HW5
Abstract data structure test report

Orange Hsu (徐子程)
B06602037
2018/12/1

1-1資料結構比較

資料結構	Dynamic array	Doubly linked list	Binary search tree
存取方式	可隨機存取	循序存取	binary search
插入	從尾端（否則要移動整個array）	從任意位置	依插入元素與現有資料的大小排序
記憶體空間	連續	連續/離散	連續/離散
適用	大量插入資料, 透過索引存取資料	隨機插入資料	排序，搜尋資料

實做方式

資料結構	Dynamic array	Doubly linked list	Binary search tree
`size()`	使用_size變數隨時更新	由於沒有_size變數可用，每次呼叫iterator traverse一次	使用_size變數隨時更新
`empty()`	檢查_size是否==0	檢查_head是否指向dummy node	檢查_size是否==0
`push_back()`	若`_size == _capacity` 則使用`expand()`長大array 令`_data[i] = x`, 並讓`_size++`	先檢查list是否被初始化 無：新增一個node，讓此node的 `_prev,_next` = dummy node 有：則新增一node，令 _prev = last element, _next = dummy node	none

insert()	none	(選用)使用iterator pos指定insert的位置，新增一node，令_prev = pos->_node, _next = (++pos)->_node	檢查BST是否被初始化 無：新增一個node，讓此node->_prev = node->_next = node->_parent = dummy node 有：使用recursive尋找insert node的位置並新增一個node, node->_prev,_next = dummy node, _parent = 要插入的node
pop_back()	使_size--	利用一變數tmp紀錄list的last element(欲刪除的node)，令(tmp->_prev)->_next = dummy node, dummy node 的_prev = tmp->_prev 最後delete tmp (若list剩下一個node, 刪除之，並使_head =_head->_prev,_head _next = dumm node, 即回復未初始化狀態)	使用GetMax()找到最大值（Right most elemtnt），並取得其iterator，以erase(iterator)刪除之
pop_front()	令_data[0] = last element，將_size--	方法同pop_back()，將對象換成first element	使用GetMin()找到最小值（Left most elemtnt），，並取得其iterator，以erase(iterator)刪除之
erase(iterartor)	令pos的_data = last element的_data，將_size--	方法近似pop_back(), 針對以下兩種情況特殊處理 1. 僅剩1個element：恢復list為未初始化狀態 2.對象為first element：設定second elememt為_head	使用erase(x)傳入*pos進行刪除
erase(x)	使用find(x)尋找欲刪除的element的iterator，使用erase(iterartor)刪除之	使用iterator traverse整個list，至尋找到目標, 若找不到則傳回end()	先以search()搜尋是否有目標可刪除 對於一個node，刪除分下列三種情況 1.沒有child

			2.有一個child 3.有兩個children
`clear()`	令_size = 0	從_head開始，iteratively delete nodes，並將list設定為未初始化狀態	從_root開始traverse，recursively刪除nodes，後將BST設定為未初始化狀態
`iterator find(x)`	對_data[i]進行linear search	使用iterator對list進行linear search	Recursively 進行 binary search
`sort()`	使用STL內建`sort()`，設定`_isSorted = true`	使用quick sort，以last elemect為pivot，設定`_isSorted = true`	插入資料時已排序完成，不須再排序

實做方式說明

Array

`size()`：在`push_back(),pop_back(),pop_front(),erase(),clear()`時直接`++`或`--_size`。
優點：動態更新`_size`，不用每次traverse一次array。
缺點：需要留意會更動到`_size`的操作，避免遺漏。

`sort()`：改為當`_isSorted == true`，不再重新sort。
優點：已排序完成的資料不用再排序一次，節省時間。
缺點：需要留意會更動資料順序的操作 (`erase()`...)。

額外實做的function

`expand()`：在array的capacity不足時，自動擴大capacity。

Doubly linked list

`empty()`：不直接traverse整個list，改為檢查`_head->_next`是否 `== _head`，因為只有在list尚未初始化，上述條件才會符合（dummy node指向itself）。
優點：節省traverse時間。
缺點：不直觀。

`size()`：由於無法自行新增`_size` data member並動態更新，每次呼叫時，traverse一次list。此舉影響效能重大（time complexity O(n)），故往後判斷list是否為空時盡量使用`empty()`內的方法（time complexity O(1)）。

`sort()`：原先使用bubble sort，後改用quick sort。經測試，有顯著效能改善。
優點：節省大量時間（time complexity $O(n^2)$ v.s. $O(nlog(n))$，worst case $O(n^2)$。
缺點：quick sort不好實作。

以下是測試 tests/do2 所花費時間(unit:s)的比較

Platform	bubble sort	quick sort	Ref.
1	11.48	2.53	17.32
2	13.78	1.72	13.94

Binary search tree

此BST的Node `BSTreeNode<T>`設計為有兩個children `_left`，`_right`和一個`_parent`。
本tree使用dummy node，在BSTree的constructor被呼叫時，建立一dummy node（pointer為`_dummy`），children與parent指向自己，此時BST為空（尚未初始化）。

`insert()`：插入第一個element時，設為`_root`，並將兩個children與parent指向dummy node；dummy node則將兩個children指向`_root`。
插入第二個（以上）element，則按照binary tree的規則，使用recursive的方式尋找適當的位置，並插入，最後將`_left`，`_right`指向dummy node（此作法類似紅黑樹將leave指向NIL）。

對於插入兩個相同的value，採用放置到node左側的方式實做，分下列兩種情況：

1. 無left children→直接插入
2. 有left children→將left children分開，插入後再接回

此作法是較容易實做的版本，但是也較浪費記憶體空間。較佳的作法是利用node裡的變數紀錄重複的次數，但是此方法在traversal時(特別是使用iterator)需要處理的方式較麻煩，故沒有採用。

額外實做的function

`GetMax()`：traverse至最右側，取得最大element的pointer。
`GetMin()`：traverse至最左側，取得最小element的pointer。

實驗比較

本文所有測試皆在以下硬體條件下完成

Platform	1	2
CPU	Intel i7-8550U	Intel Xeon 1230-V2
RAM	12G DDR4 2133	16G DDR3 1866
Compiler	g++ 7.3.0 with -O3 flag	g++ 7.3.0 with -O3 flag

1. 實驗設計

使用Brian Chao同學提供的測資生成程式，可調整各個指令（adta', 'adtd', 'adtq', 'adtr', 'adtp'）生成的比例，以測試不同存取方式與資料結構之間的關係。

實驗條件：

生成10000筆測資，測資生成長度為5的亂數字串，以下分別列出測資內['adta', 'adtd', 'adtq', 'adtr', 'adtp']所佔的比例

1) 隨機插入
   [0.6, 0.1, 0.05, 0.05, 0.2]
2) 搜尋
   [0.1, 0.1, 0.55, 0.05, 0.2]

2. 實驗預期
   1) Linked list應會有較好的效能，Binary search tree次之
   2) Binary search tree應會有較好的效能，Array次之

表格內依序為Array/DList/BST的My/Ref.所花費的時間(s)

3. 結果討論

Platform	Exp. 1	Exp. 2
1	3.83/4.57 10.42/26.42 17.39/28.92	0.47/0.53 0.68/0.82 0.77/0.82
2	5.15/5.4 11.21/23.83 14/16.36	1.34/1.53 1.44/2.05 1.85/2.39

由實驗1結果可以得知array有較佳的效能，與預期相反，推測是因為array沒有要求實做從中間insert的function，只能從back insert，故dlist無法顯現優勢。

由實驗2結果可以得知整體花費時間差異不大，但array仍有較佳的效能，推測可能是與BST實做方式有關。