**DSnP HW5 Report**

B07901016 朱哲廣

ㄧ、實作：

1. Array：

以\_capacity與\_size來控管array的大小，若在push\_back時，\_size == \_capacity，即將\_capacity放大兩倍，再將原本的陣列複製過去，用以做到動態調整陣列大小。

pop\_back的部分是將其\_size減少，而再次新增的時候就會直接覆蓋原來的值，而不用重新delete或new。

pop\_front跟erase都是將目標的值與最後一個值做交換，再減少\_size。

1. Dlist：

設一個\_head在剛建構時，會指到一個dummy node，在加入新的節點後，會 將最後一個節點的\_next指到dummy，而dummy的\_prev會指到第一個點，以此來實作begin()與end()。

Sorting的部分，我是以bubble sort實作，所以會有n平方的複雜度。

push\_back\pop\_back\pop\_front\erase的實作皆是找到目標後，對其前後的節點重新指\_prev跟\_next來實作。

1. BST：
   1. \_dummy位置：

我在每一個BSTree中設一個\_dummy node，其用途為end()用，而\_dummy的左右子樹都指到\_root，也就是我們樹的根。

* 1. Iterator::\_trace：

我用vector開一個\_trace，裡面存的是一個pair，分別存BSTreeNode<T>\*與TrStat，意思是路過的點以及其怎麼走過來的(TrStat是一個enum型態，內有LEFT\CUR\RIGHT)，而用vector不用stack的理由是vector有clear()函數可以使用，比較好實作destructor。

\_trace中存取範例：

node1

node2

node3

|  |
| --- |
| {node3, CUR} |
| {node2, RIGHT} |
| {node1, LEFT} |

* 1. Iterator::++：

++找的是「第一個比此節點大的點」，因此搜尋範圍會是「右子樹中最小的點」，使用的是\_findMin(\_node->\_r)這個private function。

而若是沒有右子樹，那就從\_trace當中找到「第一個是走LEFT過來的點」，這個點即是第一個比他大的點，作法是從\_trace中一個一個pop出來找。

特例：若是沒有右子樹，\_trace中又沒有從任何一個點走LEFT來，那可以知道此點要到達的位置是end，所以呼叫iterator::end()，將此iterator設為end()，但此特例會在後面被提到並且被解掉。

* 1. Iterator::--：

--找的是「第一個比此節點小的點」，因此搜尋範圍會是「左子樹中最大的點」，使用的是\_findMax(\_node->\_l)這個private function。

而若是沒有左子樹，那就要從\_trace當中找到「第一個是走RIGHT 過來的點」，這個點即是第一個比他小的點，作法是從\_trace中一個一個pop出來找。

特例：若是沒有左子樹，\_trace中又沒有從任何一個點走RIGHT來，那可以知道此點要到達的位置是begin，所以呼叫iterator::begin()，將此iterator設為begin()。

* 1. 在iterator::\_trace中的\_dummy：

在每一個iterator中，他的\_trace都會先被放入{\_dummy, LEFT}。此做法是為了因應在end()--時，會搜尋左子樹中最大的點的特性，也就會找到從\_root開始找最大的點，也就是整棵樹的最大點。而在最大點做++的時候，也會一直往上找第一個LEFT來的點，也會回到\_dummy中，也就可以少實作iterator::++中的特例！亦可符合我們end()的用法。

* 1. Iterator constructor中BSTree<T>\* b的用意：

在實作過程中，我碰到invalid use of non-static data member的問題，此原因是因為nested class無法直接讀去外部的data member，一個iterator並不會知道其對應的Tree是誰，所以要將其會用到的BSTree<T>\*傳入，以得到相應的\_root跟\_dummy node。

* 1. BSTree::erase(iterator pos)：

這個function可說是我實作pop\_front\pop\_back\erase中都會用到的function，以達到寫一個function來實作很多功能。此erase邏輯是：

「若是有右子樹」：則尋找右子樹中數值最小的點，並與其點交換「數值」，並將要刪除的pos指到這個點，用意是讓要被拔掉的點都只有1個或0個子樹。

「若只有一個子樹或沒有子樹」：那做法是從\_trace中找到其上一個點(parent)，看他是走哪個方向來(LEFT\RIGHT)到此點，將其對應的子樹指到要被刪除的點的子樹，再將要被刪除的點delete，即完成我們的刪除。

* 1. BSTree::\_size：

此member會在insert與erase時作更新，以儲存樹的大小，以更快拿到樹的大小。

二、實驗：

1. 實驗方式：

分別生成10/100/1000/10000/100000/1000000個node，並比較各個指令的時間與reference code的差異，與其所佔空間的差異

1. 實驗結果：

運行環境：Ubuntu 18

* 1. adtAdd (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Array | 0 | 0 | 0 | 0 | 0.01 | 0.16 |
| 0 | 0 | 0 | 0 | 0.01 | 0.16 |
| Dlist | 0 | 0 | 0 | 0 | 0.01 | 0.09 |
| 0 | 0 | 0 | 0.01 | 0.02 | 0.08 |
| BST | 0 | 0 | 0 | 0.01 | 0.05 | 1.32 |
| 0 | 0 | 0 | 0 | 0.06 | 1.64 |

* 1. adtSort (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Array | 0 | 0 | 0 | 0 | 0.03 | 0.33 |
| 0 | 0 | 0 | 0 | 0.03 | 0.33 |
| Dlist | 0 | 0 | 0 | 0.8 | 100.8 | x |
| 0 | 0 | 0 | 0.75 | 87.92 | x |
| BST | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

* 1. adtDelete (執行 ”adtDelete -s <string>” 10000 or size次) (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10  (size) | 100  (size) | 1000  (size) | 10000  (size) | 100000  (size) | 1000000  (size) |
| Array | 0 | 0 | 0.01 | 0.03 | 0.83 | 16.45 |
| 0 | 0 | 0.01 | 0.03 | 0.9 | 16.04 |
| Dlist | 0 | 0 | 0.01 | 0.06 | 1.7 | 22.83 |
| 0 | 0 | 0 | 0.18 | 1.58 | 24.34 |
| BST | 0 | 0 | 0.02 | 0.06 | 0.07 | 0.21 |
| 0 | 0 | 0.01 | 0.02 | 0.21 | 0.21 |

* 1. adtDelete (執行 “adtDelete -r 10000”) (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10  (size) | 100  (size) | 1000  (size) | 10000  (size) | 100000  (size) | 1000000  (size) |
| Array | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| Dlist | 0 | 0 | 0 | 0.13 | 2.5 | 30.36 |
| 0 | 0 | 0 | 0.13 | 2.43 | 33.31 |
| BST | 0 | 0 | 0 | 0.41 | 26.31 | 396.4 |
| 0 | 0 | 0 | 0.43 | 21.58 | 362.7 |

* 1. adtQuery (query 10000 or size次的時間) (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10  (size) | 100  (size) | 1000  (size) | 10000  (size) | 100000  (size) | 1000000  (size) |
| Array | 0 | 0 | 0 | 0.18 | 1.08 | 13.21 |
| 0 | 0 | 0.02 | 0.16 | 0.8 | 15.89 |
| Dlist | 0 | 0 | 0 | 0.1 | 1.43 | 24.17 |
| 0 | 0 | 0 | 0.16 | 1.37 | 26.48 |
| BST | 0 | 0 | 0.02 | 0.06 | 0.1 | 0.1 |
| 0 | 0 | 0 | 0.09 | 0.13 | 0.11 |

* 1. adtReset (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Array | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| Dlist | 0 | 0 | 0 | 0 | 0 | 0.01 |
| 0 | 0 | 0 | 0 | 0.01 | 0.01 |
| BST | 0 | 0 | 0 | 0 | 0.03 | 0.29 |
| 0 | 0 | 0 | 0 | 0 | 0.13 |

* 1. adtPrint (s) (單位：s)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Array | 0 | 0 | 0 | 0 | 0.02 | 0.35 |
| 0 | 0 | 0 | 0 | 0.02 | 0.26 |
| Dlist | 0 | 0 | 0 | 0 | 0.04 | 0.39 |
| 0 | 0 | 0 | 0 | 0.03 | 0.37 |
| BST | 0 | 0 | 0 | 0.02 | 0.08 | 0.91 |
| 0 | 0 | 0 | 0 | 0.09 | 1.32 |

* 1. Total Memory used (單位：M)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mine  ------  ref | 10 | 100 | 1000 | 10000 | 100000 | 1000000 |
| Array | 0 | 0 | 0 | 2.137 | 7.496 | 49.6 |
| 0 | 0 | 0 | 2.266 | 7.625 | 49.73 |
| Dlist | 0 | 0 | 0 | 2.277 | 7.691 | 62.61 |
| 0 | 0 | 0 | 2.266 | 7.68 | 62.59 |
| BST | 0 | 0 | 0 | 2.254 | 7.668 | 62.58 |
| 0 | 0 | 0 | 2.32 | 7.734 | 62.65 |

三、實驗結果分析：

1. adtAdd: BST >> Array ~ Dlist
2. adtSort: Dlist >> Array > BST
3. adtDelete -s <string>: Dlist > Array >> BST
4. adtDelete -r 10000: BST >> Dlist > Array
5. adtQuery: Dlist > Array >> BST
6. adtReset: BST > Dlist ~ Array
7. adtPrint: BST > Dlist ~ Array
8. Total Memory Used: BST ~ Dlist > Array

四、實驗結果討論：

BST在新增資料時的效率比起其他兩個來得慢，因為他會有logN的插入複雜度，但是在排序、查找跟刪除茶找到的點，都比其他兩個來的快，可以快速對應需要大量排序、查找功能場合，但在給定pos的刪除上效率很差，原因是要再重新找到iterator在樹中的位置跟生成其\_trace。

Array的特性為其所用空間較小，而且在排序上也有不錯的表現，但查找的速度會相較慢一些，查找複雜度是O(n)。

Dlist在Sorting上的複雜度最大O(n^2)，因此是最慢的，而且在刪除跟查找的效果也沒有BST跟Array好，但若有需push\_front的情況，dlist會較Array還要快。