B06505032

楊千瑩

一、BST設計方法及實作：

1. BSTreeNode 中記錄了 node的parent, left, right

紀錄parent的目的是為了在successor及predecessor中方便操作。

1. Class BSTree

2.1 首先，在Binary Tree的constructor中，建立root 及dummy node。

並運用內部class iterator 達到traversal的操作

2.2 class iterator 中的function中

運用了successor 尋找下一個node以及predecessor 尋找上一個node，意即實作出iterator ++及iterator--。

* 1. BSTree中加入了一些helper function: insertHelper, findHelper, deleteHelper以及leftmost, rightmost, successor。

**Leftmost**: 尋找以current為root之subtree中，最左邊的node。

**Rightmost**: 從「以CurrentNode為subtree」的root一路向右做Linked list的單向traversal。

**Successor**: inorder traversal中找到下一個node。

**insertHelper**:

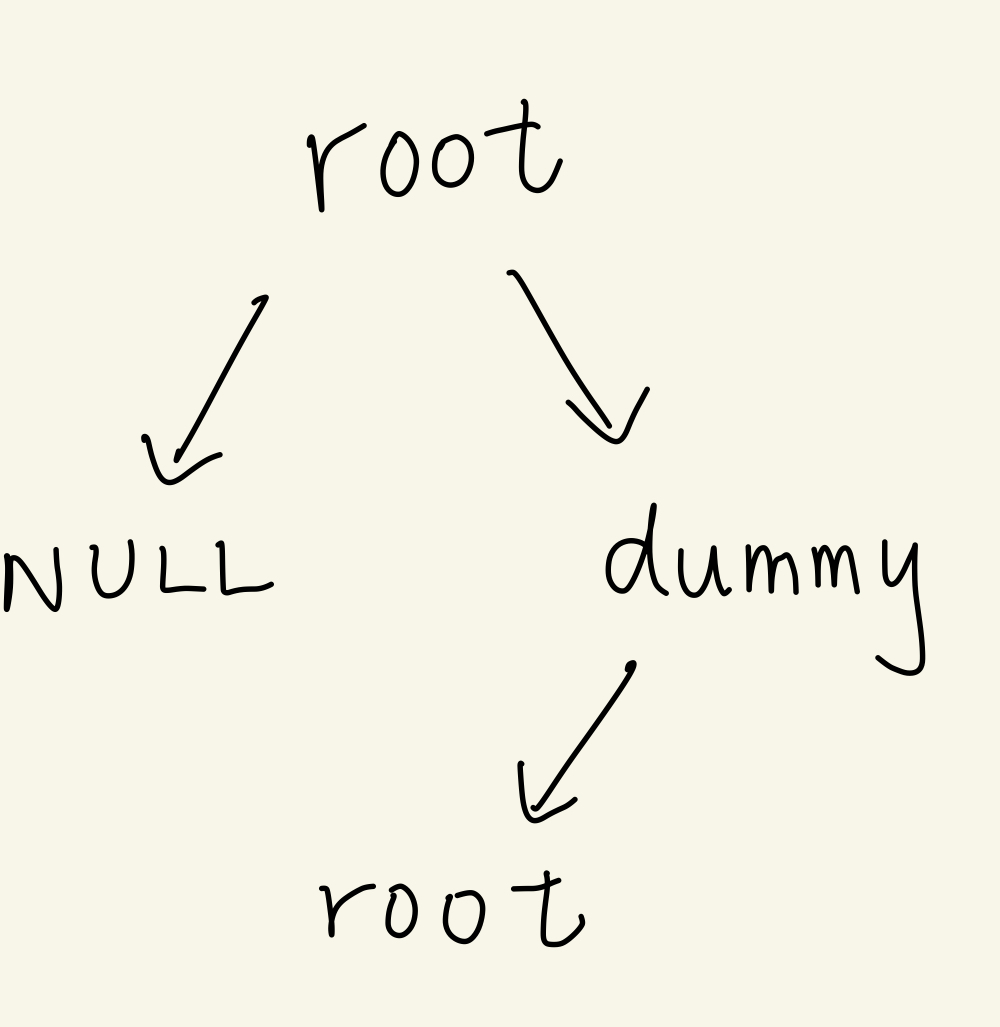
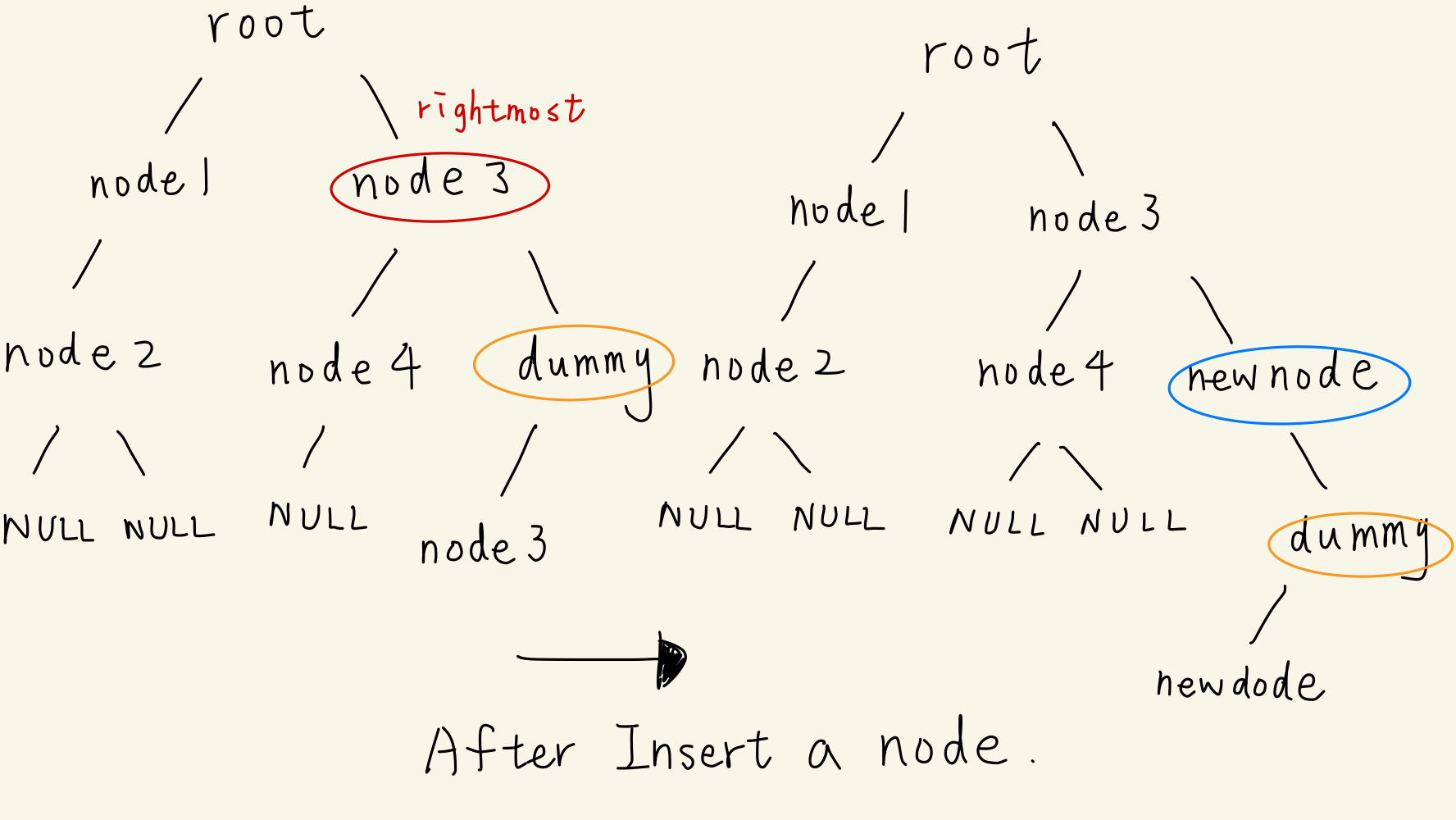
建立第一個node時，將此node設為root，將root->right設為dummy, 並將dummy->left設為root。

之後insert新的node時，若建立的位置為dummy，會把newnode->right設為dummy, dummy->left設為newnode。

如此一來，每做一次insert後，rightmost的rightchild會接至dummy，而dummy的leftchild會接到rightmost。

設計rightmost接至dummy的目的為使iterator在做traversal時能夠順利尋找至最後一個node。

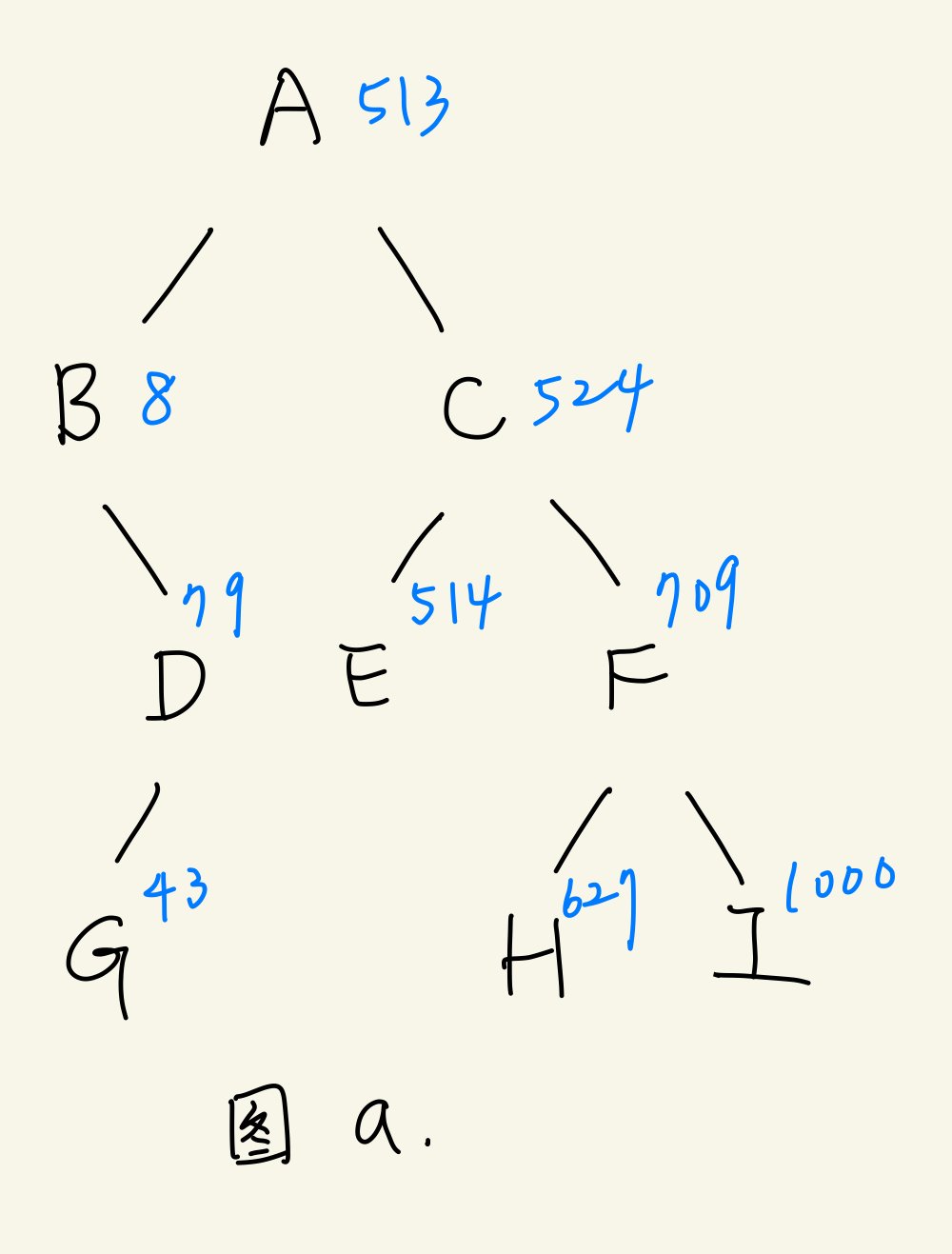
下圖為範例：

**findHelper**: 根據BST的特徵：data(L)<data(Current)<data(R)，判斷Current應該往left subtree走，還是往right subtree走。

搜尋結果可能成功，可能失敗，以下便分別以兩個data值作說明。

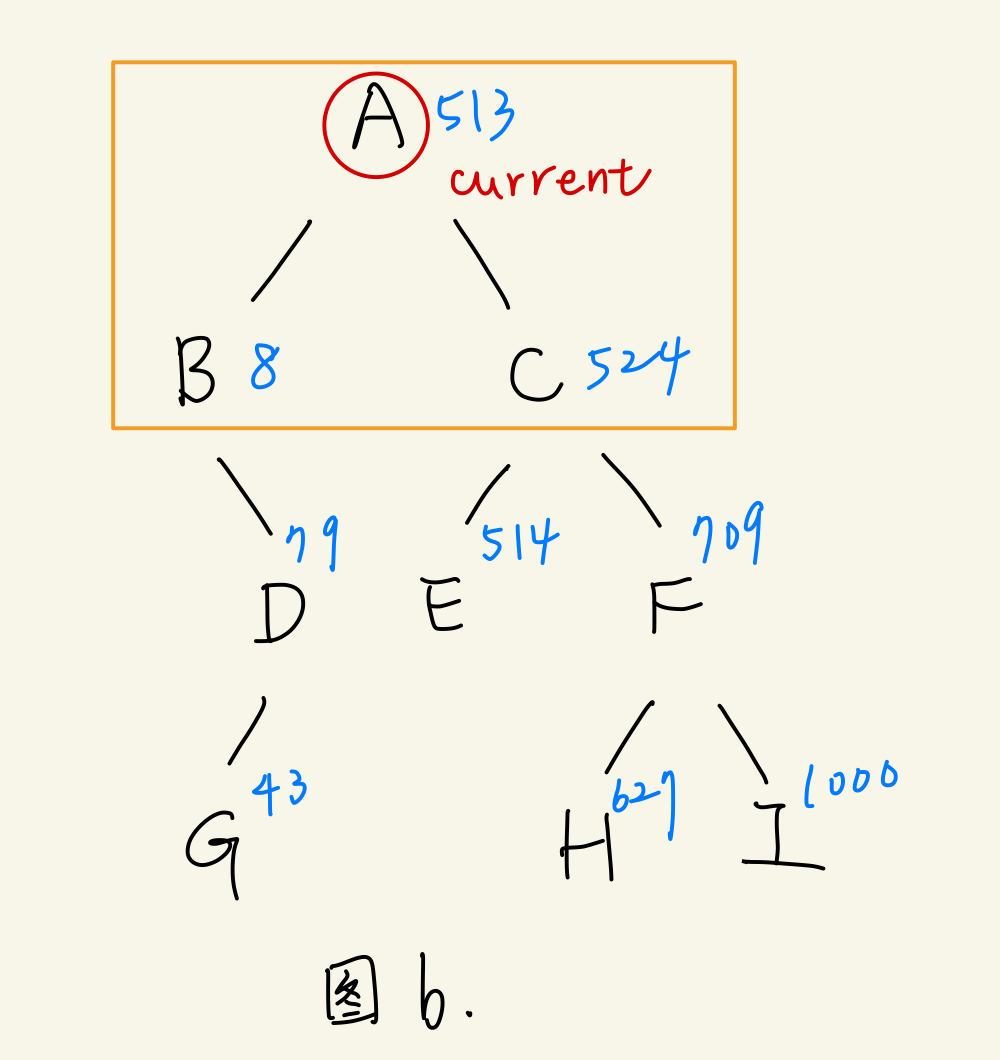
現有一棵BST如下圖a所示：



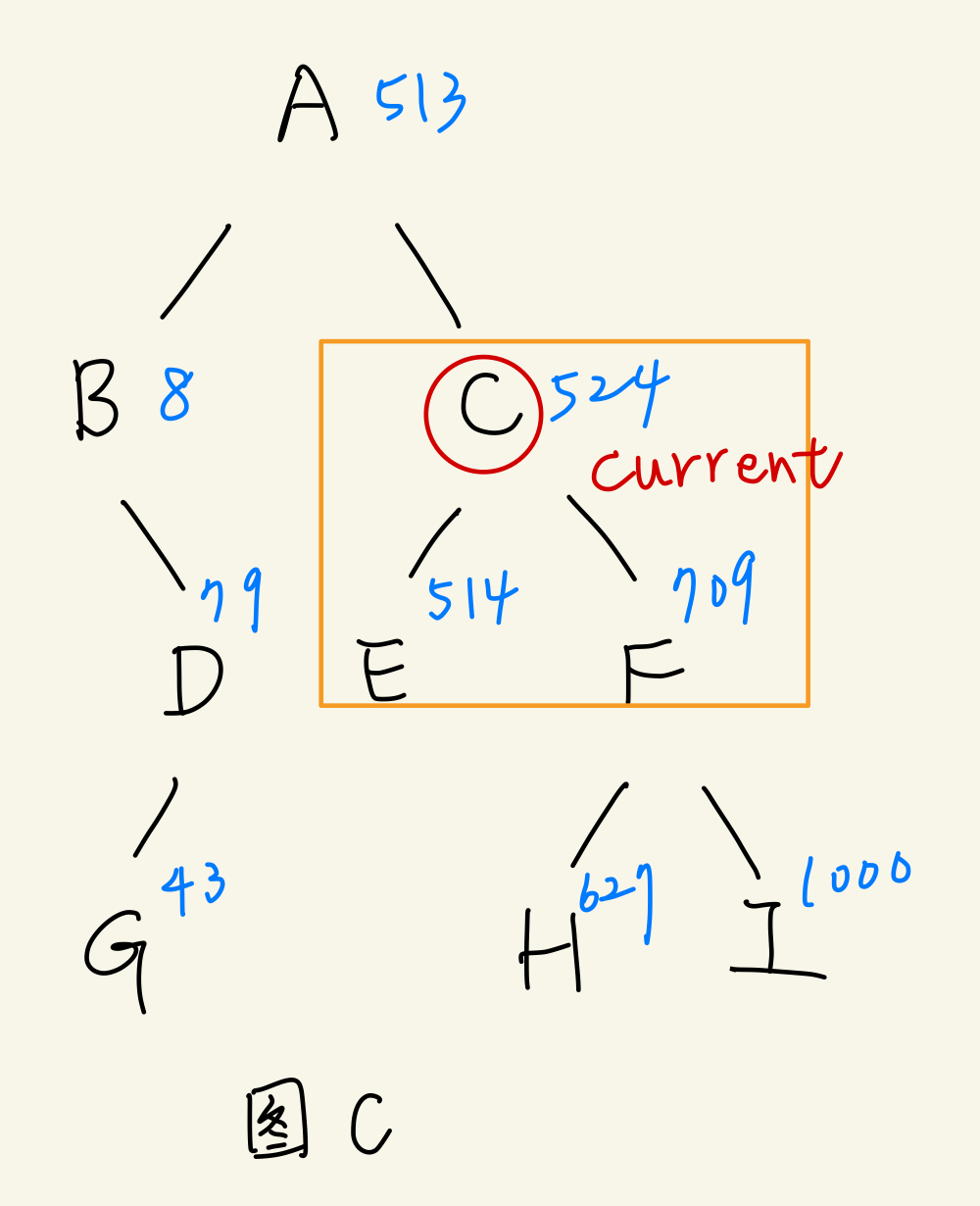
Case1：搜尋成功

若現在要從BST中搜尋node H，便以node H的data(627)進入BST。

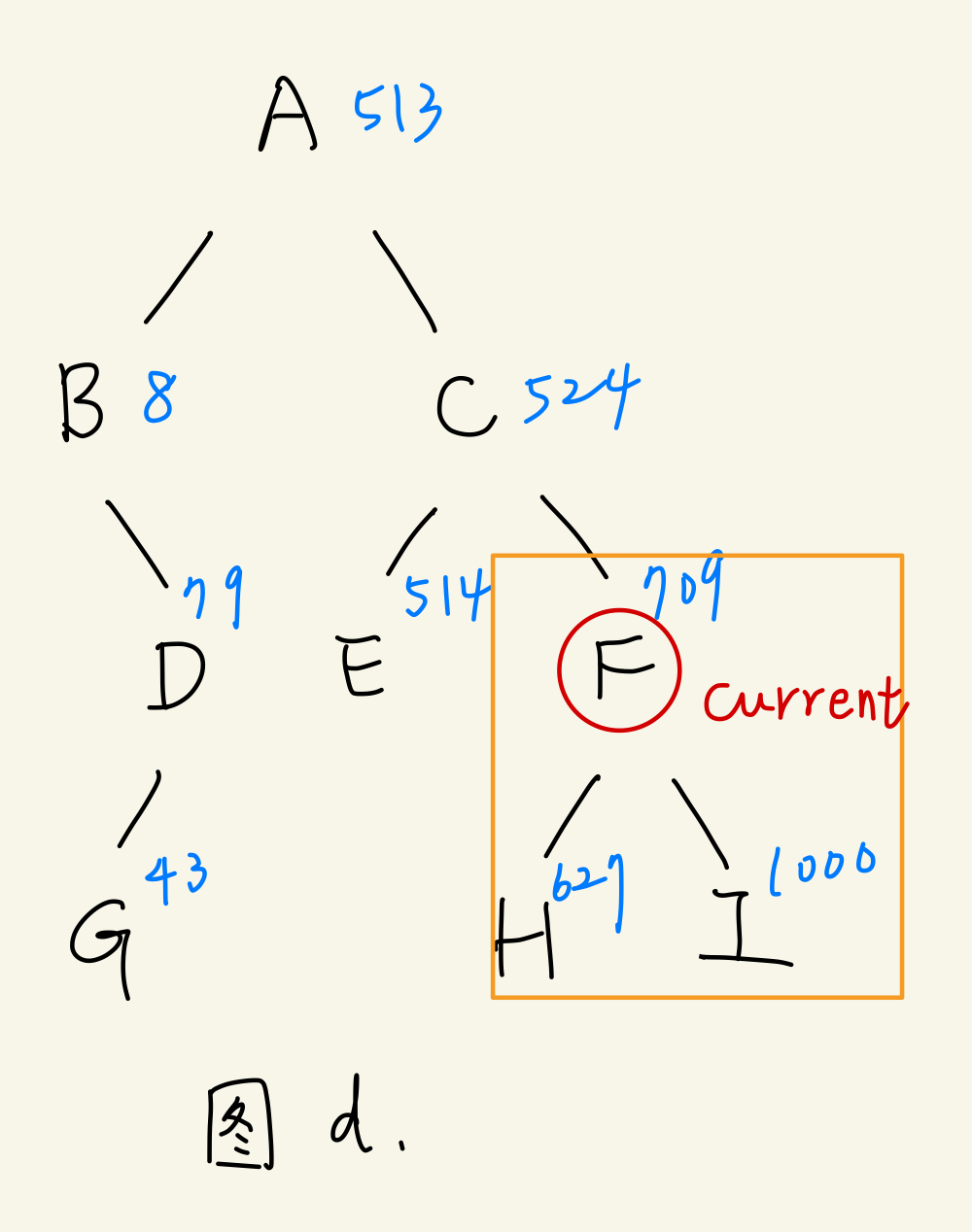
進入BST後，便把用來移動的Current指向root，如下圖b。



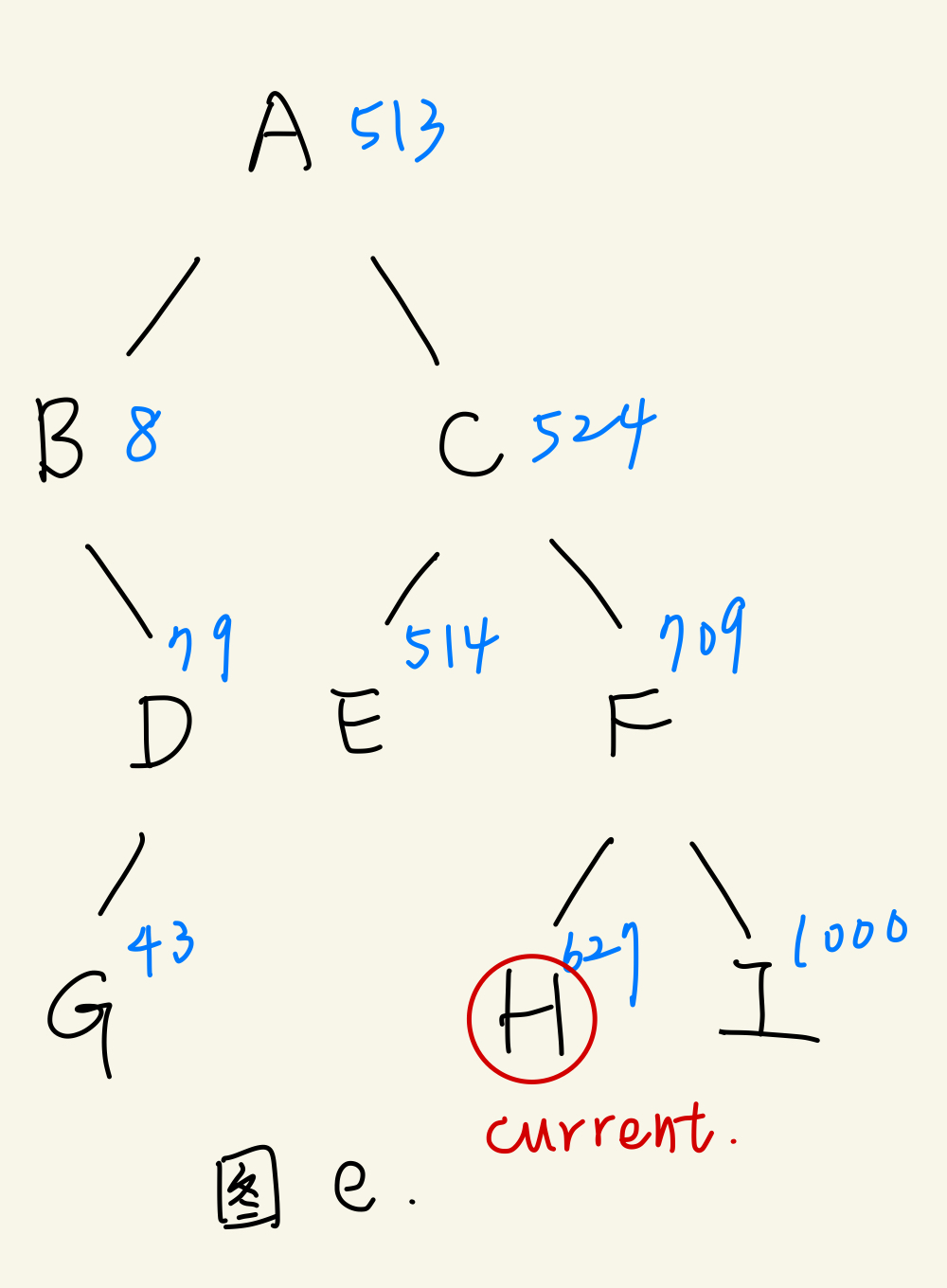
接著將data(627)和root的data(513)比較，如果node H存在，應該會長在root的right subtree裡面，於是便將Current往root的right child(node C)移動，如下圖c：



將Current移動到node C之後，再將data(627)與node C的data(524)比較，因此步驟同上，繼續將Current往node C的right child(node F)移動，如下圖d：



將Current移動到node F之後，再將data(627)與node F的data(709)比較，於是便往node F的left child尋找node H，如下圖e：



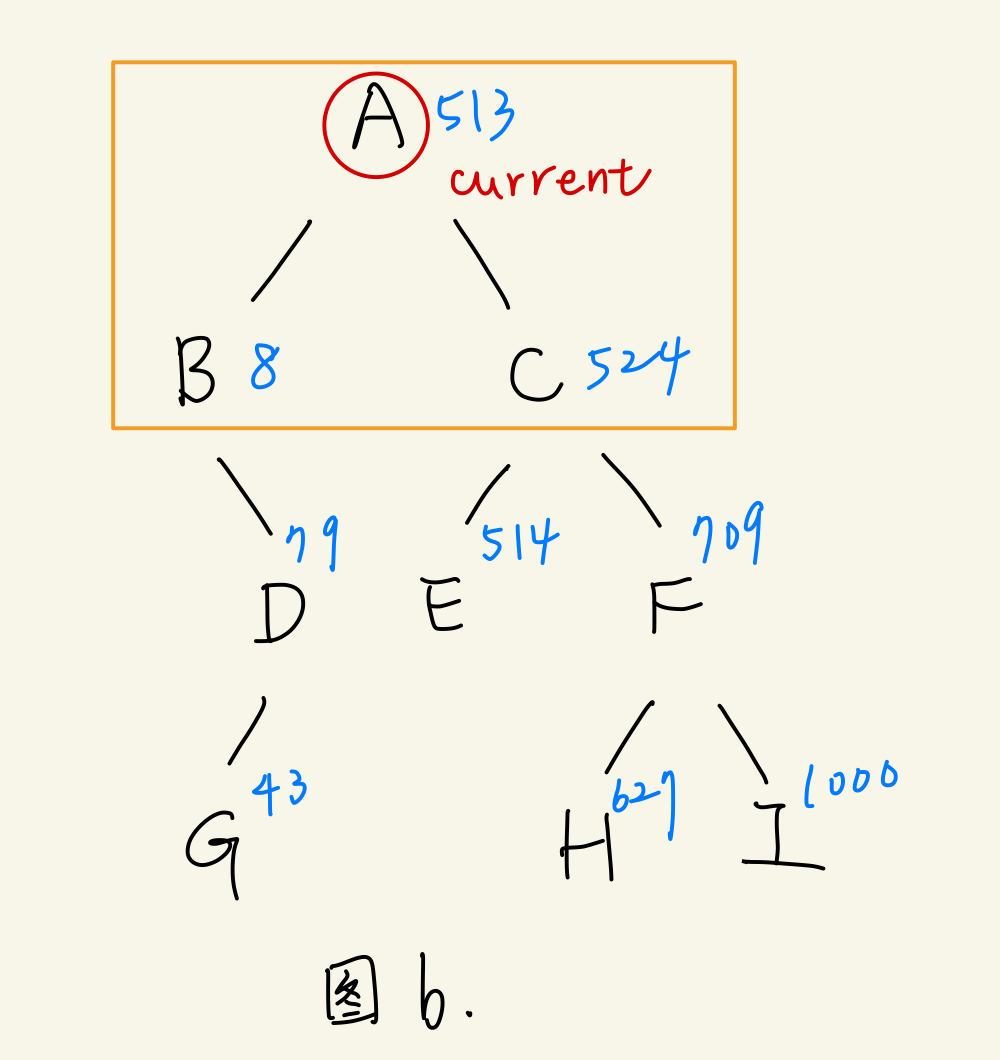
此時，Current的data(627)與傳送進findHelper()的data(627)相同，便確認Current即為node H，於是跳出while迴圈，並傳回Current。

宣告搜尋成功。

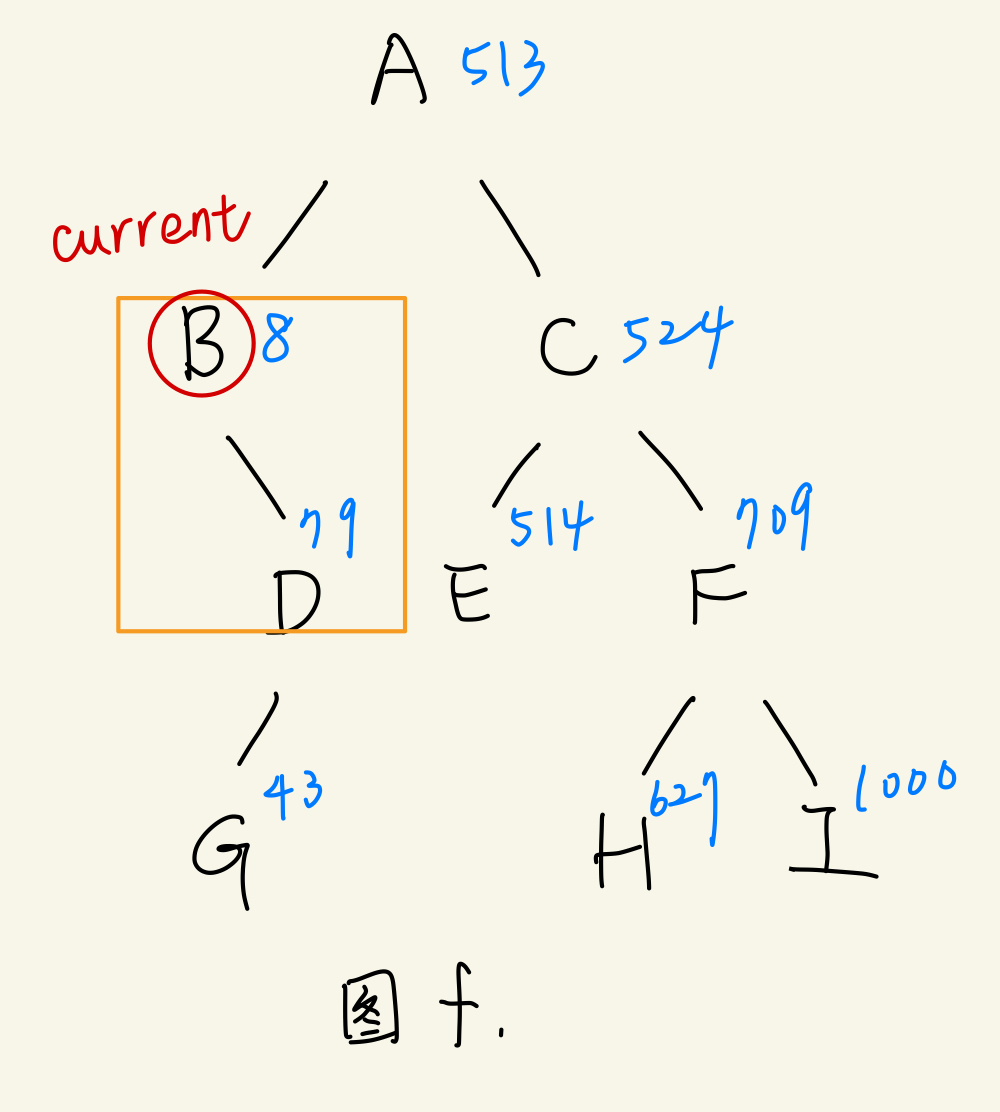
Case2：搜尋失敗

若現在要從BST中尋找node Y，便以node Y的data (2)作為data(2)，進入findHelper()。

進入BST後，同樣把用來移動的Current指向root，如下圖b。



接著便將data(2)和node A的root(513)比較，如果node Y(2)在這棵BST中，應該會長在root的left subtree上，於是將Currnet往root的left child(node B)移動，如圖f。



將Current移動至node B後，將data(2)和node B的data(8)比較，便判斷出，要將Current往node B的left child移動，如上圖f。

然而，由於node B沒有left child，於是Current指向NULL，便跳出迴圈，並回傳NULL，即表示搜尋失敗，node Y不在BST中。

在findHelper中，有兩種情況會跳出while迴圈：

Current移動到NULL或是dummy，表示搜尋失敗。

data與Current的data相同，表示搜尋成功；

**deleteHelper:**

Case1：欲刪除之deletenode沒有child pointer。由於deletenode沒有child pointer，因此只要考慮deletenode的parent，將其parent的leftchild指向NULL即可維持BST的正確性。

Case2：欲刪除之deletenode只有一個child pointer(不論是leftchild或rightchild)。由於deletenode有一個leftchild/rightchild，因此在刪除deletenode之前，需要先將leftchild/rightchild的parent指向deletenode的parent，並且將parent的rightchild從原本的deletenode指向leftchild/rightchild，因此，上述操作仍能維持BST的正確性。

Case3：欲刪除之deletenode有兩個child pointer。

由於deletenode有兩個child，若直接刪除deletenode的資料，並釋放其記憶體位置，要牽動的node較多。變通的方法就是「找替身」，原本要刪deletenode，但是實際上是釋放deletenode的Successor的記憶體位置，最後再把Successor的資料放回到deletenode的記憶體位置上，又因為BST的特徵，所有「具有兩個child」的node的Successor一定是leaf node或是只有一個child，如此，便回到如同Case2「至多只有一個child」的情境。

分成以下幾個步驟：

1先以BST::Search()確認想要刪除的node是否存在BST中；

2把真正會被釋放記憶體的pointer調整成「至多只有一個child」的node；

3把真正會被釋放記憶體的node的child指向新的parent；

4把真正會被釋放記憶體的node的parent指向新的child；

5若真正會被釋放記憶體是替身，再把替身的資料放回BST中。

即完成BST之刪除資料操作。

二、實驗：

1 新增資料

測試發現array 在資料量小時，所花時間反而比較大的原因可能是源自頻繁的擴大size (capacity\*2) 可以預期 pop 和 append 不會差很多 (除了array)

2 查詢資料

測試發現：array dlist 是 O(n) bst 是 log(n) 遠小於 n

3 traversal