ADT Performance Report

B03901030 電機三　蕭晨豪

一、資料結構的實作

- Dlist:

此 ADT 紀錄最前面的 node 的指標(_head)，並且每個 node 會記錄其前一個 node 與下一個 node 的指標，以此連接方式建立 node 的關係，在此 ADT 的最後加上一個 dummy node 並再將其連回_head 形成一個環狀，此 ADT 的每個 node 有各自分開的記憶體位址，並不是連續的。

在操作時若此 ADT 為空，先將_head 設為 dummy node，之後每當增加新的資料，以 node 包住資料後將其和欲插入位置的前後 node 的指標重設，使之串聯，即可將該 node 插入欲插入之位置。

本程式使用 insertion sort 來做 sorting，考慮到方便性以及許多情況下效能較 bubble sort 提高一些，但平均的 time complexity 還是 n^2。

- Array:

此 ADT 使用動態陣列方式儲存資料，有別於 Dlist，Array 將所有的 node 存在一大塊連續性的記憶體中，在此我們使用一項 data member *_data 來做為 dynamic array 儲存資料。

Array 有兩個重要的變數：_size 用以儲存目前已經儲存的 node 數，而 _capacity 則是目前的 Array 可用記憶體上限，透過比較兩變數便可判斷是否還有剩下的記憶體可供使用。

增加 node 時首先判斷記憶體是否足夠，若足夠，配置 Array 的一個區塊來儲存新的資料。若 node 數已滿，到達 capacity 上限，則重新和電腦要求原先兩倍的記憶體空間，並將原本的所有資料複製上去，然後刪除原本的 dynamic array(也就是 _data)，最後再新增 node 來儲存新的資料。Sort 時直接使用 std::sort，其 time complexity 為 nlog n。

- Bst:

此 ADT 的設計包含:

1.class BSTreeNode，其中有_parent, _left, _right 三個指標來指向其父和左右的 node，_data 即為該 node 的資料儲存處。

2.class BSTree，其中有兩個 BSTreeNode 的指標_root 和_end，_root 為此二元搜尋樹的根，在 empty 時此指標指向 NULL，而_end 即類似 Dlist 的 dummy node，放在最大元素的下一個，作為 iterator 的結尾。

原本的設計中我沒有設計_end，意即此樹的所有 leaves 其左右節點指標都為 NULL，但後來在 reverse print 時我發現 iterator 並無法知道最大元素的

位置，思考之後決定不在 iterator 額外加變數儲存最後的位置，而是在 BSTree 中增加了_end。

由 BST 的特性我們可以快速的找到特定資料，設計中我選擇使用_parent 而非在 iterator 中使用 trace 是因為有_parent 較方便在++和- -時進行回溯，用 trace 的 time complexity 因為必須從_root 往下找為 logn，在資料較大時，我認為使用_parent 會有較好的效能。

二、實驗比較

1. 實驗設計

以 adta -r 100000來測試隨機新增資料的所需時間。
以 adts(對50000筆隨機產生的資料)來測試 sorting 速度，由於 BST 已為有序的資料結構，因此以新增資料的時間來做為 sorting 時間。
以 adtd -r 10000來測試隨機刪除資料的所需時間。

以上均使用內建之 USAGE 指令來查看記憶體用量與時間。

2. 實驗預期與結果

- adta -r 100000

預期時間：Dlist 由於直接插入所以是 O(n)、Array 也是直接填入資料，但在需要要求兩倍記憶體時(每個二的冪次的臨界點)由於要將舊資料複製過去，因此為 O(nlogn)、而 BST 由於插入資料相當於二分搜尋，為 O(nlogn)

實驗結果：Dlist 0.02s, Array 0.06s, BST 0.12s

和預期結果相符，由於 Array 只有在臨界點時需要複製舊資料至新的_data，在資料量越來越大時兩臨界點相距較遠，因此介於兩臨界點中的資料量其 add 速度會較 BST 快一些。

- adts(對50000筆資料做排序)

預期時間：Dlist 使用 insertion sort 為 $O(n^2)$、Array 使用 std::sort 為 O(nlogn)、而 BST 的插入資料為 O(nlogn)

原本使用第一步產生的100000筆資料做排序，但 Dlist 的排序會直接死亡(時間太久)，因此最後選用50000筆來看趨勢。

實驗結果：Dlist 86.62s, Array 0.07s BST 0.05s

在此資料數量下與預期結果相去不遠，但 Dlist 在資料量較大時所需的排序時間會增長很多，猜測是因為我在 sort 裡面使用的是資料交換而非指標，因此當資料量較大時所需的複製時間會增長許多，造成遠遠超過 $O(n^2)$的時間。

- adtd -r 10000(使用第一步產生的100000筆資料來做刪除)

預期時間：Dlist 為 O(kn)（需先看長度、走到該位置再刪除)，Array 為為 O(k)(因為刪除只需將最後的元素移到該刪除位置)、而 BST 為 O(knlogn)因搜尋過程需 logn

實驗結果：Dlist 5.18s, Array 0.07s BST 83.87s 符合預期。

3.結論

就以上測試來看，Dlist 在 sorting 方面不盡理想，而 BST 在 delete 方面的時間也所需過久，Array 是三者中最好的選擇，但就對這些資料結構的認識我認為：Array 相當於使用「空間換取時間」，需要較多的記憶體來維持好的效率，並且當資料恰巧超過臨界點時，若資料量已經很大我們再要求兩倍的空間，空間的後半部卻幾乎沒有用到，相當浪費。

另外，這次的測試只是對於單一的指令，若使用者對資料結構有特殊的需求(ex:僅要求增減物件而較少使用排序指令，Dlist 較佳；或要求每一步增減時需同時排序，BST 較好)則不同的資料結構還是能更好的滿足使用者的目的，因此這些都只是 trade off，實際的選擇還是要依使用者的需求而定。