ADT Comparison Report

B06901111
電機二
張瑞騰

一.資料結構實作

A. Doubly link list

Doubly link list是一種有點像連連看的資料結構，透過雙向的指標將多筆資料連在一起。Doubly link list的實作是透過DListNode這個物件來來將每一一筆資料包成節點，裡面面裝入資料 **_data**、下一筆資料的位置 **_next**，以及上一筆資料的位置 **_prev**。

在這次實作中我將_head設為記錄開頭的位置，同時它也是dummy node，dummy node的特性是他的下一個會指到自己，上一個則可以指到最後一筆資料，將所有資料串串成一個環，如此一來可以快速取得後面的資料。

※size()：

由於Doubly link list並沒有紀錄size的大小，因此計算size時必須從頭到尾走一一遍，時間複雜度是O(n)。

※push_back() / pop_back() / pop_front() / erase()：

在插入新的資料時得確保與前一筆資料進行雙向連結，同時也得跟_head連在一起；在刪除節點時，則要確保此點的前後有重新設定連結，跳過自己連在一起，才能夠刪除。

這樣可以確保資料的順序不會因為刪除而而有所改變，因此刪除資料並不會需要重新排序。

※sort()：

在此程式中是使用bubble sort 來實作，但由於bubble sort 的本身的複雜度是$O(n_2)$，相對於Array的O(n log n)慢非常多，因此在資料比較大的情況下會跑非常久(如後面面的實驗)。

※find() / clear():

find()跟clear()的原理類似，都需要一個節點走向下一個節點來完成指定動作，因此是O(n)。

※Iterator:

Doubly link list的iterator是透過將 _node 包起來來來來實現，iterator的 ++ 和 -- 則是將node移往其本身前後所指向的位置，begin()是回傳 _head 的下一個的 iterator，end()則是回傳 dummy node 也就是 _head 的 iterator。

B. Dynamic array

Dynamic array是一種記憶體相連的資料結構。Array本身會記錄兩個資料：_size 和_capacity。_size 是用來記錄當前的陣列大小，可以用來判斷最後一筆資料的位置，而_capacity 則是代表當前可以存放的空間大小小，當_size 等於_capacity 的時候，便無法再加入資料，此時得再擴大_capacity的容量。

當陣列空間存滿時(_size == _capacity)，必須拓展_capacity 的空間，同時將原本的資料複製過去，再刪除原本的資料。而 _capacity 拓展的原則時放大成原本的兩倍，也就是 0->1->2->4->8->...依此類推，，這樣的設計可以讓資料在低於指數成長的情況下，不會一直塞滿而必須進行複製。

※ push_back() / pop_front() / pop_back()：

在新增資料時只需將資料推送進去儲存空間即可，同時記錄 _size 的大小，以便知道何時要拓展 _capacity 以及最後一筆資料的位置。

在刪除資料時，會將最後一筆資料跟將要刪除的資料進行交換，如此可以到達O(1)的複雜度，如果要用讓這項資料後面的資料全部往前一格則會變成O(n)。雖然這樣一來會使原本的排序亂掉，但是由於陣列列的排序只要O(n log n)，在大多數的形況下還蠻快的，因此我們還是使用這種可以快速刪除的方方式。

※sort()：

對於可以random access的資料，可以使用quick sort、merge

sort、heap sort等O(n log n)的方式來進行，在此程式當中使用 algorithm 的quick sort 來來實作。

Iterator:

陣列的 iterator 相當簡單，移動時只要讀取隔壁記憶體的資料即可，由於相鄰的記憶體往會存在flush裡面，因此使用陣列來提取資料往往會比較快速。

C. Binary search tree

Binary search tree 是一種將資料以插入二元樹中所形成的資料結構，其特色就是資料在插入時便已經完成排序。

Binary search tree的實作也是透過節點將資料包起來來，此次設計中跟dlist不同的是，我在節點設定了_data、leftchild 和 rightchild，以及parent。

另外設定了 _root 跟 _tail 這兩個點，_root 是最頂端的點而 _tail 則是dummy node，dummy 跟 _root的關係是 _root->rightchild = _root->parent = _tail，_tail->rightchild = _root，_tail 的左端為0，這樣方便在iterator下辨別節點是否為dummy node。

使用這種方式實作主要是因為比較直覺，寫起來思路比較容易一點，不過麻煩的就是要把 parent 和 child 之間的關係處理理好，確保在做動作時沒有連結斷掉，否則就容易crush。

※insert() / erase()：

在insert()中，如果原先是empty()的狀態，則須將此點做好跟dummy連結的步驟，其餘狀況則都用同種方式，先設定哨兵(soldier)、準父母(preparent)，利用哨兵來探索插入資料的大小需放在哪個節點，而準父母會一直跟在哨兵身邊，一但soldier指到空時，代表可以插入這個資料了，資料會根據preparent來決定成為leftchild或rightchild。

erase() 要分成三種情況來來考慮：節點沒有child、有一個child和兩個child。前兩個分別用delete_case1()、delete_case2()這兩個function來

處理。而當有兩個child 時就需要尋找successor，找到後把資料交換，再用delete_case2()把successor刪掉。

※clear() / size()：

這兩個函式方法類似，都是利用iterator來完成指定動作，皆從起點(begin())開始往 _tail 移動。

※Iterator：

BST 的iterator相較於Array和DList還要來的複雜很多，這邊採用的方法，以++為例，先尋找有沒有rightchild，有的話下一個便是它的successor，沒有的話就往上找，重複同樣的動作，直到找到rightchild，並找到下一個最左的節點。

2. 實驗比較

A. 實驗設計

速度測試：針對三種資料結構，測試其在不同的資料量量下(1k，10k，100k，1M)，不同動作(add、delete、sort、print)的時間，做成表格比較。

記憶體測試：測試不同資料量量下所消耗的記憶體，並製成表格。

B. 預期

速度測試：

Add	Array > DList > BST
Delete	Array > DList >> BST
Sort	BST > Array >> DList
Print	Array > DList > BST

C. 實驗結果與討論

時間(s)	DList				Array				BST			
資料數	1k	10k	100k	1M	1k	10k	100k	1M	1k	10k	100k	1M
Add	0	0.	0.02	0.15	0	0.01	0.04	0.31	0	0.01	1.86	1.23
Delete	0.01	0.2	29.47	TLE	0	0.01	0.02	0.16	0.03	1.86	350.1	TLE
Sort	0.04	2.67	223.1	TLE	0	0.01	0.07	0.76	0	0	0.08	0.01
Print	0	0.01	0.07	0.75	0	0.01	0.06	0.65	0	0.01	0.08	0.92

記憶體	DList				Array				BST			
資料數	1k	10k	100k	1M	1k	10k	100k	1M	1k	10k	100k	1M
記憶體(M)	0	1.137	6.812	61.73	0	1.879	8.84	65.04	0	1.066	6.961	61.75

由實驗結果可以知道，在delete方面BST>>DList>>Array；Add三個都差不多，BST稍慢；而sort方面DList則是非常慢，BST因為在插入資料時就有進行分類所以在sort方面非常快。

從這邊可以看出來三種資料結構分別的特色：BST適合拿來做需要持續的排序的情況，但不能有太多的刪除動作；DList則是在存入資料的時候稍快，但刪除和搜尋都有點慘不忍睹；而Array 則是平均素質優秀，做適合拿來來做各種不同的應用用。

3. 結論

整體而言， Array的表現最優秀，雖然有些項目並非最快，但其實在100萬資料量量的情況下其實也差不到一秒，因此最好用的還是Array， Array在消耗記憶體方面稍大，但其實三者都在同一個數量量級之下，也沒有太大的差距，就實驗結果來看Array是比較好用的工具。