Tree

上課補充 by erichung

Credit by PixelCat

Sprout

二元搜尋樹

Sprout

二元搜尋樹

「二元搜尋樹」（Binary Search Tree，BST）
是一種特別的樹，在資料結構中佔有重要的地位

二元：一棵二元樹
搜尋：你可以在樹上搜尋（???）

Sprout

二元搜尋樹

讓二元樹上每個節點都存一個值，而且對於每個節點：

- 他的左子樹內，所有節點的值都 $\leq$ 他的值
- 他的右子樹內，所有節點的值都 $\geq$ 他的值
- 他的左右子樹分別都是 BST

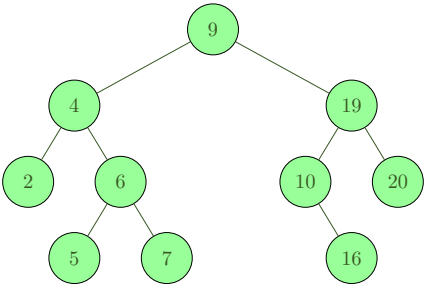
$\Rightarrow$ 二元搜尋樹的**中序遍歷**是從小到大排序好的

二元搜尋樹

我們希望能一邊維護二元搜尋樹的性質，一邊支援：

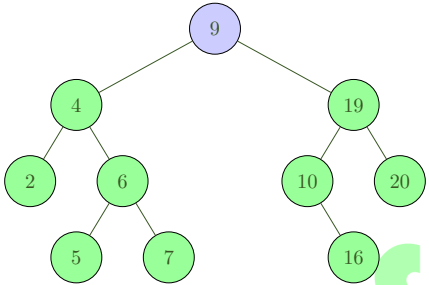
- 查詢數字 x 有沒有在樹上
- 加一個數字 x 到樹上
- 從樹上刪掉一個數字 x

二元搜尋樹：範例



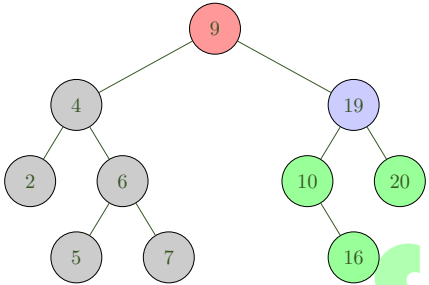
二元搜尋樹：查詢

查詢 10 有沒有在樹上？



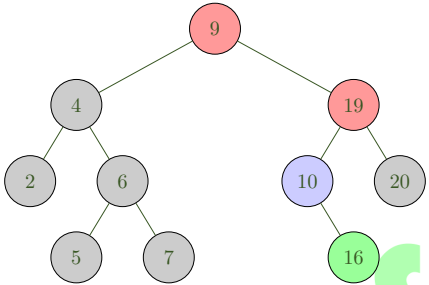
二元搜尋樹：查詢

查詢 10 有沒有在樹上？



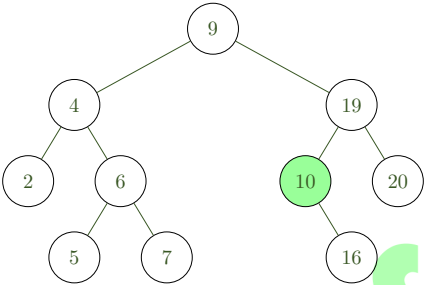
二元搜尋樹：查詢

查詢 10 有沒有在樹上？



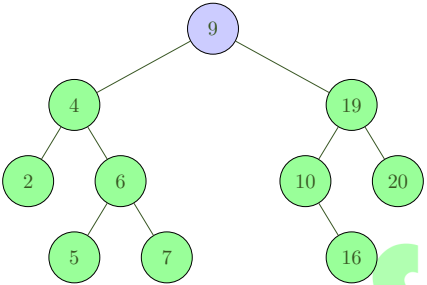
二元搜尋樹：查詢

好欸！找到了！



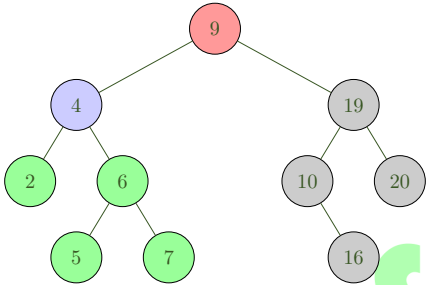
二元搜尋樹：查詢

查詢 8 有沒有在樹上？



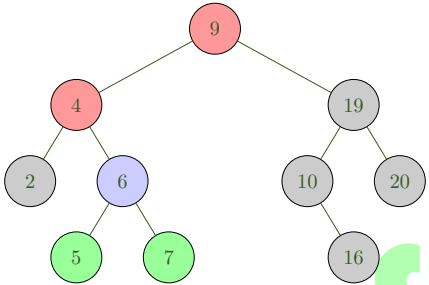
二元搜尋樹：查詢

查詢 8 有沒有在樹上？



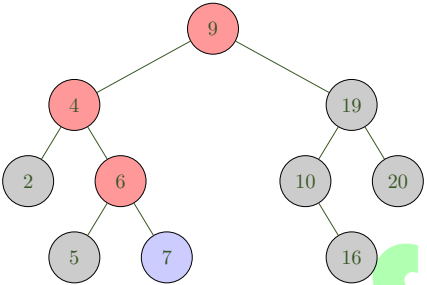
二元搜尋樹：查詢

查詢 8 有沒有在樹上？



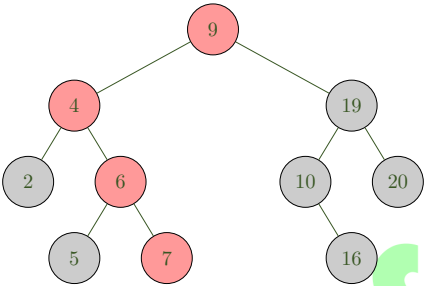
二元搜尋樹：查詢

查詢 8 有沒有在樹上？



二元搜尋樹：查詢

找不到 QQ



二元搜尋樹：查詢

回憶二元搜尋樹的性質「每個節點的左右子樹分別都是 BST」

用遞迴的想法：

- 如果根節點就是要找的數字，那就是找到了
- 如果找到沒得找了還沒找到，那就是找不到
- 否則繼續找，拿根和目標數字比，決定往左還是往右

怎麼覺得有點像二分搜？

Sprout

二元搜尋樹：查詢

回憶二元搜尋樹的性質「每個節點的左右子樹分別都是 BST」

用遞迴的想法：

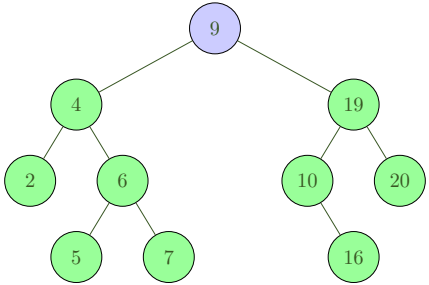
- 如果根節點就是要找的數字，那就是找到了
- 如果找到沒得找了還沒找到，那就是找不到
- 否則繼續找，拿根和目標數字比，決定往左還是往右

怎麼覺得有點像二分搜？

再回憶性質「BST 的中序遍歷是從小到大排序好的」

二元搜尋樹：查詢

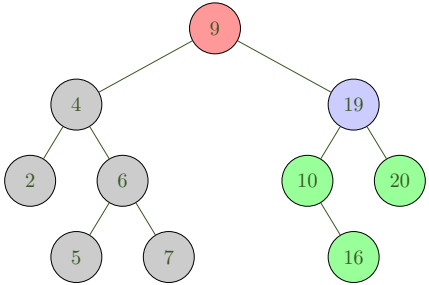
查詢 10 有沒有在樹上？



2	4	5	6	7	9	10	16	19	20

二元搜尋樹：查詢

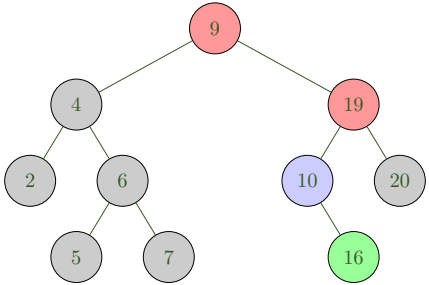
查詢 10 有沒有在樹上？

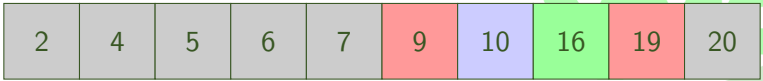


2	4	5	6	7	9	10	16	19	20

二元搜尋樹：查詢

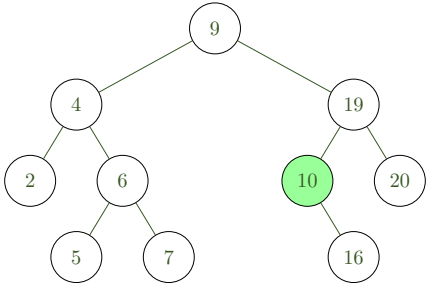
查詢 10 有沒有在樹上？





二元搜尋樹：查詢

好欸！找到了！



2	4	5	6	7	9	10	16	19	20

二元搜尋樹：插入

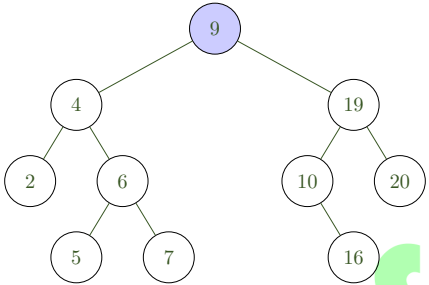
一般陣列不能隨便插入，但二元搜尋樹可以

對於一個新節點，我們讓他當葉子，找個地方接在最下面

Sprout

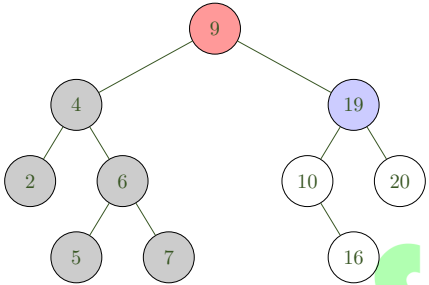
二元搜尋樹：插入

插入節點 10



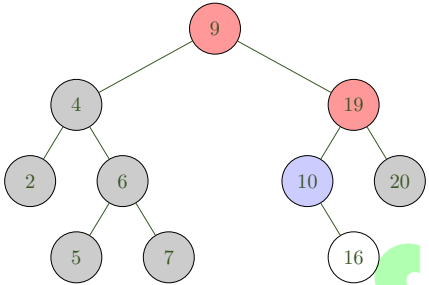
二元搜尋樹：插入

插入節點 10



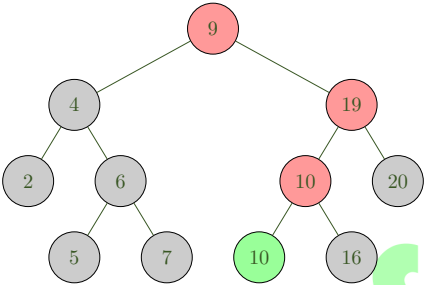
二元搜尋樹：插入

插入節點 10



二元搜尋樹：插入

插入節點 10



二元搜尋樹：刪除

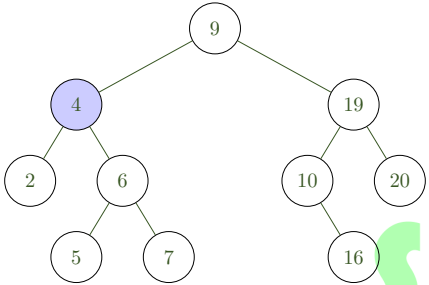
一般陣列不能隨便刪除，但二元搜尋樹可以

第一階段：找到要刪掉的節點在哪
第二階段：刪掉他
第三階段：找個人把洞補起來

Sprout

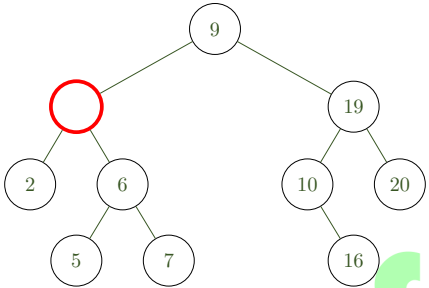
二元搜尋樹：插入

刪除節點 4
第一階段：找到要刪掉的節點



二元搜尋樹：插入

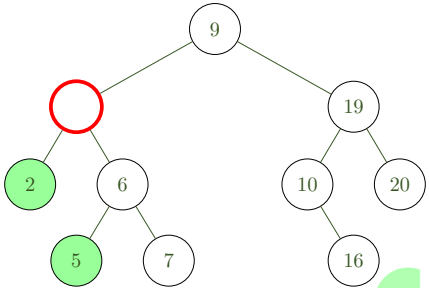
刪除節點 4
第二階段：刪掉他



樹上出現一個「洞」

二元搜尋樹：插入

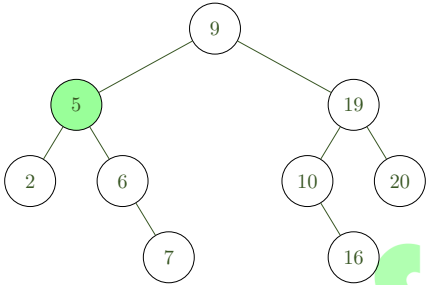
刪除節點 4
第三階段：找個人把洞補起來



用「左子樹最右邊的節點」或「右子樹最左邊的節點」來補
沒子樹怎麼辦？葉子可以直接拔掉

二元搜尋樹：插入

刪除節點 4



二元搜尋樹：時間複雜度

搜尋、插入、刪除，都是每回合需要 $O(1)$ 時間，從根出發每做完一個回合都往更深的節點走

時間複雜度都是 O(樹的深度)

二元搜尋樹：時間複雜度

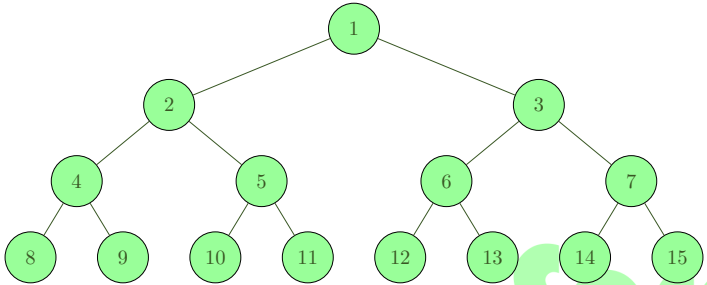
搜尋、插入、刪除，都是每回合需要 $O(1)$ 時間，從根出發每做完一個回合都往更深的節點走

時間複雜度都是 O(樹的深度)

二元搜尋樹能有多深？

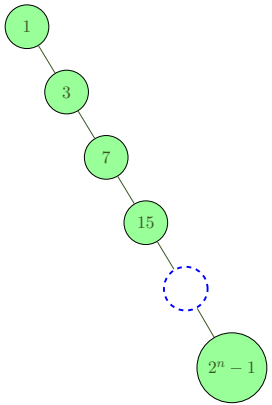
二元搜尋樹：時間複雜度

善良的二元樹



二元搜尋樹：時間複雜度

邪惡的二元樹



二元搜尋樹：時間複雜度

二元搜尋樹能有多深？

理想上「平衡的」二元搜尋樹深度應該要只有 $O(\log N)$
但是最差甚至會到 $O(N)$

Sprout

二元搜尋樹：時間複雜度

二元搜尋樹能有多深？

理想上「平衡的」二元搜尋樹深度應該要只有 $O(\log N)$
但是最差甚至會到 $O(N)$

所以對於一般二元搜尋樹，搜尋、插入、刪除的複雜度都是最糟 $O(N)$

Sprout

二元搜尋樹：時間複雜度

要怎麼禁止二元搜尋樹不平衡？

- B-tree
- AVL tree
- 紅黑樹
- Treap
- Splay
- ...etc.

都好難好難，今天不會教

Sprout

指標與偽指標

指標

以二元樹為例，我們需要存左右子節點的指標

但是好像很多人討厭指標？

Sprout

指標

```
struct Node {
    Node *l, *r;
    int data;
};

Node *root = new Node();
cout << root->data << "\n";
delete root;  // 不釋放記憶體會怎樣嗎？
```

偽指標

```
struct Node {
    int l, r;
    int data;
};
Node nodes[MAX_NODE];  // 0 號節點代表空指標
int _nodes_count = 0;

int root = ++_nodes_count;
cout << nodes[root].data << "\n";
// 不需要釋放記憶體
```

指標 v.s. 偽指標

指標：

- 用**指標**代表每個節點
- 程式碼美觀
- 不用事先決定要開多少節點

偽指標：

- 用**編號**代表每個節點
- 方便 debug、輸出節點長相
- (一定程度上) 避免戳到空指標原地 RE
- 不需要動態分配記憶體，比 new/delete 快一點點
- 整數比指標不佔空間

聰明的指標

```cpp
struct Node {
    std::shared_ptr<Node> l, r;
    int data;
};

std::shared_ptr<Node> root = make_shared<Node>();
cout << root->data << "\n";
// 不需要特別釋放記憶體！
```

Sprout

聰明的指標

- C++ smart pointers (C++ 11 或更新版本)
- 用法跟一般指標完全相同
- 自動回收記憶體避免 memory leak
- 競程幾乎（完全）沒人用

Sprout