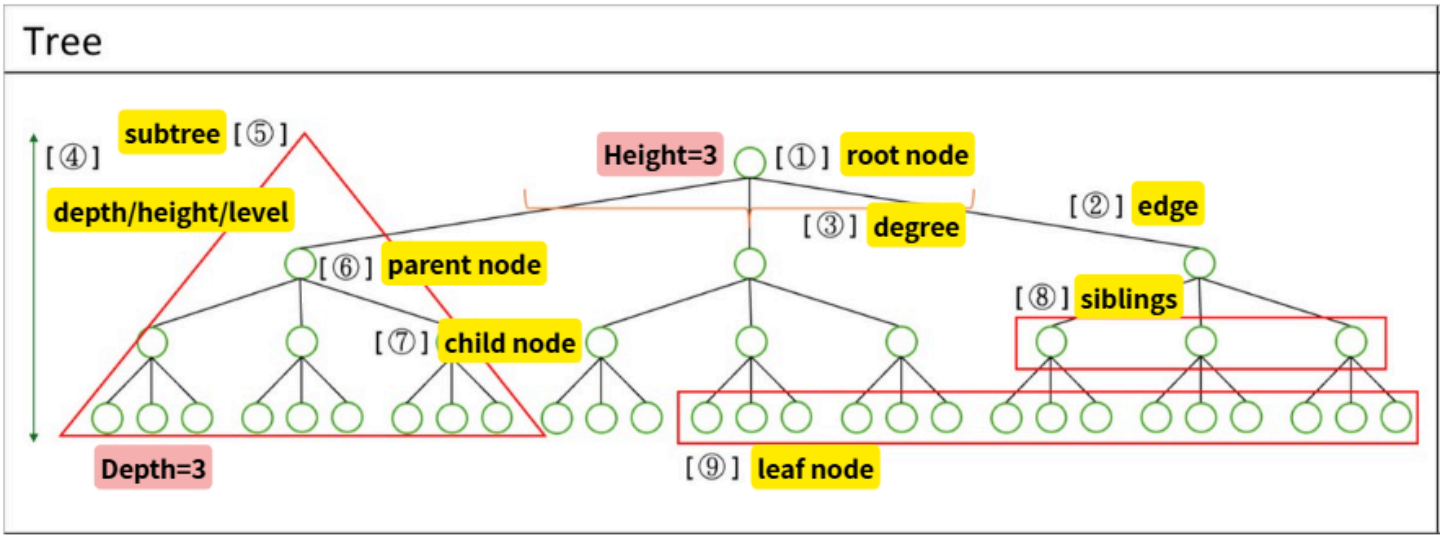
Tree

一、核心概念

樹是一種非線性、具階層性的資料結構。

Terminology



- Depth : 深度是從根節點到該節點的距離

- Height : 高度是從該節點到最遠葉節點的最長路徑

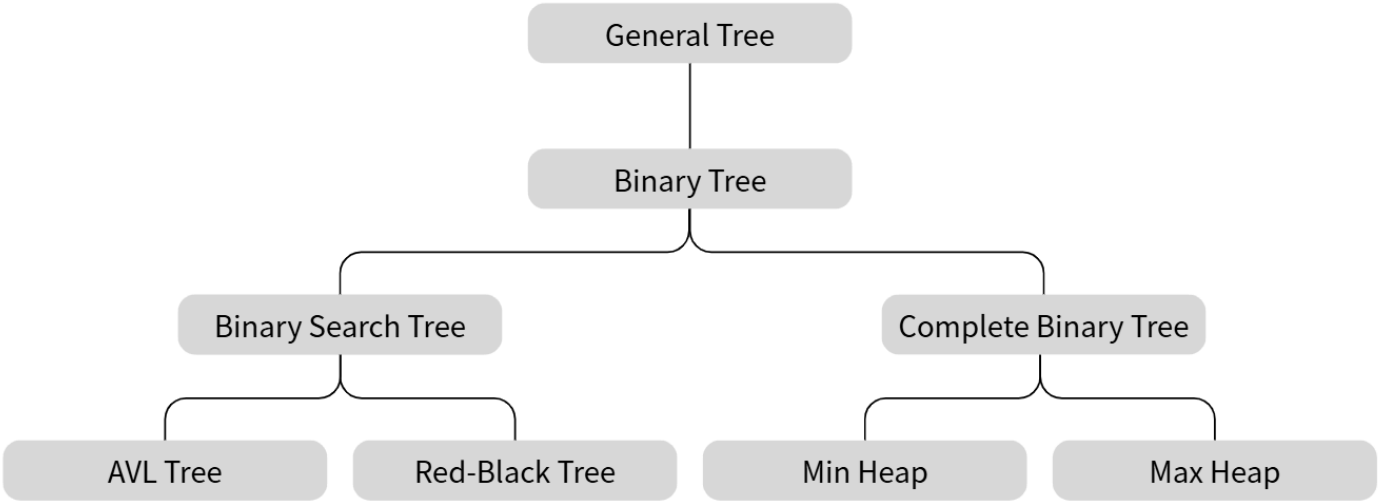
- Degree : 一個節點擁有的子節點數量

- Edge : 若節點數為 N，則邊數必為 N−1

Binary Tree ADT

- Create(): 建立空二元樹
- IsEmpty(bt): 檢查是否為空
- MakeBT(bt1, item, bt2): 以 item 為根，bt1 為左子樹，bt2 為右子樹組合新樹
- Lchild(bt) / Rchild(bt): 回傳左 / 右子樹
- Data(bt): 回傳根節點的資料

二、樹的演化



General Tree : 一種最基礎的非線性階層結構，由一組有限的節點組成（至少包含一個根節點）。在此結構中，每個節點可以延伸出任意數量的子節點，且子節點之間沒有強制的排序規則。

Binary Tree : 一種受限的樹狀結構，每個節點最多只能有兩個子節點（0,1或2個）。子節點被嚴格區分為「左子節點」和「右子節點」。(Type : Strict / Complete / Degenerate / Perfect)

→ 檔案系統或組織架構圖，利用節點間的階層分支特性，能自然地呈現資料夾嵌套或組織內「一對多」的上下級歸屬關係 。

Shaped-based binary tree (Fill : node -> 0,2 node / Perfect : node -> 0,2 node and same depth)

Complete Binary Tree : 一棵高度為 h 的 Binary tree，除第h層外，其他各層節點數都達到最大值，且第 h 層的節點都連續集中在最左邊。

→ 二元堆積的陣列實作 ，由於節點緊密排列無空洞，適合用陣列儲存並以索引公式（2i, 2i+1）定位子節點，省去指標空間且便於計算。

Criteria-based binary tree

Binary Search Tree (BST) : 一棵 Binary tree，對任意節點皆滿足：左子樹所有節點的值 < 根節點的值，右子樹所有節點的值 > 根節點的值。

→ 字典查詢、編譯器中的符號表，利用「左小右大」的排序特性，在平均情況下能以 O(log n) 的速度完成資料的搜尋、插入與刪除。

AVL Tree : 一種自我平衡的 Binary search tree (BST)，對任一節點的左子樹與右子樹的高度差，其絕對值不能超過 1（即只能是 -1, 0, 1）。

→ 讀取頻繁的資料庫索引，藉由嚴格限制左右子樹的高度差，確保搜尋路徑永遠維持在最短範圍（log n），從而提供最穩定且極速的查詢效能。

Red-Black Tree : 一種自我平衡的 Binary search tree (BST)，節點帶有顏色（紅或黑），需滿足：根是黑色、不能有兩個連續的紅色節點、從任一節點到其所有後代 Null 節點的黑色節點數量必須相同。

→ C++ STL 的 std::map 與 std::set，採用較寬鬆的平衡規則以減少插入與刪除時的旋轉次數，在搜尋與更新頻繁變動的情境下展現穩定的綜合效能 。

Max Heap：一棵 Complete Binary Tree，滿足 Heap Property：父節點的值 ≥ 子節點的值根，因此根節點永遠是最大的。

→ 作業系統的行程排程，確保優先權最高（數值最大）的任務永遠位於根節點，讓 CPU 能以 O(1) 時間立即取得下一個待執行的行程。

Min Heap：一棵 Complete Binary Tree，滿足 Heap Property：父節點的值 ≤ 子節點的值，因此根節點永遠是最小的。

→ Dijkstra 最短路徑演算法，協助演算法在每一輪擴展路徑時，能最有效率地從未訪問節點中找出距離起點最短（數值最小）的節點 。

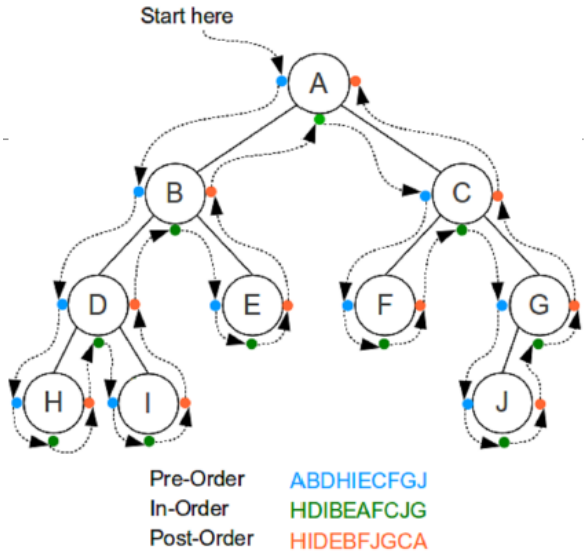
From	To	New property / constraint added
General Tree	Binary Tree	**限制子節點數量：**每個節點最多只能有2個子節點。
Binary Tree	Complete Binary Tree	**結構形狀限制：**樹必須從上到下、從左到右填滿。除了最後一層外，每一層都必須是滿的，且最後一層節點靠左排列。
Complete Binary Tree	Max Heap	**父子大小限制+結構限制：**必須是Complete Binary Tree，且父節點的值必須大於或等於子節點。
Complete Binary Tree	Min Heap	**父子大小限制+結構限制：**必須是Complete Binary Tree，且父節點的值必須小於或等於子節點。
Binary Tree	BST	**數值順序限制：**對於任一節點，左子樹的所有值都小於根節點，右子樹的所有值都大於根節點。
BST	AVL Tree	**嚴格高度平衡：**增加平衡因子限制。任一節點的左右子樹高度差絕對值不能超過1(即-1,0,1)。
BST	Red-Black Tree	**著色與規則限制：**節點分為紅或黑，透過顏色規則（根是黑、無連續紅節點、路徑黑節點數相同）來維持大致平衡。

三、樹的走訪（Traversal）

- 深度優先搜尋 (DFS) -> O(n)

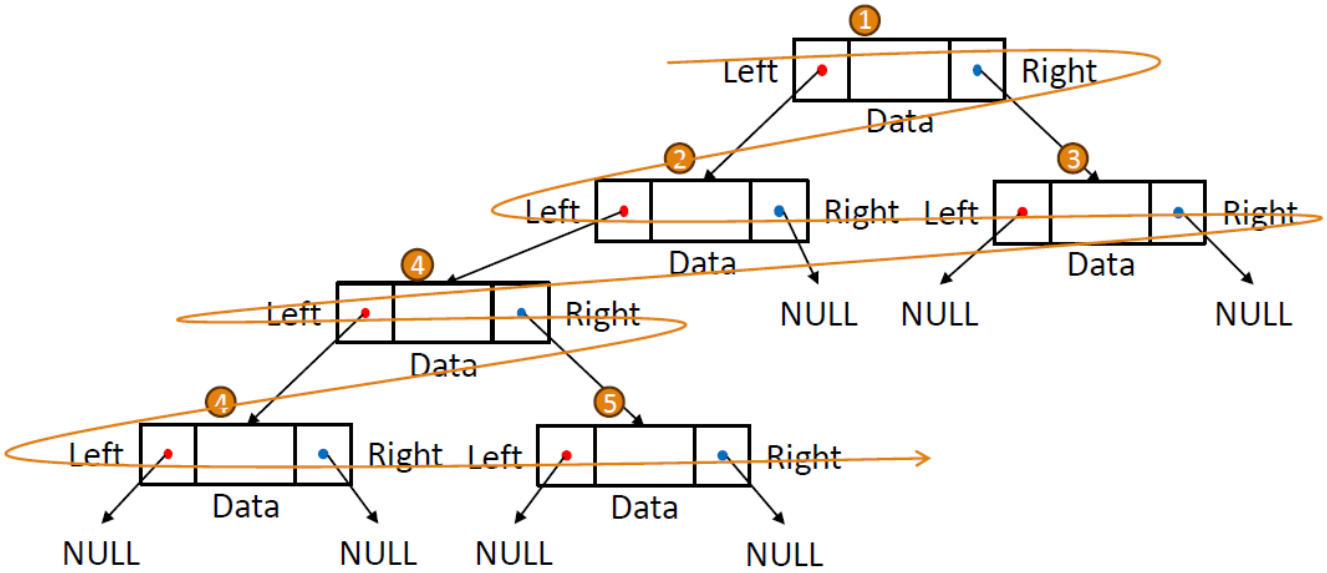
1. 前序 (Preorder): 根 -> 左 -> 右，常用於複製樹。
2. 中序 (Inorder): 左 → 根 -> 右，在 BST 中會得到排序好的序列。
3. 後序 (Postorder): 左 -> 右 -> 根，常用於刪除樹（先刪子節點再刪父節點）。

- 使用 **Stack** 或遞迴



- 廣度優先搜尋 (BFS) / Level Order Traversal -> O(n)

- 利用佇列 (Queue) 實作，一層一層由左向右訪問 。
- 重要觀念： 在一般圖形中，BFS 從根出發時，同一層級的鄰居走訪順序可以有多種排列



四、Balanced與效能

- **關鍵屬性：** 樹的高度決定了操作效能。
- **平衡時 O(log n):** 搜尋像二分搜尋一樣快。
- **退化時 O(n):** 當輸入已排序資料，BST 會變成**鏈結串列 (Degenerate/Skewed)** 。

Q : BST 在什麼情況下搜尋效率最差？ A : 當輸入資料已經由小到大或由大到小排序好時,因為高度退化為 n

進階平衡策略

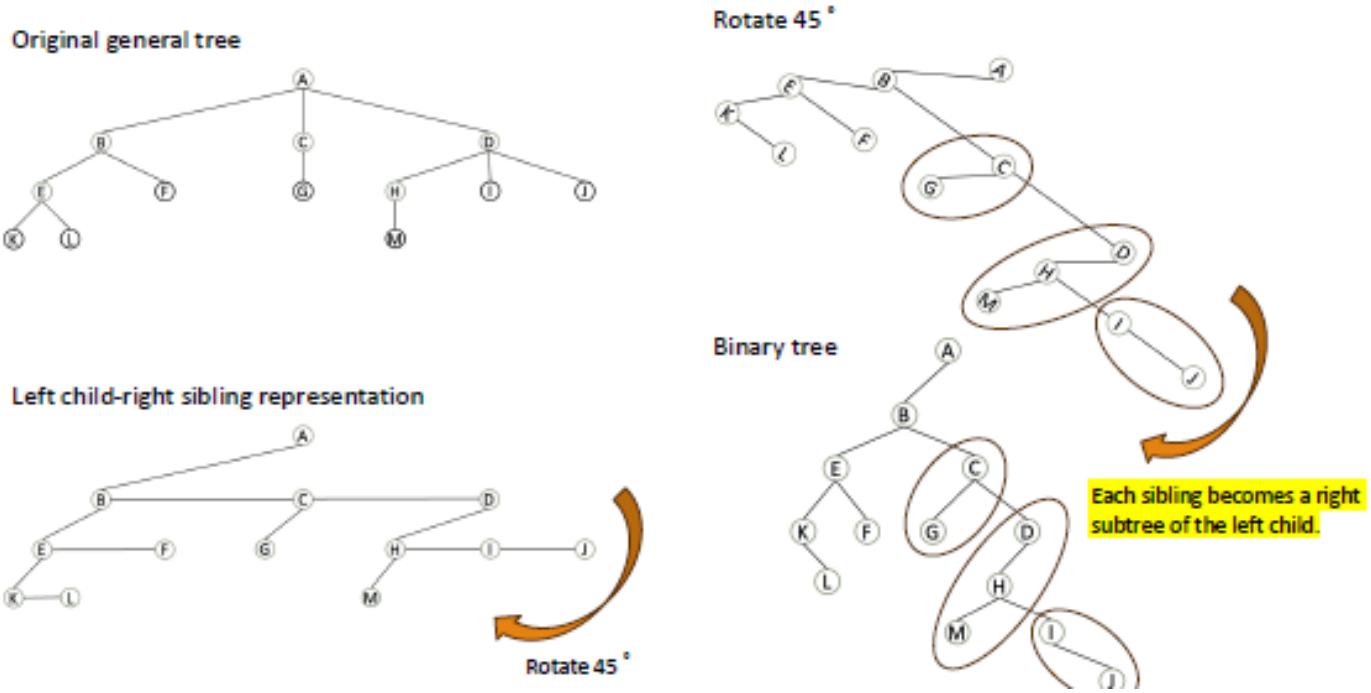
結構	平衡策略	效能取捨
AVL Tree	嚴格高度差	搜尋最快，但插入刪除時旋轉頻繁 。
Red-Black Tree	顏色約束	搜尋略慢於 AVL，但插入刪除時旋轉較少 。
排序陣列	靜態結構	適合**不變動資料**的搜尋，快取局部性 (Cache Locality) 最佳 。

五、 實作與應用精要

樹的轉化 (Transformation) : 一般樹 -> 二元樹

如何將一般樹轉化為二元樹？口訣：「**左兒子，右兄弟**」

1. 連結所有同層的兄弟節點。
2. 對每個節點，只保留與「第一個孩子」的連結，切斷與其餘孩子的連結。
3. 旋轉 45 度，原本的兄弟連結就會變成右子樹。



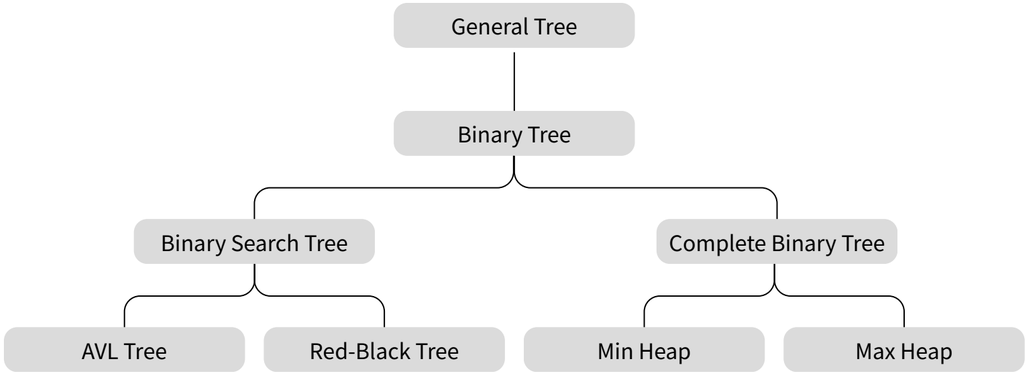
BST 刪除操作 (Delete)

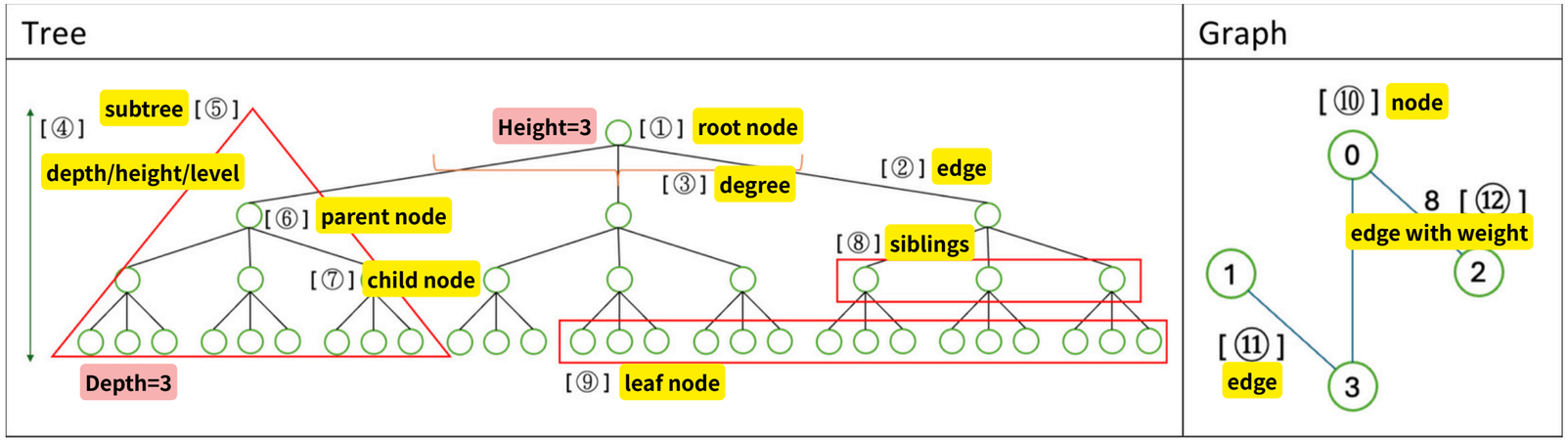
- 刪除葉節點： 直接刪除。
- 刪除有一個子節點： 將子節點上移取代父節點位置。
- 刪除有兩個子節點： 尋找**左子樹最大值**或 **右子樹最小值**來替換 。

陣列表示法索引公式 (用於 Complete Tree/Heap)

若根節點位於索引 i=1：

- 左子節點： 2i
- 右子節點： 2i+1
- 父節點： floor (i/2)


General Tree
Binary Tree
Binary Search Tree
Complete Binary Tree
AVL Tree
Red-Black Tree
Min Heap
Max Heap


Tree
Graph
[④] subtree [⑤]
depth/height/level
Height=3
[①] root node
[②] edge
[③] degree
[⑥] parent node
[⑦] child node
[⑧] siblings
Depth=3
[⑨] leaf node
[⑩] node
0
8 [⑫]
edge with weight
2
1
[⑪]
edge
3