DSnP HW5 Report

B07901016 朱哲廣

一、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。

2. 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。

3. BST：

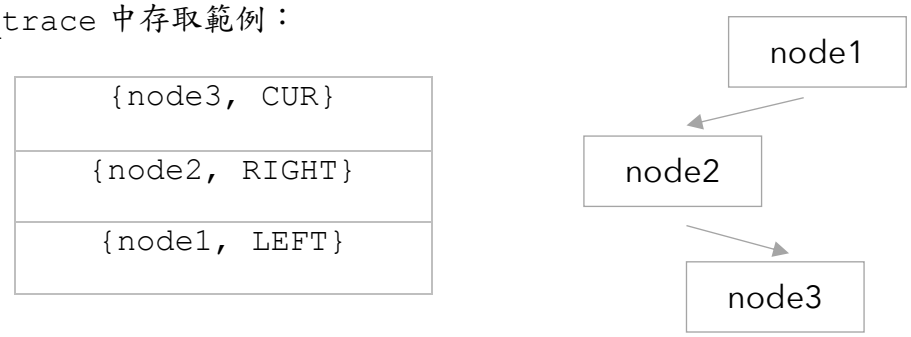
A. _dummy 位置：

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。

B. 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：



C. 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。

D. 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()。

E. 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。

F. 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。

G. 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。

H. 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

2. 實驗結果：

運行環境：Ubuntu 18

A. 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

B. 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

C. 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

D. 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

E. 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

F. 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

G. 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

H. 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。