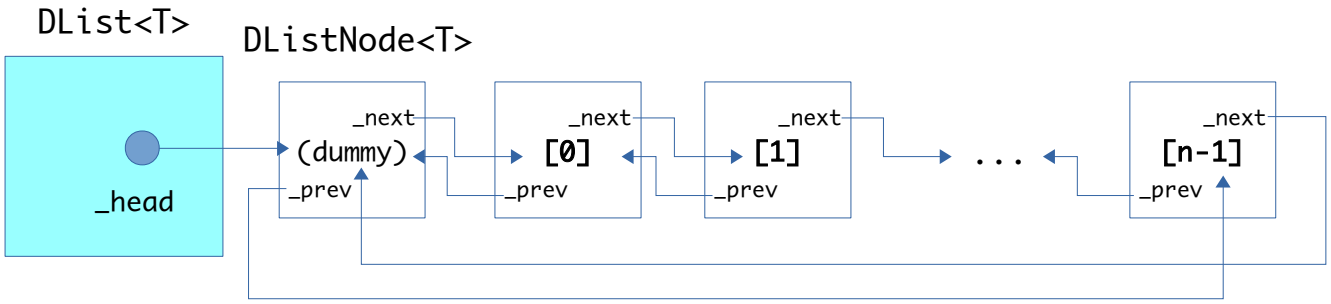
ADT Performance Study Report

化學二 B03203004 潘廣霖

資料結構的實作

Double-Linked List

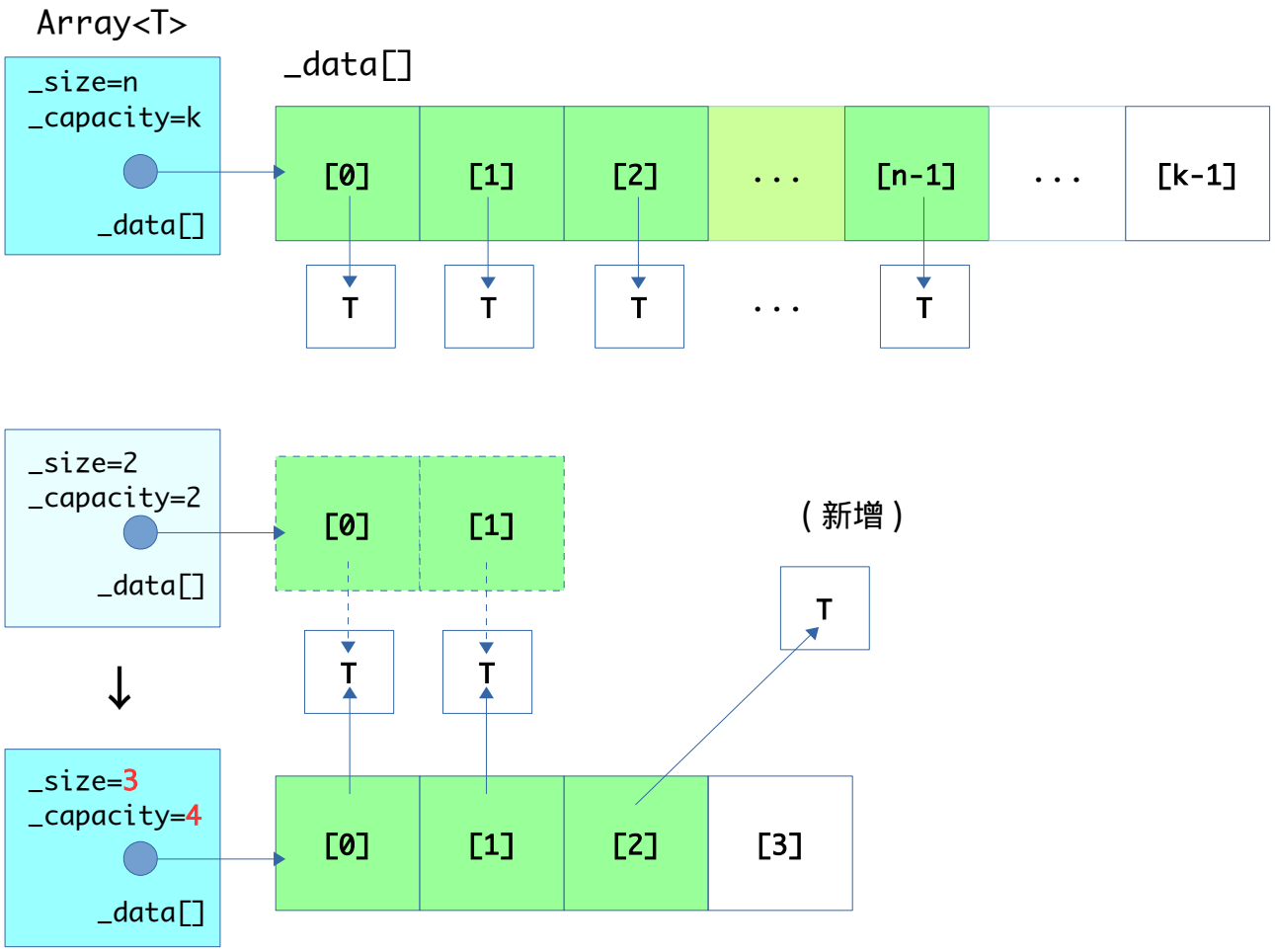
建立一連串物件，ADT 紀錄最前面物件的指標(_head)，每個物件紀錄兩個指標，分別指向前一個元素(_prev)與後一個元素(_next)。_head 是個 dummy node，它的下一項才是整個 list 的第一項，為了維護方便，將整個 linked list 設計成環狀的，最後一項的下一項指向 _head，_head 的前一項指向最後一項，如此一來頭尾相連，如下圖。便於做雙向走訪、在任意兩項間插入或刪除(直接改動指標即可)，但只能做循序走訪，無法隨機走訪，排序時使用氣泡排序(bubble sort)。



Dynamic Array

ADT 於執行時期動態宣告一段連續的記憶體，存放指向物件的指標，紀錄其開頭指標(_data)、資料長度(_size)、容量(表示這個 array 最多可以存放的物件數，也就是動態宣告時的長度_capacity，起始為 1)。

新增物件時，直接將物件開在 array 的最後面，若 _size 即將超過其 _capacity，則重新建立一個容量為本來兩倍的 array，將原內容複製進去，再將本來的 array 釋放掉，換成這個，如下圖所示。刪除物件時，將後面所有元素向前挪一格，因此刪除所花時間跟物件與陣列尾部的距離成正比。排序時直接呼叫 std::sort。



Binary Search Tree

主體為一棵二元樹，每個節點物件紀錄其左子節點(_left)和右子節點(_right)，保持其左子節點的資料恆小於等於本身、右子節點的資料恆大於等於本身，這樣設計可以使得中序(in-order)走訪的資料永遠保持排序的狀態。

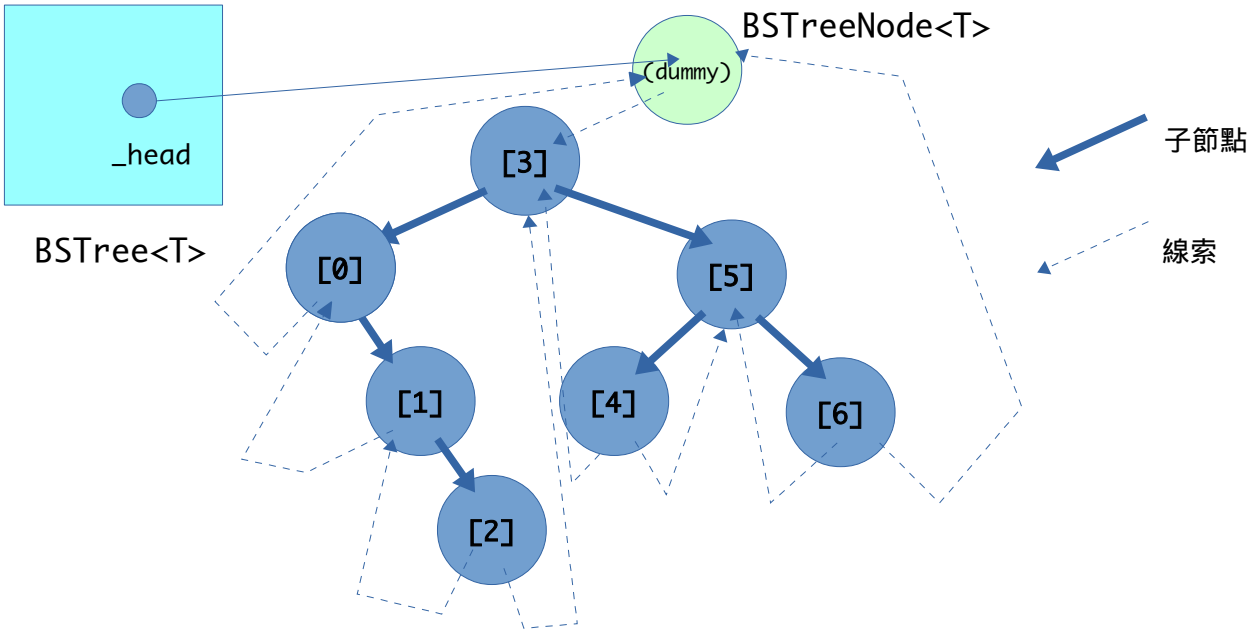
因為看到老師沒用 _parent，也想要試著實作看看，於是做出了 threaded 版本。左子樹或右子樹不存在的節點，分別將其指向中序走訪前一個、後一個節點，稱為線索(thread)，這樣可以加速前後走訪，而且插入、尋找不用遞迴，代價是多開了兩個 flag 值 _lthr、_rthr，來維護_left 和_right 這兩個指標是子節點(false)還是 thread(true)，可以用 bit-slicing 將這兩個值併進指標裡消除這個 overhead，但是來不及實作，目前只寫成幾個方法：has(Left|Right)Child 與 set(Left|Right)Flag，保留實作空間。

仿造 double-linked list 的設計，_head 依然為 dummy node，左子樹指向這棵樹的根節點，並且將中序走訪第一項的左線索和最後一項的右線索指向它。為了維持操作的一致性，_head 的左右節點都設計成線索。線索會增加實作的困難，但是可以稍微加速前後走訪，因為不用遞迴，執行時不會有大量的 stack。

新增資料時從根節點出發，向左或右移動直到樹葉為止，資料作為樹葉插入。刪除資料時，若為樹葉則直接刪除，若底下有一棵子樹直接替換，若底下有兩棵子樹，則將次大的節點(一定是葉子)換到這個位置上(因為太複雜所以直接用 T 的 copy constructor 寫，但效率上可能會變差)再遞迴呼叫刪除葉子。重複值的插入比較麻煩，採用的是一個性質：中序走訪的任意連續 3 個節點，必至少有 1 個是葉子，這樣可以保證若要插入的值已經存在時，這個節點與前後至少有一個位置可以插入，且不會影響到大小順序，因此只要依次嘗試即可。加線索對時間的 overhead 主要在刪除時的資料複製。

總而言之，採取這樣的作法，新增、走訪節點會比記 _parent 的版本來得快，但刪除會相對慢很多。(大量重複插入同樣的值可能會造成樹嚴重傾斜，想到可以加一個蹺蹺板 static variable 來決定遇到這種情況時該選左邊或右邊，並在選完之後 flip 它，但還未實作)

不用排序，資料永遠都保持有序狀態。



效能比較

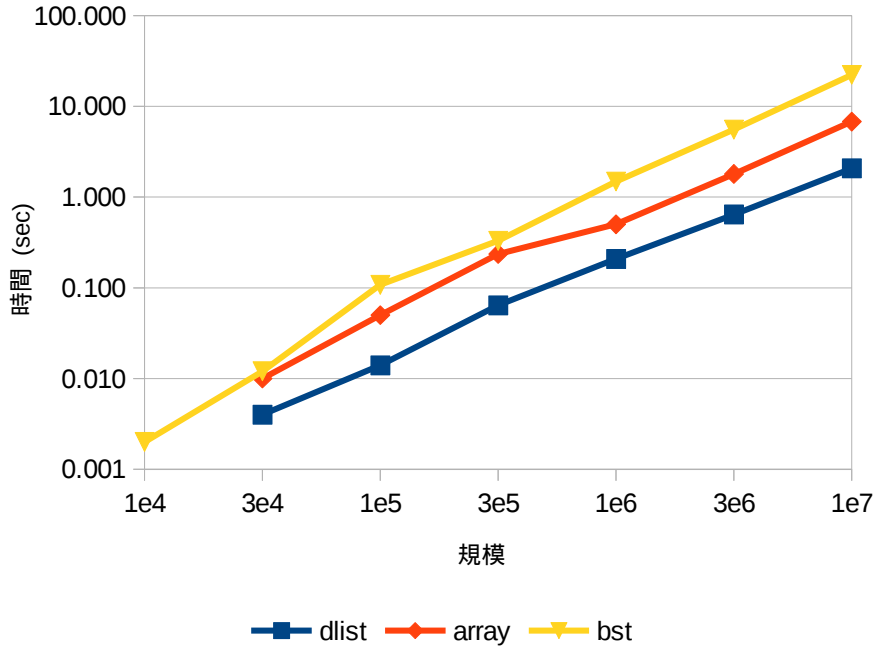
測試環境：Ubuntu 15.04，Intel i5，4GB RAM，Swap-off
每個測試做 5 次取其平均，每筆測試限時 300 秒。
時間、記憶體資料來源：程式內建的 USAGE 指令

一、空容器插入 n 筆資料所需時間

設計：用 `adta -r` 直接新增資料

預期：dlist 因為設計成環形，新增一筆的時間複雜度為 O(1)，新增 n 筆為 O(n)。array 會略大於 dlist，因為資料滿時需重新 allocate，需要花 O(n)的時間，均攤為 O(log n)。BST 插入資料類似二分搜，時間複雜度為 O(log n)。array 會略小於 BST，因為其只有容量不足的時候才會重新 allocate，每次 allocate 的記憶體大小為指數成長，因此隨著 n 越來越大會較不需要頻繁進行此操作。並且，其分配記憶體為連續的，應有利於 random access。

	dlist	array	bst
1e4	0.000	0.000	0.002
3e4	0.004	0.010	0.012
1e5	0.014	0.050	0.108
3e5	0.064	0.236	0.330
1e6	0.208	0.502	1.486
3e6	0.642	1.798	5.548
1e7	2.070	6.790	22.254



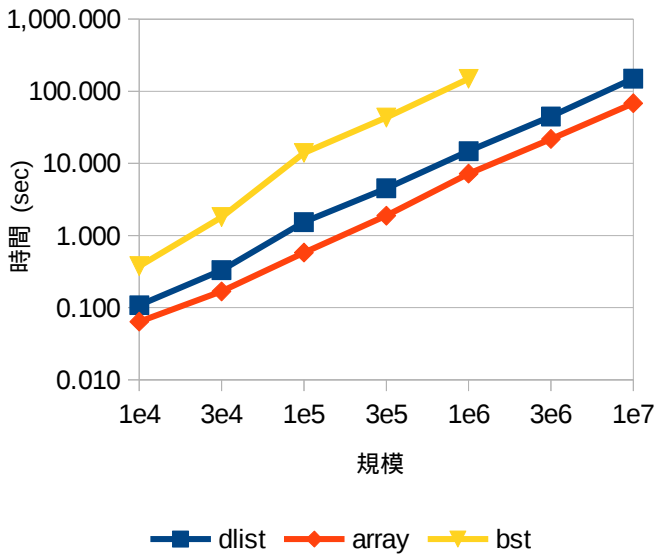
結果：dlist 為 O(n)，array 和 BST 皆接近 O(nlog n)，而且時間關係是 dlist < array < BST，符合理論預測。

二、在 n 筆資料裡交替增刪所需時間

設計：新增 n 筆資料，然後是 1000 次命令，奇數次為新增 1 筆(於尾部)、偶數次為刪除 1 筆(隨機)，計算命令的時間。

預期：單筆資料而言，dlist 新增刪除均為 O(1)、array 新增約略為均攤 O(log n) 刪除為 O(n)、bst 應為 O(log n)。

	dlist	array	bst
1e4	0.108	0.064	0.376
3e4	0.330	0.168	1.798
1e5	1.536	0.580	13.940
3e5	4.512	1.886	43.252
1e6	14.762	7.180	148.480
3e6	44.756	21.808	killed
1e7	149.540	68.092	killed



結果：dlist 在單筆資料的操作上亦為 O(n)，經過檢查 code 後發現是因為隨機刪除時需要先取長度、隨機選需要刪除的元素，走到需要的位置再刪除，而這操作是 O(n)。這部份是 `adtd -r` 指令的特性，而非 ADT 的效能問題。

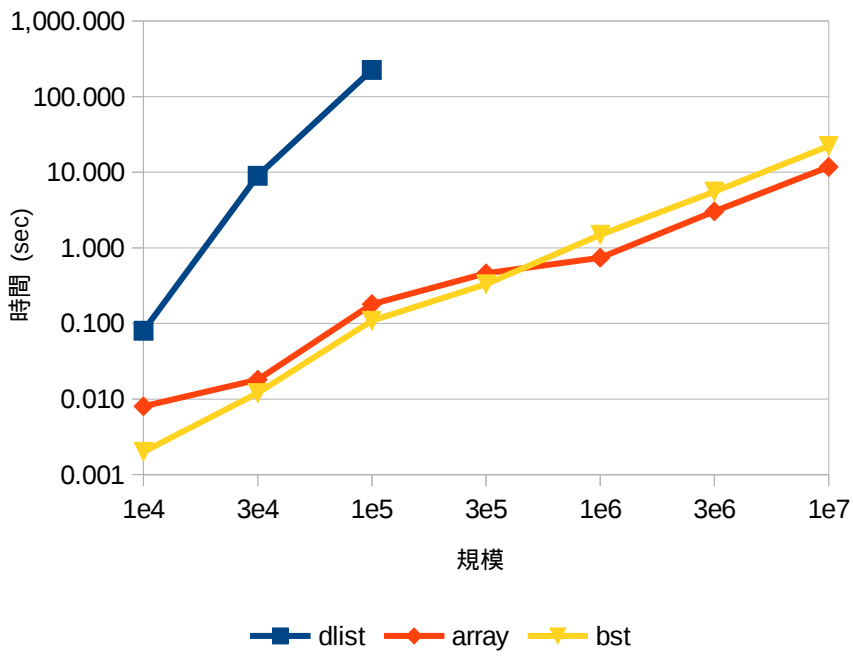
三、排序 n 筆隨機資料所需時間

設計：新增 n 筆資料，並且用 `adts` 進行排序，計算排序時間。BST 因為資料不必排序，直接取之前設計的實驗中新增資料的時間。

預測：dlist 用 bubble sort 所以是 $O(n^2)$，array 是 O(n log n)，BST 是 O(n log n)，但 BST 會略多，因為走訪會有 overhead，且 `std::sort` 有優化過所致。

結果：dlist 只做到 10^6 就做不下去了，沒有足夠資料可以分析。從小規模的資料看起來並不像是 $O(n^2)$，看起來像 $O(n^3)$ 甚至更多，非常奇怪。array 和 BST 的排序效能十分接近，但 array 的時間並不如 BST 均勻，這兩個都接近 O(n log n)，符合預測。

	dlist	array	bst
1e4	0.080	0.008	0.002
3e4	8.988	0.018	0.012
1e5	225.320	0.180	0.108
3e5	killed	0.460	0.330
1e6	killed	0.738	1.486
3e6	killed	3.026	5.548
1e7	killed	11.744	22.254



PS. double-linked list 在氣泡排序時，交換指標與交換資料的效能比較

設計：為了比較直接 copy 資料 T 或是改動其容器的指標何者效能較佳，寫了兩個版本，並分別測試其做 `tests/do2` 所花費的時間。copy 資料 T 不涉及容器的改動，但需一個 temp 的 T，並進行 3 次複製操作；改動容器指標需要進行 6 次(向前 3 次、向後 3 次)的指標操作，並在完成時將 iterator 倒退一格。

預測：交換指標的效能較佳。因為每 copy 一次 T 都需要呼叫一次 constructor，並且將內部的 string copy 一份，交換指標是單純的記憶體複製操作，應較省時。

結果：交換指標的效能較佳，並且其時間有顯著的差異。ref 約需 103 秒完成、交換指標版本約需 40 秒；交換容器版本約需 60 秒。採用交換指標，約減少 1/3 的時間。