資料結構實作 report

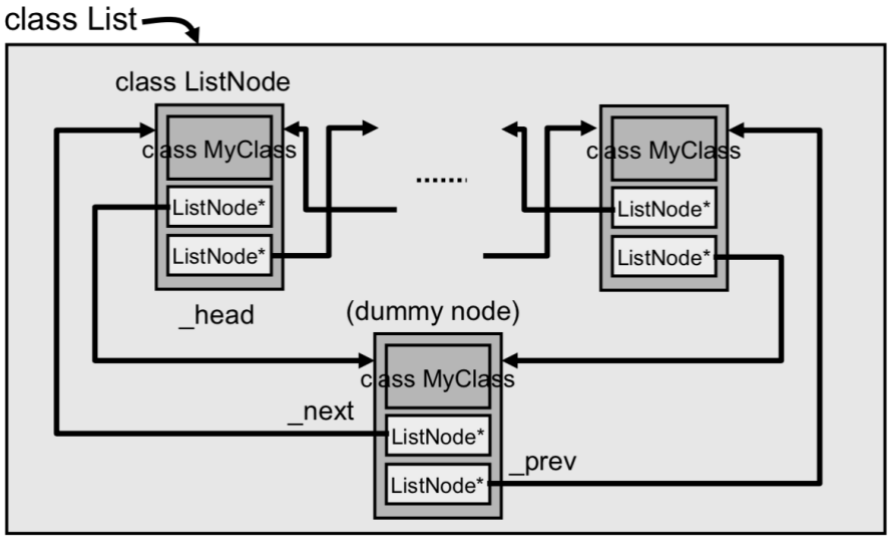
電機三舒泓諭 B05602052

Double Link list

將資料排列成環狀的一種 ADT。（概念圖如下圖）

在 DList 這個 class 裡面存著 dummy node(_head)，每一個 node 除了存自己所儲存的資料外，都存有兩個變數：_prev、_next，分別指著前一個 node 和下一個 node。藉此方法可將資料以雙向的連結方式連接在一起。

在一開始還沒有資料填入的時候，dummy node 的_prev、_next 都會指向自己，當第一個資料存入的時候，會在 dummy node 的後面接上第一個 node。換言之，在我的實作中 dummy node 不會被用來儲存資料，他存在的功能是為了讓整個物件首尾相連，會選擇這樣實作的原因還蠻簡單的，只是覺得這樣概念上蠻直觀的，此外，在刪除時也不用花很多心力在維持 dummy node。

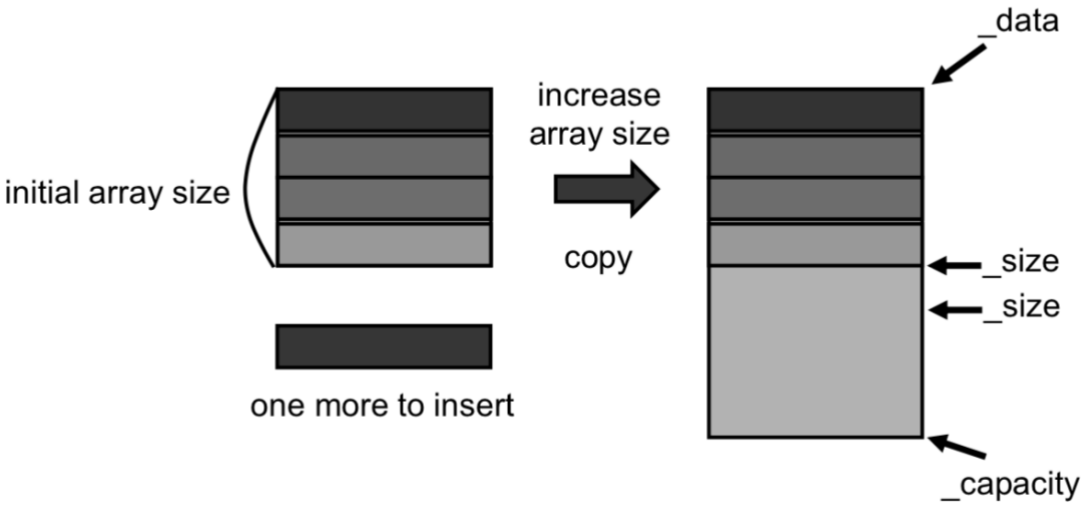


Dynamic Array

會隨著資料的數量改變 array 大小的一種 ADT（概念圖如下圖）

在 Array 這個 class 裡面會存者：_data(指著 array 第一個資料的 pointer)、_size(現在裡面具有的資料個數)、_capacity（array 最大可以容納資料個數的大小）。會那麼做的動機還蠻單純的，因為 **Dynamic Array** 好像這樣寫就是可以讓 memory overhead 最少的方法了。

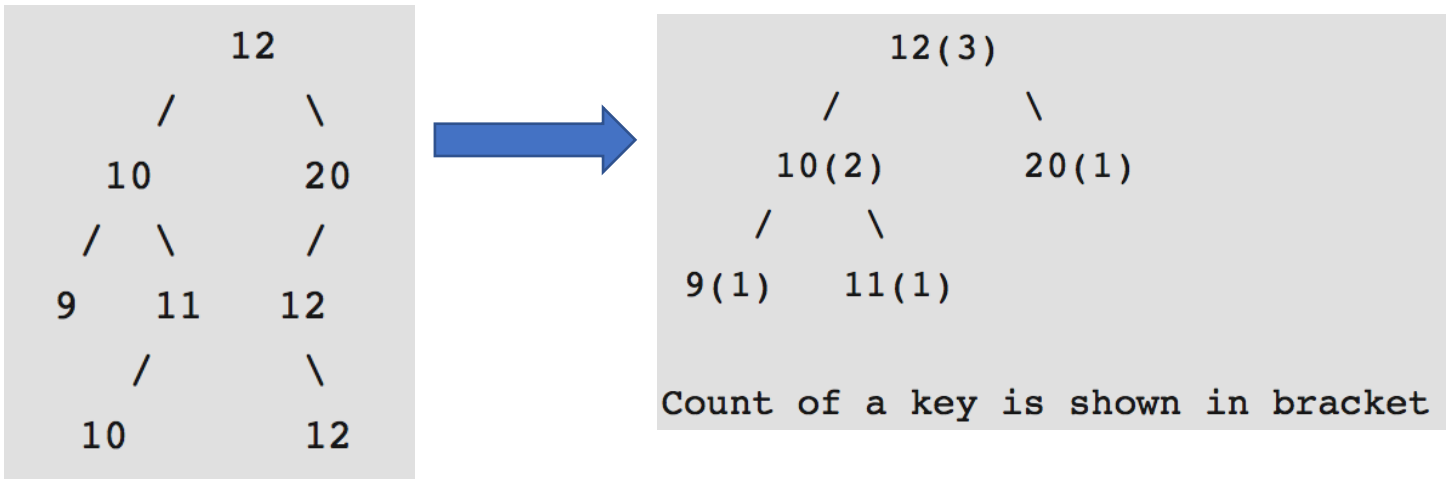
每當填入新的資料的時候會去確認_size 和_capacity 的大小，若_capacity 不夠的話，則會將原本的資料先暫存起來，再將_capacity 的大小翻倍，在和系統要一塊新的記憶體。此外，因為這個 ADT 不考慮順序的原因，所以不論是刪除一筆資料還是新增一筆資料，幾乎所以的運算都是 O(1)，除了 find()是 O(n)。



Binary Search Tree

將資料按照大小，排列在左右節點的一種 ADT。（概念圖如下圖）

左邊節點所存的資料會小於本身，且右邊節點所存的資料會大於本身，所以在建構此 ADT 時同時資料也按造順序建立好了。



一般的 BST 如果出現重複的資料，都會被放置於節點的右手邊，然而我最後實作出來的 BST 是右邊的那一種，相同的資料會被放置於相同的節點。會那麼做的動機是因爲覺得這樣寫比較直觀，而且 memory overhead 當同樣的 data 重複出現時，會明顯的下降很多（後面實驗會細講）。

在 BST 這個 class 裡面存著 dummy node，但和 DLIST 不太一樣的是，BST 並不像 DLIST 是個封閉的圖形。每一個 node 除了存自己所儲存的資料外，都存有 4 個變數：_left、_right、_parent、_count，前面三個 pointer 分別指著自己左邊的節點、右邊的節點、和自己的上一個節點、而_count 則是同一個節點資料的個數。存著最大數值的節點，會接到 dummy node，其他不是最大值的節點，則會接到 NULL，但要維持 dummy node 和節點相接花了我很多力氣，也引起了很多很難處理的 bug。

實驗比較

- 分析 **insert** 資料的時間複雜度

  實驗過程：

  adta -r n(資料數)

  實驗預期：

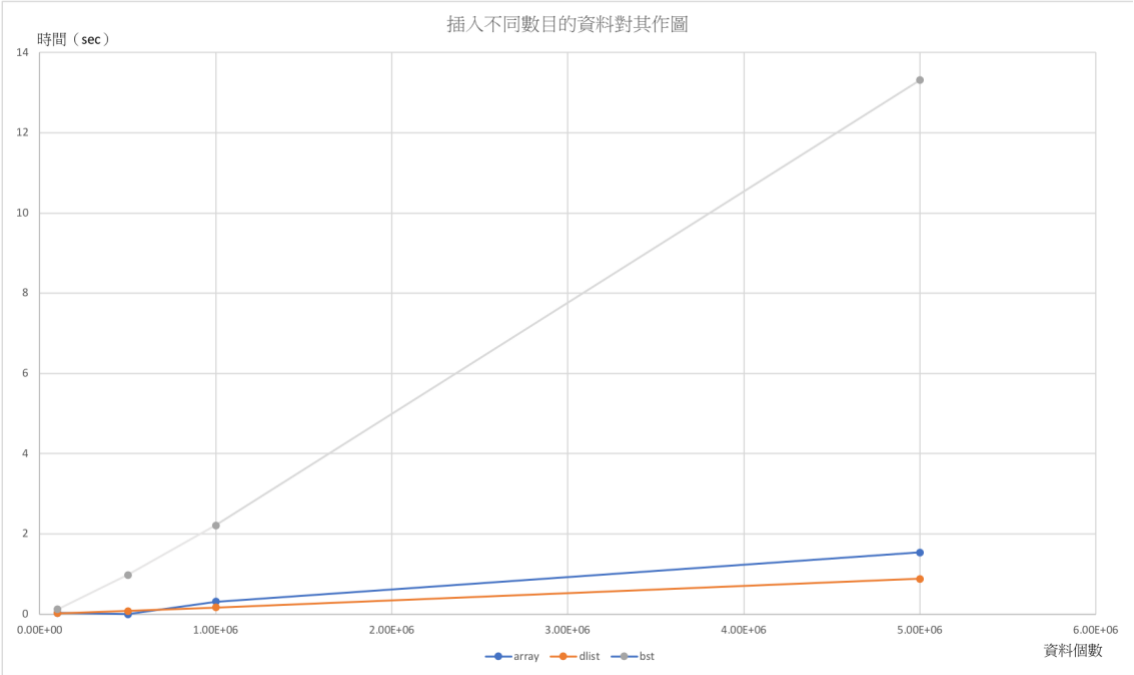
  理論上插入 **1** 筆資料，**array** 和 **dlist** 應該是 **O(1)** ，**bst** 應該是 **O(logn)**。插入 **n** 筆資料，所以 **array** 和 **dlist** 應該是 **O(n)** ，**bst** 應該是 **O(nlogn)**。所以預測起來時間應該是：**array=dlist<bst**

  實驗結果：

  時間： **dlist< array<bst**

  bst 的部分符合預期，但 **array** 和 **dlist** 和預期不太一樣，後來想想可能是因為每次 **array** 要重新更新 **size** 時，都會重新更新 **capacity** 的緣故。

	1.00E+05	5.00E+05	1.00E+06	5.00E+06
array	0.03	0.21	0.31	1.54
dlist	0.02	0.08	0.17	0.88
bst	0.12	0.98	2.21	13.31



- 分析 **delete** 資料的時間複雜度

  實驗過程：

每一個 ADT 都建好一百萬筆資料，接著 adtd -r n(資料數)

實驗預期：

理論上隨機刪除 1 筆資料，array 和 dlist 應該是 O(n) ，bst 應該是 O(logn)。刪除 n 筆資料，所以 array 和 dlist 應該是 O(n^2) ，bst 應該是 O(nlogn)。所以預測起來時間應該是：bst<array=dlist

實驗結果：

	1.00E+02	1.00E+03	1.00E+04
array	0.95	9.96	107.3
dlist	1.47	14.41	164.3
bst	16.74	154.4	1807

時間：array<dlist<bst

Array 和 dlist 的部分應該算符合，因為 dlist 刪除資料會去更新 pointer 所以花的時間會比 Array ，再多一些。但 bst 就蠻奇怪的，可能要再去看一下 code 的其他部分。