

資料結構與程式設計 hw5 書面報告

B06901049 林泓均

一、資料結構的實做

Array (實做方式):藉由 `_data` 指標、`_size` 及 `_capacity` 三個 data member 實做出的資料結構。概念上，資料是儲存在藉由 `_data = new T[_capacity]` 所產生的大量指標中。Access 資料的方法則十分簡單，只需要利用 `_data[num]` 即可（須注意 `num` 要在 `_size` 的範圍內）。（我並沒有用 `_isSorted`，所以在 `find` 上面應當還是維持 $O(n)$ 的時間複雜度）

為何如此實做？：由於不在乎資料順序的關係，此種資料結構的速度相當快，無論是在 `pop` 或是 `push` 的表現上皆可達到 $O(1)$ 的水準。另外，其記憶體消耗不大，每多儲存一筆資料就是多使用一個 `_data` 指標產生的 `pointer` 而已，且 code 實做上也相當簡單。個人認為這種實做方法是相當理想的一種資料結構。

Dlist(Doubly linked list) (實做方式)：此種資料結構乃是藉由 `DListNode<T>` 進行資料的儲存。每個 `DListNode<T>` 裡面會儲存一筆資料(`_data`)以及另外兩個 `node(_prev, _next)`，分別指向此 `node` 的前一個 `node` 及下

一個 node。藉由此法可以把所有資料串連，只須設置一個 dummy node(`_head`)，使得第一個 node 的 `_prev` 及最後一個 node 的 `_next` 連到 `_head` 即可完成。一般而言，吾人會利用

```
DListNode<T> *_node;  
_node->_data;
```

的方式取得資料，並且藉由 `_node->_next` 進行 traversal。

為何如此實做？ 和沒有 dummy node 的情況相比，有 dummy node 的時候，若知道 `_head` 的位置即可使 pop 和 push 皆有 $O(1)$ 的良好表現。另外，在 sort 的 function 中，個人選擇使用 quicksort 來進行，原因是因為其 worst case $O(n^2)$ 不常出現，一般而言還是在 $\Theta(n \log n)$ 的範疇下。而 mergesort 雖然有 $O(n \log n)$ 的優勢，但比起 quicksort 在實做上面比較困難，所以在不會影響速度太多的狀況下就決定用 quicksort 實做。

BST(Binary search tree) (實做方式)：在此資料結構中，資料在 insert 階段便會進行好 sort 的動作。資料是存在每一個 `BSTreeNode<T>` 之中，此 class 包含一筆資料 (`_data`) 和兩個 `BSTreeNode<T>` (`_left`, `_right`)，分別指向其兩邊的 child。第一個被 insert 進入的資料被稱作 `_root`，接下來被 insert 的資料會和原本 tree 中的資料進行比較，

若是較小會被分配到左邊，若較大或相等則去右邊（註：因為我的 erase 寫法是看被 delete 的 node 是否左右腳都有小孩。在被 delete 的 node 之 _left 和 _right 皆存在時，程式是尋找其 successor 而不是 predecessor，因此相等的 case 丟右邊會比較好處理）。

為何如此實做？：和多數人不同的是，我在 BST 的實做中 並沒有使用 dummy node、也沒有用 parent 以及 trace（我的 BSTree 多了一個 bool _change_root 的 data member）。這樣實做的好處是在基本概念上比較清晰，不需要對每一個 node 去 handle 其 _parent，或考慮其他的 data member。但缺點就是會出現不少例外狀況要考慮，在 debug 時著實花了不少心思。以下舉出幾個實做時遇到的難題：

(1) iterator 在 ++(--) 時會需要用到 _root 來找出比當前 node 之 _data 大的中最小的(小的中最大的)，但是 iterator 並沒有辦法得到 _root 的資料。

解決方式：在 iterator 裡面再多一個名為 _root_in_iter 的 data member，初始化 iterator 時同時告訴她 _root 的位置。

(2) 由於沒有 dummy node 也沒有 _trace，所以 end() 就不知道要回傳什麼，才能使得 --end() 回傳 BST 中的最大值。

解決方式：end() 直接 return NULL。但是在 iterator 的 -- 中，若是 pos==NULL，則把 node=NULL 更新成最大值

的位置。(是有點 tricky 啦，但至少在本次作業的指令下都可以運作的很好)

(3) 在 `clear()` 中執行 `erase(iterator pos)` 時，若 `pos._node == _root`，則在刪除 `_root` 之後，`pos._root_in_iter` 不會被更新到(因為我的 `clear()` 是採用 `erase(temp++)`，因此直接在 `erase()` 裡面更新 `pos._root_in_iter` 沒有用)。這個 bug 會使得日後用到 `_root_in_iter` 時產生不可預期的結果。

解決方式：額外設置一個 `bool _change_root` 在 `BSTree` 中，若 `erase` 時傳進的 `pos._node == _root` 則將這個 flag 設為 `true`，並在執行完 `erase` 之後於 `clear` 中把 `_root_in_iter` 更新為新的 `_root`。

總結：三種資料結構與操作上的不同

`Array` 比較特別，是直接利用指標存值，並用 `[]` 的 operator 取值，接近過去對指標功能的想法。而 `dlist` 和 `bst` 都是利用節點存值，每個節點皆是存一個資料和兩個 `node` 的指標。差別在於兩個 `node` 的指標指法並不一樣，前者是一個連前面令一個連後面，但後者的指標是接到他的兩個小孩。另外，`BST` 在 `insert` 的時候就已經進行好 `sort` 了，未來在 `find` 時理論上會具有 $\Theta(\log n)$ 的速度優勢。

二、實驗比較：測試在 `worst case` 下，各

個資料結構的表現

實驗設計：由於 average case 下的測資比較不好生，因此我決定測試各個資料結構的 worst case，也就是 Big O 的時間複雜度，檢驗和理論值的差異。

註：這邊的 ADT 是用自己寫的比較，而不是 reference code。

以下為各比較項目的實做方式：

Add -s:輸入 90000 筆 ADTA -s (10000~99999)的資料，比較
花費時間。

Add -r:輸入十次的 add -r 2000，比較花費時間。

Delete -all:刪除 add -s 產生的資料，比較花費時間。

Delete -f/-b/-r:刪除 add -s 產生的資料，比較花費時間。

Find(Query):一次輸入 90000 筆 ADTA -s (10000~99999) 的資料並找出 99999 的位置並比較時間。

每個實驗都會進行三次。

實驗預期：理論上的時間複雜度

[illegible]

實驗結果比較與討論：

資料結構	Add -s	Add -r	Delete -all	Delete -f	Delete -b	Delete -r	find
Array	1.3s	0.04s	0.01s	0s	0	0s	0
	1.23s	0.03s	0.01s	0s	0s	0s	0
	1.38s	0.03s	0.01s	0.01s	0s	0s	0
Dlist	1.22s	0s	0.01s	0s	0s	0.46s	0
	1.33s	0s	0s	0s	0s	0.5s	0
	0.11s	0.01s	0.01s	0s	0.01s	0.47s	0
BST	46.19s	0.02s	0.01s	0s	0s	7.01s	0
	51.10s	0.02s	0s	0s	0.01s	7.17s	0
	50.57s	0.02s	0s	0s	0s	6.88s	0

ADTA -r、Delete -all/-f/-b 我也不知道為什麼 BST 速度會這麼快...這個部份相當值得討論...

在 `adta -s` 的部份，BST 的速度明顯落後於 Array 以及 Dlist，符合預期表現。

另外可以明顯看出差異的部份則是於 `delete -r` 的部份。此處 BST 速度依然落後於 Dlist 以及 Array。又，Dlist 相較於 Array 又需要多花費一點時間才能 `delete` 完，這是和 Array 比較時沒思考到的部份。在 `AdtTest.h` 裡面，可以看出 `delete -r` 時呼叫的函式應為 `erase(iterator pos)`。而 Array 在此 function 下只需要改變 `_size` 和移動 `_data`，Dlist 則需要額外執行 `delete` 的動作，BST 除了 `Delete` 以外需要先執行 `find_parent` 的函式，因此需時最久，符合預期。

Find 的話推測是因為找的時間真的太快了，所以看起來沒有差距，甚至很難和 $O(1)$ 區別。