DSnP hw5: ADT Implementation & Comparison

學號：B03901108 姓名：黃暐倫

The Introduction to ADTs:

dynamic array (以下簡稱 array)以一段可以持續倍增容量的連續記憶體區塊作為基礎；doubly linked-list (以下簡稱 dlist)以動態增減的分散節點們組成，節點本身具有連接左、右節點的橋樑，藉此組成一條完整的 linked-list；binary search tree (以下簡稱 bst)以動態增減的分散節點們組成，然而 bst 的節點具有三座橋梁，分別連接 key value 小於自身 key value 的節點(left child)、key value 大於等於自身 key value 的節點(right child)(stable insertion if in ascending order)以及 parent。

在新增資料上：array 倍增容量的措施，使 push_back()的時間複雜度在 amortized analysis 之下為常數；dlist 可直接斷開與銜接節點間的連結，故 push_back()時間複雜度亦為常數；bst 為維護排序，插入過程包含一串的比較，時間複雜度為 O(h), h: the height of the bst。在刪除資料上：由於 array 的記憶體區塊連續，故時間複雜度正比於目標與最末端的距離；dlist 則可以達成 const. time deletion，同是因為自由的節點連結；bst 為維持排序，需要尋找 right descendants 中的 leftmost 或是 left descendants 中的 rightmost 來替換，故時間複雜度為 O(h)。在排序上：array 可直接接受 Q-sort 或是 merge sort，時間複雜度 Θ(nlogn)；dlist 可藉由創建暫時的 array，以 Q-sort 或 merge sort 達成時間複雜度 Θ(nlogn)；bst 隨時維持排序，故時間複雜度為常數。在搜尋上：sorted array 可直接做 binary search (時間複雜度 O(logn))，但廣義上只能使用 linear search，時間複雜度 O(n)；dlist 因為節點分散，故只能使用 linear search；bst 本身即為 binary search 所建構，時間複雜度 O(h)。

The Implementation to ADTs: How and Why

由於 array 與 dlist 的實踐上，我們發揮部分甚少，故先探討 bst.：我的 bst.並無 dummy node，每一節點具有連接 left child , right child, and parent 的三只指標，bst 本身除了連接 root 的指標_entry (無任何資料時，為 NULL)以外，尚有_size 記錄節點數量與 bool _balanced 緩和不平衡的發展(效果不明，算是一種經驗法則？)。因為缺乏 dummy node，故 the parent of root == NULL (新增判斷 root 的選項，且事後來看有助於程式碼的統整，因為減少函式內_entry 的氾濫)。

Iterator increment & decrement 大致無異於 bst traversal，然而因缺乏 dummy node 卻需要可逆的 end()，故利用指向大小足夠之物件的指標必為偶數這項特性，increment the address of the rightmost node，設之為 end()；進行 decrement 時，先行檢查指標奇偶，奇者 decrement 後即還原成 the rightmost node。

erase()先檢查目標是否同時具有 left child 與 right child，若是如此則根據 _balanced 的真假，分別挑選 the rightmost left descendant 與 the leftmost right

descendant 調換目標，使目標退化為 single/no child。_balanced 的設計單純倚靠直覺：設定上右邊節點是大於等於、左邊是小於，所以未有 erase()的狀態下，結構趨向右大左小；根據上述，先設定_balanced 為假，取右邊節點調換目標，下次 erase()又進入 double children 狀況時，再取左邊節點，於兩態之間振盪。

之所以捨棄 dummy node 卻保留_parent，是因個人認為 dummy node 的維護麻煩程度高於其帶來的便利性，以一條指令的完整執行來看，_entry 的效能可與之相比；但缺乏_parent 卻會讓 iterator tracing 難度提升、iterator 牽扯的空間增加(可能需要多出 stack)，而且在 class BSTree 裡的運作將因此大量仰仗 iterator，而非直接對指標進行操作，這與個人的 coding style 不符。

_balanced 的使用看似可協助平衡，但 erase()的狀況不一、pop_front()與 pop_back()對於平衡的侵害、insert()的資料序列等因素皆非單一個_balanced 可以解決，只能希望測試環境讓此資結貼近 randomly built bst。本試圖建構紅黑樹，後來因為結構複雜，且測試的隨機性質縮短 BST 與 BBST 之間的時間複雜度，BBST 實務上的常數衝擊提升，所以作罷。

原本考慮以遞迴方式建構 erase()，但因節點的替換是單向進行，故 function stack 的便利性無法抵銷其過度的空間使用，個人最後採用迭代方式完成。然而因為 bst 的清空是由上往下搜尋、再由下往上清掃，故個人使用 function stack 完成 clear()的功能。(-Verbose 基於相似理由使用遞迴方式建構)

dlist 則有 dummy node 作為入口、isSorted 記錄排序與否。整體而言，dlist 的設計並無奇特之處，畢竟 dlist 是一套極容易建構的資料結構。但由於 isSorted 的存在，資料的搜索得以從 $\Theta(n)$ 變為 $O(n)$；而在新增資料時，藉由設計以避免 isSorted 不必要的破壞。而在排序部分，個人先以陣列裝入指向各節點的指標，再針對該陣列進行 comparison sorting ($\Theta(n\log n)$ time complexity)，最後完成節點的指標更新，故排序時間複雜度 $\Theta(n\log n)$ $(\Theta(n) + \Theta(n\log n) + \Theta(n))$，使用的比較函式為 static bool cmpr()。

由於 dlist 容易建構，所以程式的複雜性甚小，個人在建構 dlist 時，專注於時間複雜度的精進，盡量利用現有的空間(如：isSorted)，來避免時間的浪費(如：防止排序已排序資料、藉由排序特性減少搜索時間、盡力維持 isSorted 的真值)。

Array 的建構中，我並沒有使用 isSorted：因為此次作業 erase()與 push_front()的設計，排序特性極易遭受刪除資料的破壞。不過這樣的設計業讓 array 得以巧妙繞過記憶體區段連續造成的不便，達到時間複雜度為常數的刪除，代價是需要時時防止 self-assignment 的發生。個人也有建構 reserve()以及 resize()，參照 C++ vector 相關函式的運作。_capcity 的存在與資料刪除的設計允許一連串不包含排序與搜索的操作時間複雜度正比於操作數。由於架構本是如此，個人可發揮處不多，故不多著墨實踐原因。若是 isSorted 較不容易遭受破壞，則可新增 binary search 的功能於 array 之上，盡量利用排序特性。

The Experiments with ADTs: Design, Expectations and Results

實驗方式為每次只跑達成待測操作的最少指令，並隨即 q –f，觀察 user time 作為實驗結果：待測操作有建構(資料隨機)、排序、隨機刪除。

Array 各操作的時間複雜度，理論上如下：建構 $\Theta(n)$、排序 $\Theta(nlogn)$、隨機刪除 $\Theta(1)$，n: the size of the array (to be constructed)；Dlist 各操作的時間複雜度，理應如下：建構 $\Theta(n)$、排序 $\Theta(nlogn)$、隨機刪除 $O(n)$ (search: $O(n)$, deletion: $\Theta(1)$), n: the size of the dlist (to be constructed)；BST 各操作的時間複雜度，理論上如下：建構 $\Omega(nlogn)$ $but\ O(n^2)$、排序 $O(1)$、隨機刪除 $O(h)$, h: the height of the tree, (search: $O(h)$, deletion: $O(h)$)，n: the size of the bst (to be constructed)，若是環境具有隨機性質，則從機率觀點來看，BST 可視為平衡，以下操作的時間複雜度：建構 $\Theta(nlogn)$、隨機刪除 $O(logn)$。

建構：

n	array	dlist	bst
500000	0.56	0.25	1.48
1000000	0.98	0.47	3.38
2000000	1.98	0.95	8.24
5000000	6.93	2.34	23.75
10000000	15.24	4.7	52.78

排序：

n	array	dlist	bst
500000	0.62	1.34	0
1000000	1.21	3.25	0
2000000	2.55	8.17	0
5000000	6.78	24.8	0
10000000	14.24	54.06	0

隨機刪除：

n	array	dlist	bst
500000	0.24 / 500000	0.38 / 5	0.25 / 5
1000000	0.44 / 1000000	0.65 / 5	0.42 / 5
2000000	0.75 / 2000000	1.64 / 5	1.14 / 5
5000000	1.8 / 5000000	4.28 / 5	2.54 / 5
10000000	3.47/ 10000000	9.55 / 5	5.36 / 5