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。
2. 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--。

2.3 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:

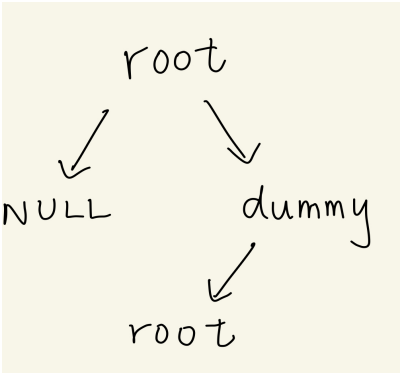
建立第一個 node 時，將此 node 設為 root，將 root->right 設為 dummy, 並將 dummy->left 設為 root。

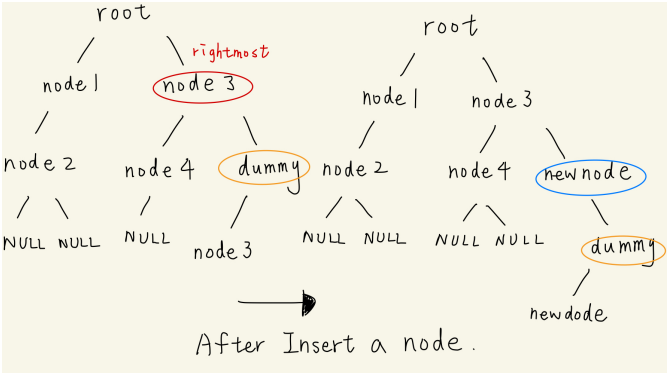
之後 insert 新的 node 時，若建立的位置為 dummy，會把 newnode->right 設為 dummy, dummy->left 設為 newnode。

<u>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。</u>

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：

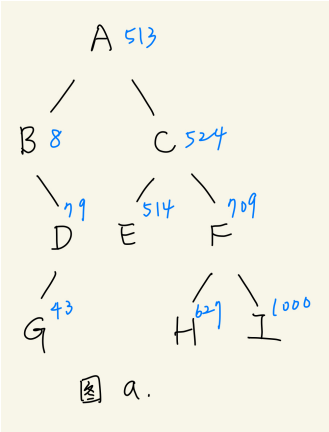


圖 a.

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。

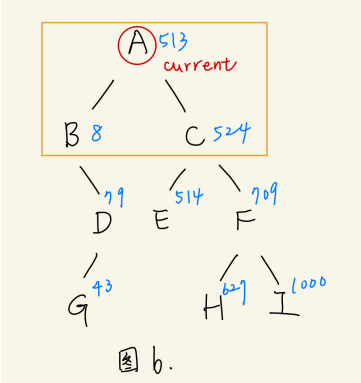
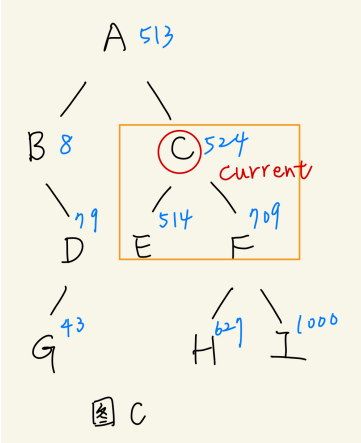
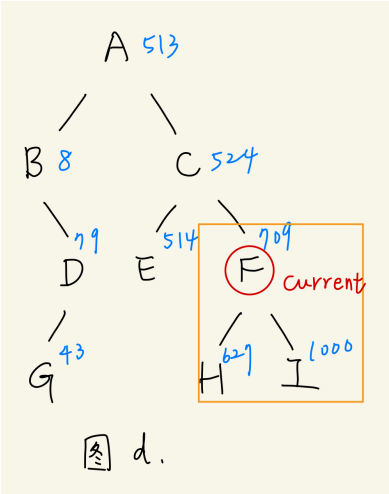


圖 b.

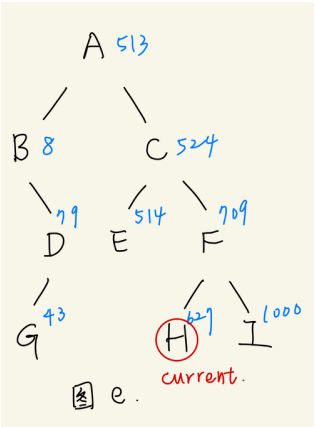
接著將 data(627)和 root 的 data(513)比較，如果 node H 存在，應該會長在 root 的 right subtree 裡面，於是便將 Current 往 root 的 right child(node C)移動，如下圖 c：



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：



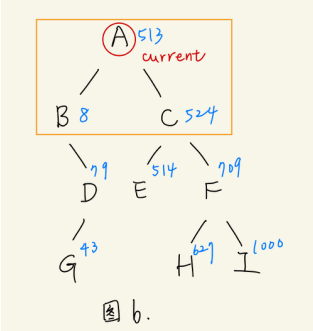
此時，Current 的 data(627)與傳送進 findHelper()的 data(627)相同，便確認 Current 即為 node H，於是跳出 while 迴圈，並傳回 Current。

宣告搜尋成功。

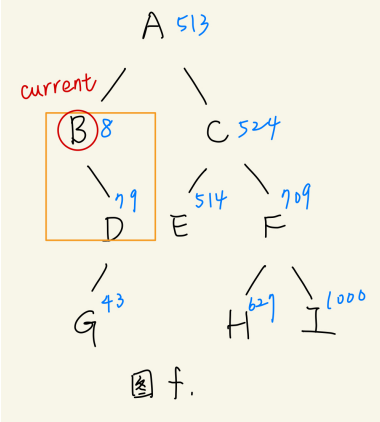
Case2：搜尋失敗

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；

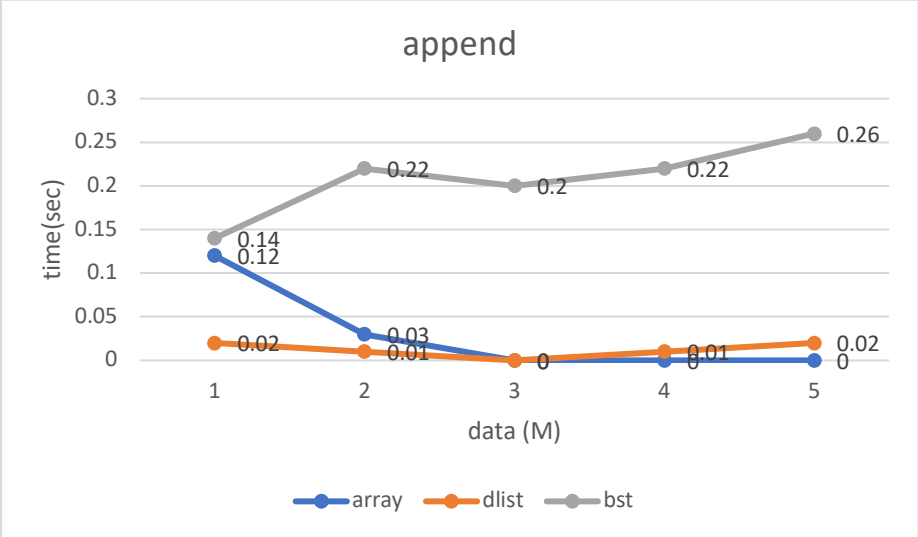
4 把真正會被釋放記憶體的 node 的 parent 指向新的 child；

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。

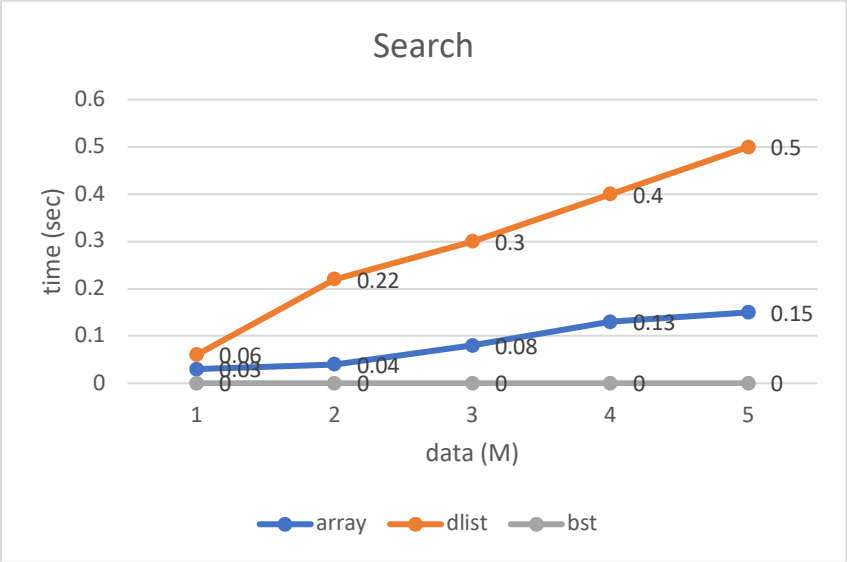
二、實驗：

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