都處理完才處理該節點，隨後處理該節點的右子樹。

同樣的二元樹以中序處理，從 A 開始，並不能直接處理 A，而是先往其左子樹方向移動到 B，到了 B，也不能直接處理，而是要先處理其左子樹中的 D。

到了 D 之後，發現左子樹不存在，因此這時就可以處理 D，並且因為 D 的右子樹也不存在，D 節點已經處理完成。隨後，因為左子樹 D 已經被處理，B 此時也就可以被處理，並且處理完 B 本身後，接著處理 B 的右子樹 E。

依此類推，中序尋訪的處理順序為 DBEAFCG。

7. 演練後序尋訪

後序尋訪指的是一個節點要在其左右子樹依序都被處理完後，才進行處理。同樣從 A 出發，先往左移動到 B，再網左移動到 D。

因為 D 的左右子樹都不存在，兩個方向都檢查（不需做處理）後，就可以處理 D 本身。D 處理完後，可以退回 B，但是因為 B 的右子樹還沒處理完成，因此要優先處理 E。

E 也處理完後，B 隨之就可以被處理。接下來退回 A，但在處理 A 之前，必須用同樣的方法處理完 A 的右子樹。

後序尋訪的處理順序為 DEBFGCA。

8. 比較不同尋訪順序

前序：ABDECFG

中序：DBEAFCG

後序：DEBFGCA

把前序、中序、後序的處理順序都列出，會發現一個特性：前序中，根節點 A 是第一個處理，後序中，根節點最後一個處理，中序裡，根節點則會夾在左子樹與右子樹間處理。

9. Level-Order Traversal

這是另外一種尋訪方式，以「階層」為單位，階層的順序由上至下，而同一階層內的處理順序則是由左至右。下圖中，處理順序會是 ABCDEFG。

Level-Order Traversal 的步驟

A. 準備一個 Queue

B. 把根節點放入 Queue

C. 依序從 Queue 中取出節點

D. 把上一步中取出節點的子節點放入 Queue

E. 重複步驟 C 與 D

把根節點 A 放入 Queue(Q)

Q = {A}

取出 A，把 A 的子節點 B、C 放入 Q

Q = {B, C}

取出 B，把 B 的子節點 D、E 放入 Q

Q = {C, D, E}

取出 C，把 C 的子節點 F、G 放入 Q

Q = {D, E, F, G}

依序取出 D、E、F、G

先把 A 放入 Queue，現在 Queue 中只有 A，因此取出後，放入子節點 B 和 C。重複步驟，把 B 取出，並放入 B 的子節點 D 和 E，接著把 C 取出，放入 C 的子節點 F 和 G。

最後，依序取出此時 Queue 中的 D、E、F、G 就完成尋訪了。

10. Left most 與 Right most

Left most

A. 回傳該子樹中，最左（小）的節點

B. 不斷往左節點走，直到遇到空指標

C. 58 的 Left most：21

Right most

A. 回傳該子樹中，最右（大）的節點

B. 不斷往右節點走，直到遇到空指標

C. 58 的 Right most：73

11. Predecessor 與 Successor

按照先前介紹的編號方式，中序 Inorder 處理的方式正好按照編號（索引）由小而大的順序。

DBEAFCG 對應的順序正好是 21 -> 32 -> 47 -> 58 -> 62 -> 69 -> 73。

這種對應關係之所以會成立，是因為中序的處理方式就是先處理「左子樹」，所有編號比該節點小的節點，接著處理該節點，隨後處理「右子樹」，也就是所有編號比該節點大的節點。

因此中序處理的順序會與編號從小到大的順序相同。

(1) Predecessor

A. 依照編號大小排序時，該節點的前一個節點

B. 又名 Inorder\_Predecessor

C. 32 的 Predecessor：21

(2) Successor

A. 依照編號大小排序時，該節點的後一個節點

B. 又名 Inorder\_Successor

C. 32 的 Successor：47

12. 尋找 Predecessor 的方式

找到的 Predecessor 會是「比當前節點的編號小的所有節點」中編號最大者。

第一種狀況：左子樹不為空時，左子樹的 Right most（找到左子樹中最大的值）

第二種狀況：左子樹為空時，往該節點的祖先節點方向移動，直到該節點位於目前祖先節點的右子樹（在圖中，代表一路往上走的過程中，經過至少一條左上斜往右下的邊）。

24 的 Predecessor：12

12 的 Predecessor：null

97 的 Predecessor：82

82 的 Predecessor：58（往上移動到 97 時邊的方向不對，要再往上走到 58 才會經過左上到右下的邊）

因為要往祖先節點的方向移動，所以有 Parent 指標時，寫法會較簡便。

13. 尋找 Successor 的方式

找到的 Successor 會是「比當前節點的編號大的所有節點」中編號最小者。

第一種狀況：右子樹不為空時，右子樹的 Left most（找到右子樹中最小的值）

第二種狀況：右子樹為空時，往該節點的祖先節點方向移動，直到該節點位於目前祖先節點的左子樹（在圖中，代表一路往上走的過程中，經過至少一條右上斜往左下的邊）。

24 的 Successor：49

12 的 Successor：24

97 的 Successor：null

82 的 Successor：55（往上移動到 49、24 時邊的方向都不對，要再往上走到 58 才會經過右上到左下的邊）

同樣的，要尋找 Successor 時，有 Parent 指標會使得寫法會較簡便。

14. 寫出尋訪函式

A. 前序 Pre-order

B. 中序 In-order

C. 後序 Post-order

D. Level-Order Traversal

擴寫先前的程式碼：

|  |  |
| --- | --- |
| Binary\_Tree 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <queue> // for Level\_Order  template<typename T>  class Binary\_Tree{  public:  ...  void Pre\_Order(Node<T>\*);  void In\_Order(Node<T>\*);  void Post\_Order(Node<T>\*);  void Level\_Order(Node<T>\*);  } |

|  |  |
| --- | --- |
| 前序尋訪 Pre\_Order | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template<typename T>  void Binary\_Tree<T>::Pre\_Order(Node<T>\* node){  // 例外處理：樹是空的，不處理  if (node == nullptr)  return ;  // 處理當前節點  cout << node->index << " ";  // 處理左節點（遞迴）  Pre\_Order(node->left);  // 處理右節點（遞迴）  Pre\_Order(node->right);  } |

|  |  |
| --- | --- |
| 中序尋訪 In\_Order | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template<typename T>  void Binary\_Tree<T>::In\_Order(Node<T>\* node){  // 例外處理：樹是空的，不處理  if (node == nullptr)  return ;  // 處理左節點（遞迴）  In\_Order(node->left);  // 處理當前節點  cout << node->index << " ";  // 處理右節點（遞迴）  In\_Order(node->right);  } |

|  |  |
| --- | --- |
| 後序尋訪 Post\_Order | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template<typename T>  void Binary\_Tree<T>::Post\_Order(Node<T>\* node){  // 例外處理：樹是空的，不處理  if (node == nullptr)  return ;  // 處理左節點（遞迴）  Post\_Order(node->left);  // 處理右節點（遞迴）  Post\_Order(node->right);  // 處理當前節點  cout << node->index << " ";  } |

|  |  |
| --- | --- |
| Level Order Traversal | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | template<typename T>  void Binary\_Tree<T>::Level\_Order(Node<T>\* node){  // 準備一個 Queue  queue<Node<T>\*> Q;  Q.push(node);  // 當 Queue 中還有資料時  while (!Q.empty()){  // 取出 Queue 中最前面的一筆資料  Node<T>\* current = Q.front();  Q.pop();  // 印出當前這筆資料  cout << current->index << " ";  // 如果有左節點，把左節點放入 Queue  if (current->left){  Q.push(current->left);  }  // 如果有右節點，把右節點放入 Queue  if (current->right){  Q.push(current->right);  }  } // end of while  } // end of Level\_Order |

|  |  |
| --- | --- |
| 測試前序、中序、後序與 Level-order 尋訪 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | int main()  {  Binary\_Tree<string> tree;  tree.Insert(58, "A");  tree.Insert(24, "B");  tree.Insert(97, "C");  tree.Insert(12, "D");  tree.Insert(49, "E");  tree.Insert(82, "F");  tree.Insert(55, "G");  // 測試 Pre\_order  cout << "Pre-order: ";  tree.Pre\_Order(tree.root);  cout << endl;  // 測試 In\_order  cout << "In-order: ";  tree.In\_Order(tree.root);  cout << endl;  // 測試 Post\_order  cout << "Post-order: ";  tree.Post\_Order(tree.root);  cout << endl;  // 測試 Level\_order  cout << "Level-order: ";  tree.Level\_Order(tree.root);  cout << endl;  } |
| 執行結果 | |
| Pre-order: 58 24 12 49 55 97 82  In-order: 12 24 49 55 58 82 97  Post-order: 12 55 49 24 82 97 58  Level-order: 58 24 97 12 49 82 55 | |

15. 寫出四個函式

A. Left\_Most

B. Right\_Most

C. Predecessor

D. Successor

|  |  |
| --- | --- |
| 在 Node 結構中新增 Parent 指標 | |
| 1  2  3  4  5  6  7  8 | template<typename T>  struct Node{  int index;  T data;  Node<T> \*left;  Node<T> \*right;  Node<T> \*parent;  } |

插入資料時要代入 parent，因此要修改 Insert 函式：

|  |  |
| --- | --- |
| 修改 Insert 函式 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | template<typename T>  bool Binary\_Tree<T>::Insert(int target, T value){  if (root == nullptr){  root = new Node<T>{target, value, nullptr, nullptr, nullptr};  return true;  }  Node<T>\* current = root;  while (true){  if (current->index == target){  return false;  }  // 往左移動  else if (target < current->index){  if (current->left == nullptr){  // 要插入新的節點為 current 的左節點  // 因此 parent 是 current  current->left = new Node<T>{target, value, nullptr, nullptr,  current};  return true;  }  current = current->left;  // 往右移動  } else {  if (current->right == nullptr){  // 要插入新的節點為 current 的右節點  // 因此 parent 是 current  current->right = new Node<T>{target, value, nullptr,  nullptr, current};  return true;  }  current = current->right;  } //end of if  } // end of while  } // end of Insert |

|  |  |
| --- | --- |
| 新增函式的宣告 | |
| 1  2  3  4  5  6  7  8  9 | template<typename T>  class Binary\_Tree{  public:  ...  Node<T>\* Left\_Most(Node<T>\*);  Node<T>\* Right\_Most(Node<T>\*);  Node<T>\* Predecessor(Node<T>\*);  Node<T>\* Successor(Node<T>\*);  } |

|  |  |
| --- | --- |
| Left Most | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template<typename T>  Node<T>\* Binary\_Tree<T>::Left\_Most(Node<T>\* node){  // 例外處理  if (node == nullptr)  return nullptr;  // 一般情況  // node 不斷往左節點移動  while (node->left != nullptr){  node = node->left;  }  return node;  } |

|  |  |
| --- | --- |
| Right Most | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | template<typename T>  Node<T>\* Binary\_Tree<T>::Right\_Most(Node<T>\* node){  // 例外處理  if (node == nullptr)  return nullptr;  // node 不斷往右節點移動  while (node->right != nullptr){  node = node->right;  }  return node;  } |

|  |  |
| --- | --- |
| Predecessor | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | template<typename T>  Node<T>\* Binary\_Tree<T>::Predecessor(Node<T>\* node){  // 第一種情況：有左子樹  if (node->left != nullptr)  // 回傳左子樹中的 Right Most  return Right\_Most(node->left);  // 第二種情況：沒有左子樹  Node<T>\* current = node->parent;  // 在遇到一條左上到右下（node 是 current->right）的邊前  // 繼續處理  while (current != nullptr && node == current->left){  // node 和 current 都往上移一層  node = current;  current = current->parent;  }  return current;  } |

|  |  |
| --- | --- |
| Successor | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | template<typename T>  Node<T>\* Binary\_Tree<T>::Successor(Node<T>\* node){  // 第一種情況：有右子樹  if (node->right != nullptr)  // 回傳右子樹中的 Left Most  return Left\_Most(node->right);  // 第二種情況：沒有右子樹  Node<T>\* current = node->parent;  // 在遇到一條右上到左下（node 是 current->left）的邊前  // 繼續處理  while (current != nullptr && node == current->right){  // node 和 current 都往上移一層  node = current;  current = current->parent;  }  return current;  } |

|  |  |
| --- | --- |
| 測試 Left most、Right most、Predecessor 與 Successor | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | int main()  {  Binary\_Tree<string> tree;  tree.Insert(58, "A");  tree.Insert(24, "B");  tree.Insert(97, "C");  tree.Insert(12, "D");  tree.Insert(49, "E");  tree.Insert(82, "F");  tree.Insert(55, "G");  cout << "Left most of 58: " << tree.Left\_Most(tree.Search(58))->index  << endl;  cout << "Right most of 58: " << tree.Right\_Most(tree.Search(58))->index  << endl;  cout << "Successor of 58: " << tree.Successor(tree.Search(55))->index  << endl;  cout << "Predecessor of 82: "  << tree.Predecessor(tree.Search(82))->index << endl;  } |
| 執行結果 | |
| Left most of 58: 12  Right most of 58: 97  Successor of 55: 58  Predecessor of 82: 58 | |

下一節介紹二元樹的刪除時，也會用到 Predecessor 與 Successor 函式。

**第四節：二元樹的刪除**

二元搜尋樹的刪除，依據要刪除的節點不同，可以分為三種狀況

A. 該節點沒有任何子節點，即該節點為葉節點：直接刪除即可

B. 該節點有一個子節點：把子節點放到該節點的位置

C. 有兩個子節點：必須另找一個節點來替代

1. 演練二元樹的刪除

(1) 第一種情況：葉節點

上圖中，32 是子節點，因此符合第一種情況。直接刪除 32 這個節點後，讓 38 的左指標改指向空指標，就完成了整個刪除的動作。

(2) 第二種情況：有一個子節點

如果要刪除的 49 這個節點，因為它有一個子節點，所以符合第二種情況。刪除 49 後，把 38 代入 49 原本的位置，也就是說，24 的右指標要改為指向 38。

(3) 第三種情況：有兩個子節點

如果要刪除的是根節點 24，因為 24 有兩個子節點，所以符合第三種狀況。把 24 刪除後，用 32 這個節點代替（選擇 32 的理由待會說明），而不需改變其他節點的次序，就完成了刪除的動作。

2. 選擇刪除後替代的節點

A. 刪除後的二元樹仍必須遵守編號規則

B. 從其右子樹找最小的 node 來取代，即 Successor

C. 從其左子樹找最大的 node 來取代，即 Predecessor

上圖中，刪除 24 後，為何以 24 的 Successor 來代替仍可以滿足編號次序的要求呢？代替 24 的節點必須符合下列條件：該節點比所有 24 的左子樹中節點的編號大，且比 24 所有右子樹中節點的編號小。

這也就是說，要用來代替的節點需要是整棵樹中比 24 小，而不比 24 的左子樹中其它節點小者，或者，也可以是比 24 大，但不比 24 的右子樹中其它節點大者。

觀察這兩個條件，符合者正好就是 24 的 Predecessor 和 Successor。

3. 替代節點的處理步驟

以上圖為例，步驟為

A. 把 D 節點的資料複製給 A 節點

B. 把 D 節點的記憶體釋放掉

C. 把 C 節點原先指向 D 節點的指標設定為空指標

上面這種做法可以使需要改變的指標數量最少。

4. 實作二元搜尋樹的刪除函式

|  |  |
| --- | --- |
| Delete 刪除節點 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | template<typename T>  bool Binary\_Tree<T>::Delete(int target){  // 用 Search 函式找到要刪除的節點  Node<T>\* delete\_node = Search(target);  // 例外處理：要刪除的 target 不在二元樹中  if (delete\_node == nullptr)  return false;  // Case1：刪除葉節點  if (delete\_node->left == nullptr && delete\_node->right == nullptr){  // Case1-1：刪除的是根節點（整棵樹只有一個節點）  if (delete\_node == root){  root = nullptr;  }  // Case1-2：刪除的是父節點的「左節點」  if (delete\_node == delete\_node->parent->left){  // 父節點的左指標改指向 nullptr  delete\_node->parent->left = nullptr;  }  // Case1-3：刪除的是父節點的「右節點」  else {  // 父節點的右指標改指向 nullptr  delete\_node->parent->right = nullptr;  }    // 釋放掉要刪除的指標  delete delete\_node;  }  // Case3：要刪除的節點有兩個子節點  // 先處理的原因是 else if 的條件較好寫  else if (delete\_node->left != nullptr && delete\_node->right != nullptr){  // 取得 successor 中的資料  Node<T>\* successor = Left\_Most(delete\_node->right);  int tmp1 = successor->index;  T tmp2 = successor->data;  // 釋放掉 successor （注意不是釋放要刪除的節點！）  // 調整 successor 的父節點的工作同樣會由 Delete 函式完成  // successor 是 left most，一定是葉節點或只有一個右節點  // 會符合 Case1 或 Case 2，不會造成無窮迴圈  //（有兩個節點的話，可以再往左走，就不是 left most）  Delete(successor->index);  // 把 successor 中的資料複製到 delete\_node 中  delete\_node->index = tmp1;  delete\_node->data = tmp2;  }  // Case2：要刪除的節點有一個子節點  else{  // 宣告一個 child 節點指向要刪除的節點唯一的子節點  Node<T>\* child = (delete\_node->left != nullptr) ?  delete\_node->left : delete\_node->right;  // Case2-1：刪除的是根節點  if (delete\_node == root){  root = child;  }  // Case2-2：要刪除的節點是父節點的左節點  if (delete\_node == delete\_node->parent->left){  // 父節點的左指標改指向 child  delete\_node->parent->left = child;  }  // Case2-3：要刪除的節點是父節點的右節點  else {  // 父節點的右指標改指向 child  delete\_node->parent->right = child;  }  // 釋放掉要刪除的節點  delete delete\_node;  }  // 上面三種情形處理完後都要回傳 true  return true;  } |

|  |  |
| --- | --- |
| 測試 Delete | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | int main()  {  Binary\_Tree<string> tree;  tree.Insert(58, "A");  tree.Insert(24, "B");  tree.Insert(97, "C");  tree.Insert(12, "D");  tree.Insert(49, "E");  tree.Insert(82, "F");  tree.Insert(55, "G");  // 刪除根節點 58  tree.Delete(58);  // 測試 Pre\_order  cout << "Pre-order: "; tree.Pre\_Order(tree.root);  cout << endl;  // 測試 In\_order  cout << "In-order: "; tree.In\_Order(tree.root);  cout << endl;  // 測試 Post\_order  cout << "Post-order: "; tree.Post\_Order(tree.root);  cout << endl;  // 測試 Level\_order  cout << "Level-order: "; tree.Level\_Order(tree.root);  cout << endl;  } |
| 執行結果 | |
| Pre-order: 82 24 12 49 55 97  In-order: 12 24 49 55 82 97  Post-order: 12 55 49 24 97 82  Level-order: 82 24 97 12 49 55 | |

刪除節點前的二元樹：

58

24 97

12 49 82

55

刪除節點後的二元樹：

82

24 97

12 49

55

**第五節：二元樹的基礎題目**

接下來，來做幾題 leetcode 上關於二元樹的基本題目。

這一章是二元樹的「基礎」，下一章則是「應用」，因此本章示範的題目會比較簡單。

二元樹的題目大部分可以用「遞迴」來解，因為每個節點的「左節點形成的左子樹」和「右節點形成的右子樹」，都可以用與處理該節點同樣的函式來處理，前面介紹的前序、中序、後序尋訪就是這樣實作的。

1. LeetCode#226. 反轉二元樹 Invert Binary Tree

A. 題目

給定一個二元樹的根節點，反轉該二元樹，並回傳根節點。

B. 出處：https://leetcode.com/problems/invert-binary-tree/

C. 說明

給定一個二元樹的根節點，請把整個二元樹「反轉」，處理完成後，再回傳該根節點。

一開始二元樹如果像上圖，Level-Traversal 的順序是 4271369，那麼反轉後，順序會是 4729631。

對於根節點 4 而言，左右節點 2 和 7 要對調，2 變成右節點，7 變成左節點。對於 7 而言，左右節點 9 和 6 同樣要對調，6 變為右節點，9 變為左節點。

就像這樣，本題只要把每個節點的左右節點對調即可。

|  |  |
| --- | --- |
| 反轉二元樹 Invert Binary Tree | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Solution{  public:  TreeNode\* invertTree(TreeNode\* root){  // 例外處理：樹是空的  if (root == nullptr)  return root;  // 一般情形  // 把左節點和右節點互換  TreeNode\* temp = root->left;  root->left = root->right;  root->right=temp;  // 遞迴處理左子樹和右子樹  invertTree(root->left);  invertTree(root->right);  // 回傳根節點  return root;  } //end of invertTree  } // end of class Solution |

D. 評論

本題背後有個有趣的故事：如果讀者使用的是 mac 電腦，可能有聽過 Homebrew 軟體。

Homebrew 軟體的開發者 Max Howell 曾經被邀請到 Google 面試，面試時就被考到的這題「反轉二元樹」，結果他因為沒有做出這題，沒有被錄取。

他隨後覺得這是很不可思議的情況，因此在推特上寫下自己的諷刺：

Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so f\*\*\* off.

Google：我們 90% 的工程師都使用你開發的 Homebrew 軟體，但是你沒辦法在白板上反轉二元樹，所以滾開。

這引發了軟體工程圈相當大的討論：用對資料結構和演算法的熟悉度來篩選面試者，究竟是不是好的做法？這見仁見智。

如果面試者是剛畢業的學生，那要求大型軟體的開發經驗幾乎是不可能的，因此也只能考察學生對資料結構與演算法的熟悉程度，因為這兩個科目某種程度反映了面試者在軟體開發上的潛力，只不過即使無法回答出相關問題，也不代表一定之後就無法開發出優秀的軟體。

因為每個學生在校時選修的應用課程都不同，有專心在 AI、ML 上的，也有關注前端開發的，因此設計面試題目就變得很不容易。此時，用資料結構和演算法來觀察面試者基本的數學邏輯運算能力，仍可以說是一個不錯的方式，畢竟數學和邏輯的基礎好，許多開發框架可以快速上手，相對的，數學和邏輯基礎不好，則即使錄取，之後可能也會比較辛苦。

總結來說，如果面試者已經從事軟體開發十幾二十年，那麼面試時大可以參考作品集來判斷能力；對於剛畢業的學生，則考資料結構與演算法的相關題目，仍是較為合理的。

2. LeetCode#94. 二元樹的中序尋訪 Binary Tree Inorder Traversal

A. 題目

給定一個二元樹的根節點，請按照中序尋訪的規則列出節點順序。

B. 出處：https://leetcode.com/problems/binary-tree-inorder-traversal/

C. 輸入與目標輸出

a. 輸入：root = [1,null,2,3]

b. 輸出：[1,3,2]

|  |  |
| --- | --- |
| 二元樹的中序尋訪 Binary Tree Inorder Traversal | |
| 1  2  3  4  5  6  7  8  9  10  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class Solution{  public:  vector<int> inorderTraversal(TreeNode\* root){  // 宣告向量 result 來存放結果  vector<int> result;    // 呼叫 inorder 函式  // 因為使用傳參考呼叫，result 的值會被函式改變  inorder(root, result);  return result;  }  // path 是 inorder 的順序  void inorder(TreeNode\* current, vector<int>& path){  // 例外處理：樹是空的  if (current == nullptr)  return ;    // 一般情形  // 處理左子樹  inorder(current->left, path);  // 處理 current 節點path.push\_back(current->val);  // 處理右子樹  inorder(current->right, path);  }  }; |

3. LeetCode#450. 刪除二元搜尋樹的節點 Delete Node in a BST

A. 題目

給定一個二元搜尋樹的根節點與 key 值，把該值對應的節點刪除，並回傳新的根節點（可能與原本不同）。

刪除的步驟可以分為兩部分：

a. 搜尋該節點

b. 如果找到該節點，進行刪除

B. 出處：https://leetcode.com/problems/delete-node-in-a-bst/

|  |  |
| --- | --- |
| 刪除二元搜尋樹的節點 Delete Node in a BST | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | class Solution{  public:  // 先寫出刪除時會用到的 Left\_Most 函式  TreeNode\* Left\_Most(TreeNode\* root){  if (root == nullptr)  return nullptr;  while (root->left != nullptr){  root = root->left;  }  return root;  }  TreeNode\* deleteNode(TreeNode\* root, int key){  // 例外處理：樹是空的  if (root == nullptr)  return nullptr;  // 本題要手動取得父節點  TreeNode\* parent = root;  // 要回傳的是新的根節點  TreeNode\* new\_root = root;  // 一般情形  // 找到 key 的位置，即 Search 函式  while (key != root->val){  // 往子節點移動之前，先把目前節點賦值給 parent  parent = root;  if (key < root->val){  root = root->left;  } else {  root=root->right;  }  }  // 如果 root 仍是空的，代表沒有找到  // 因為此時不做處理，所以回傳直接回傳原本的根節點  // 這裡 new\_root 是移動前的根節點  if (root == nullptr)  return new\_root;  // Case1：要刪除的是葉節點  if (root->left == nullptr && root->right == nullptr){  // 例外處理：要刪除的是根節點  // 整個二元樹只有一個節點  if (root==new\_root)  new\_root = nullptr;    // 修改父節點指標  if (root == parent->left){  parent->left = nullptr;  } else {  parent->right = nullptr;  }  // Case3：有兩個子節點  } else if (root->left != nullptr && root->right != nullptr){  // 取得要刪除節點的 successor  TreeNode\* successor = Left\_Most(root->right);  // 用 successor 來取代 root，先複製內容  int value = successor->val;  // 把該 successor 的節點刪除  deleteNode(root, successor->val);  // 把 successor 的內容複製到要刪除的節點中  root->val = value;    // Case2：只有一個子節點  } else {  // 判斷子節點是左節點還是右節點，並取得子節點  TreeNode\* child = root->left == nullptr? root->right:root->left;  // 要刪除的是根節點的話，刪除後新的根節點是 child  if (root==new\_root)  new\_root = child;  // 修改要刪除節點的父節點的子節點  // 以要刪除節點的 child 代替  if (root==parent->left){  parent->left = child;  } else {  parent->right = child;  }  }  // 回傳新的根節點  return new\_root;  } // end of deleteNode  }; // end of class Solution |

4. LeetCode#543. 二元樹的直徑 Diameter of Binary Tree

A. 題目

給定一個二元樹的根節點，回傳該二元樹中最大的「直徑」。

「直徑」定義為該二元樹中任兩個節點間「最長的路徑」，路徑可能經過根節點，也可能不經過。

路徑的「長度」則定義為該路徑經過的邊數。

B. 出處：https://leetcode.com/problems/diameter-of-binary-tree/

C. 說明

上圖中，從節點 3 到節點 4 的路徑是 [3,1,2,4]，長度是 3。從節點 3 到節點 5 的路徑是 [3,1,2,5]，長度同樣是 3，因為樹中沒有兩個節點間的路徑更長，因此回傳 3。

D. 解題邏輯

最長的路徑一定是某個節點「左子樹的高度」加上「右子樹的高度」，在範例中，根節點 1 的左子樹高度是 2，右子樹高度是 1，最長路徑是 2+1=3。

不過該路徑不一定會經過根節點，因此要依序取得每個節點的「左子樹與右子樹高度相加」的值，並取其中最大者。

|  |  |
| --- | --- |
| 二元樹的直徑 Diameter of Binary Tree | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | class Solution{  public:  int diameterOfBinaryTree(TreeNode\* root){  int max = 0;  calculate(root, max);  return max;  }  // 計算對於樹中所有節點來說「左子樹與右子樹高度相加」  // 最大者的值  int calculate(TreeNode\* root, int &max){  // 傳入的是空指標（發生在處理葉節點的子節點時）  if (root==nullptr)  return 0;  // 遞迴處理左節點與右節點  // 回傳往左節點方向走到底的最長邊數  int max\_left = calculate(root->left, max);  // 回傳往右節點方向走到底的最長邊數  int max\_right = calculate(root->right, max);  // max 存放比較後發現的最大值  // 把當前的 max 和  max = max>max\_left+max\_right ? max : max\_left+max\_right;  // 葉節點 3、4、5 計算出的高度和是 0  // 這些葉節點的 max\_left 和 max\_right 都會回傳 0  // 處理節點 2 時，max\_left 和 max\_right 分別回傳 1、1  // 因此高度和是 2；回傳的max\_height+1 = 2  // 代表對於父節點的 1 來說  // 往節點 2 走的最長路徑是 2（[1,2,4] 或 [1,2,5]）  // 處理節點 1 時，目前的 max 是處理節點 2 時得到的 2  // max\_left 是節點 2 回傳的左邊最長路徑 2  // max\_right 則是節點 3 回傳的右邊最長路徑 1  // 因此高度和是 2+1=3  // 因為這個高度和 3 比先前得到的 max = 2 還要大  // 因此 max 被取代為 3  int max\_height = max\_left>max\_right?max\_left:max\_right;  // 回傳的是父節點往這個方向走的最長邊數  // 因為往父節點還有一條邊，要加上 1  // 最後處理根節點 1 時，這個回傳值不會用到  // 用到的是用傳參考呼叫傳入而修改的 max  return max\_height+1;  } // end of diameterOfBinaryTree  }; // end of class Solution |