資料結構與程式設計 hw5 書面報告

B06901049 林泓均

1. **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)不常出現，一般而言還是在Θ(nlogn)的範疇下。而mergesort雖然有O(nlogn)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。

1. 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）

1. 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Θ(logn)的速度優勢。

1. **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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 資料結構 | Add -s | Add -r | Delete -all | Delete-f/-b | Delete-r | find |  |
| Array | O(1) | O(1) | O(1) | O(1) | O(1) | O(n) |  |
| Dlist | O(1) | O(1) | O(1) | O(1) | O(1) | O(n) |  |
| BST | O(n) | O(n) | O(n) | O(n) | O(n) | O(n) |  |

**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...(對不起...)

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。