ADT 實作報告

一、 資料結構的實作

1. dynamic array

概述：就是動態版的傳統 array。

操作：與傳統 array 基本上一致。唯一的差別在於當記憶體不夠時，必須自己額外新增記憶體空間。

2. dlist

概述：可以將資料分開存放，但須耗費額外的記憶體來記錄前後的資料位置。

操作：這裡的 sort 我使用的是 insertion sort，理由是因為不太會出錯，而且實作只花我五分鐘的時間。而這裡的 sort 互換資料時，我並沒有移動節點的相對位置，而是直接調換節點間的內容。

另外，在 dlist 裡面我使用了兩個 private function。

Delete()：考慮到在刪除資料時，必須將要刪除的資料的前後資料對接。這個動作經常重複，所以直接以一個 function 來表示，增加可讀性及避免重複輸入相同的 code。

Connect()：在新增資料時，也必須將新增的資料及其錢嘔

資料的 pointer 更新，因此使用了一個 function 來代替多次重複的這個動作。

3. bst

概述：其實就是二維的 dlist，不過因為資料的排放有規則性，所以不需要 sort 就能保持資料的整齊性。

操作：這裡我使用 parent 來做++跟--。

++的判斷原則如下：1.先找有沒有右節點，如果有的話，就取右子樹的 min。2.如果沒有右節點，就找 parent，如果 parent 比較小，就找 parent 的 parent，以此類推。而到 end()的情況，我使用 tail 來接在 rightmost 的右邊。

--的判斷規則如下：1.先找有沒有左節點，如果有的話，就取左子樹的 max。2.如果沒有左節點，就找 parent，如果 parent 比較大，就找 parent 的 parent，以此類推。在這裡如果用 tail 當 end()會碰到一個問題，就是當 iterator 指向 tail 時，因為 tail 的_data 我們沒辦法決定，因此在跟 rightmost 比大小時，如果 tail 的_data 比 rightmost 小的話，rightmost 會被跳過。為了解決這個問題，我在每個 node 裡面加了一個 bool 的 member，判斷這個 node 是不是 tail，--的判斷會優先判斷指向是否為

tail，若為真，則直接回傳 parent。

另外，因為有用 tail 這個 dummy node 的關係，每個操作都要額外考慮 rightmost 的 case，去對 tail 進行維護，其中最容易忽略的是，tail 要接在新的 rightmost 上，而不是本來的 rightmost 的 parent 上。

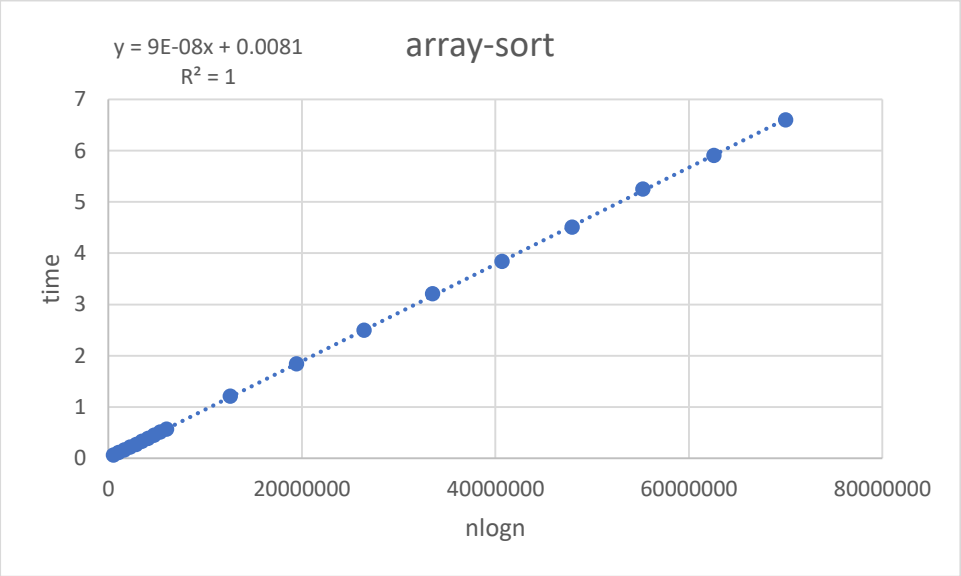
而之所以用 parent 跟 tail 來做，純粹是因為這樣理解最直觀。

二、 實驗比較

1. array 的 sort

推測：O(nlogn)

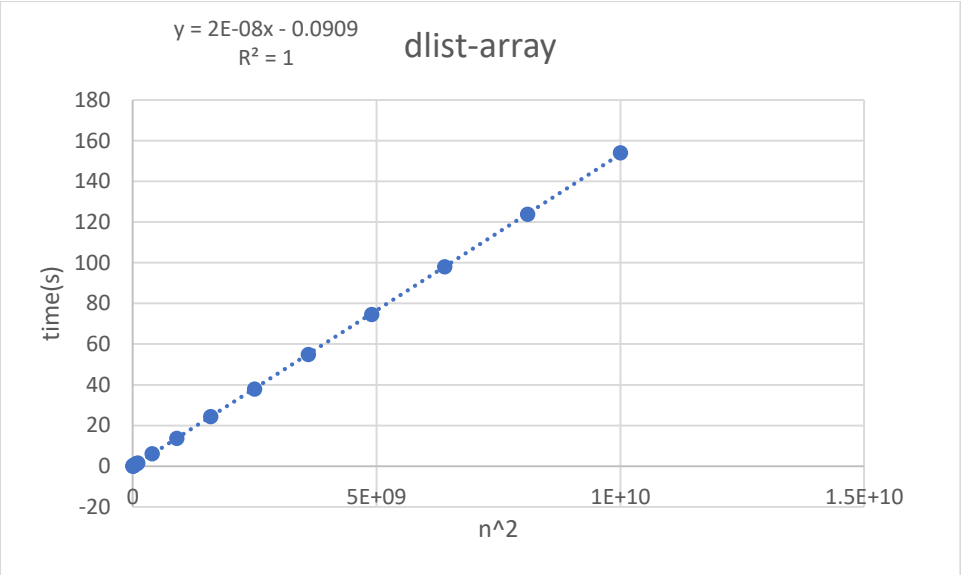
數據：



2. dlist 的 sort(insertion sort)

推測：$O(n^2)$

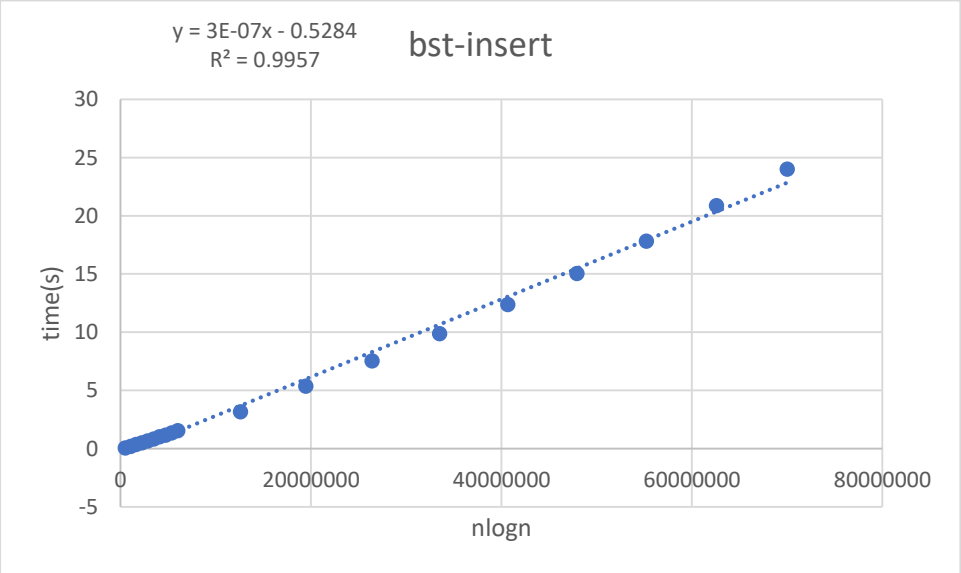
數據：



3. bst 的 insert

推測：O(nlogn)(本來是 empty 的情況下)

數據：

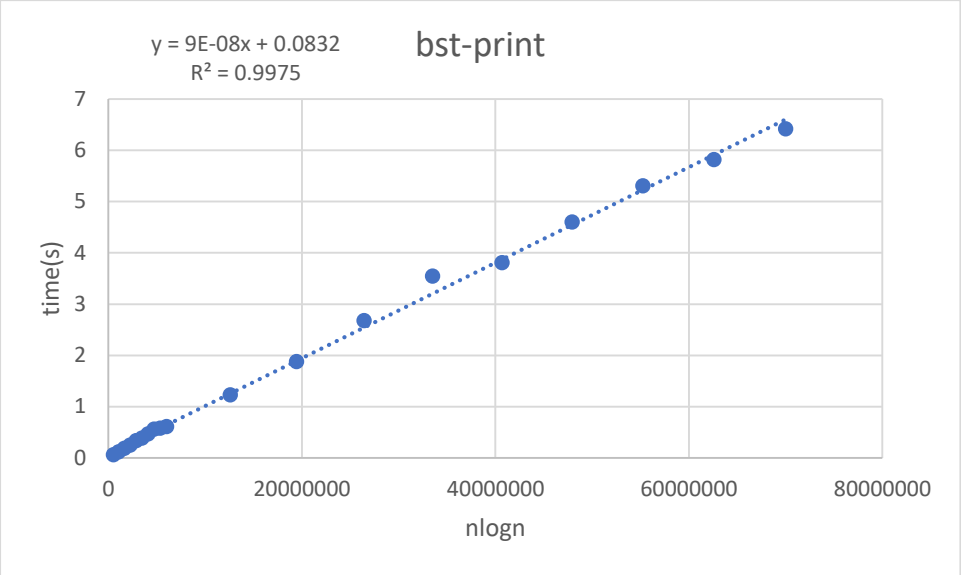


分析：發現 n 越大，會比 nlogn 稍大，原因應該是因為 n 越大，bst 就越有可能 unbalanced。

4. bst 的 print

推測：0(nlogn)，因++應為 0(logn)

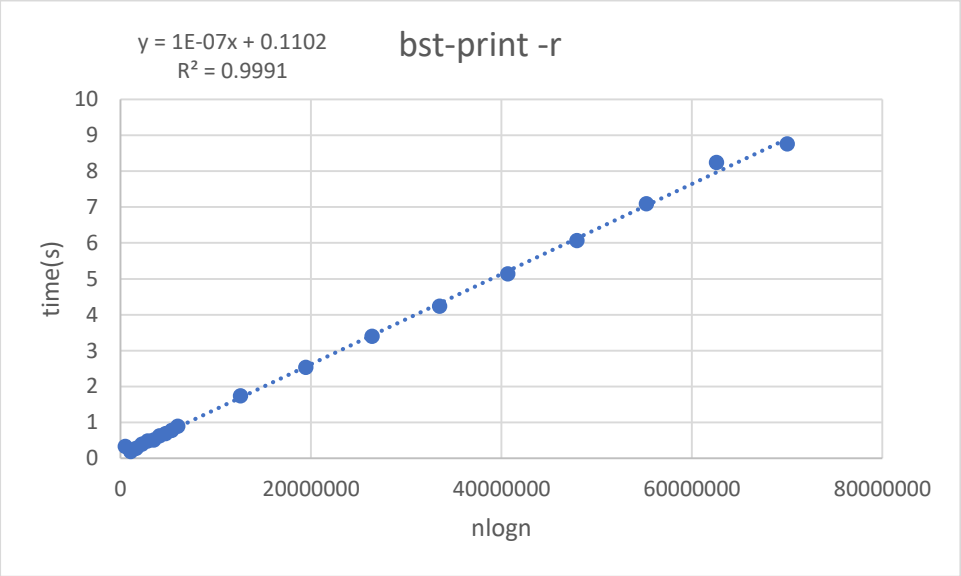
數據：



5. bst 的 print -r

推測：0(nlogn)，因--應為 0(logn)

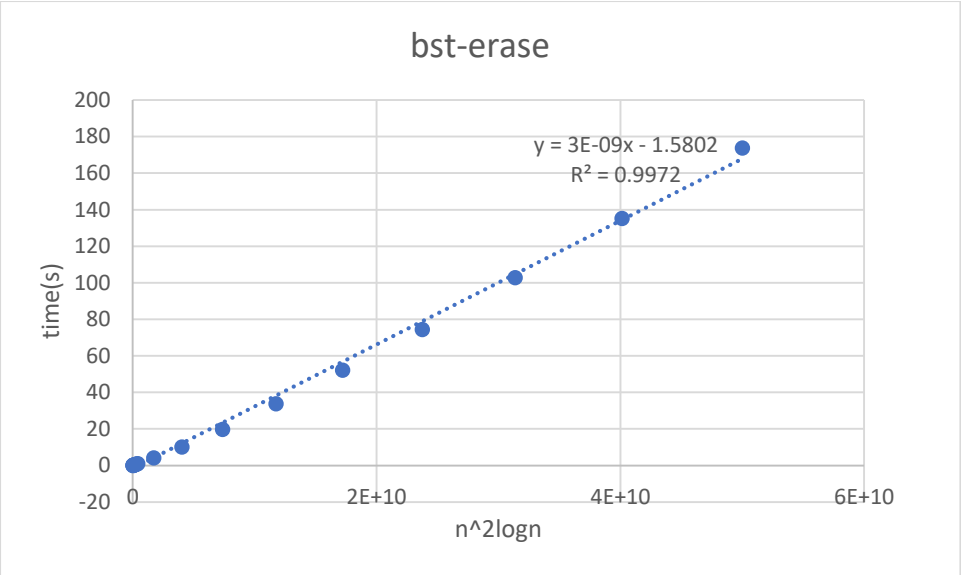
數據：



6. bst 的 erase

推測：0(nlogn)(在 n 個資料中隨機刪除 n 次)

數據：



分析：結果實際測試，竟然是 $O(n^2 \log n)$…，而且還會因為 unbalanced 而越來越像 $O(n^3)$。

結論：就線性資料結構來講，不論是 coding 的容易度，或是 performance，dynamic array 都比 dlist 好上許多，所以以後有機會的話，我會傾向使用 dynamic array。

而 bst 在 insert 的時候就已經放好位置，所以不需要 sort。不過跟 array 相比起來，bst 的 insert 是 O(nlogn)(best case)，print 也是 O(nlogn)(best case)，不如 array 的 push_back 是 O(1)，sort 是 O(nlogn)。