DSNP HW5 report

電機三 B04901020 解正平

一、資料結構的實作

1. ADT 操作差異：

<Double Linked List>

每個資料之間像是環串在一起，具有既是起點也是終點的 dummy node，可以很快 access 到 front 還有 back，加減 node 只需要改變他的前後關係，附圖為課堂上大概提到的時間複雜度，可以很明顯看到除了 sort 跟 find 其他幾乎都是O(1)。

Complexity Analysis (Doubly Linked List)

- push_front() O(1)
  push_back() O(1)
  pop_front() O(1)
  pop_back() O(1)
  size() O(n) or O(1)
  empty() O(1) // != (size() == 0)
  insert(pos, data) O(1)
  erase(pos) O(1)
  find(data) O(n) ⟵
- Sorting on Linked List: $O(n^2)$
  - Bubble sort, selection sort, etc.

<Dynamic Array>

每個資料都存在 pointer 指到的陣列當中，需要存資料的時候會 delete 原始陣列再重新 new 一個夠大的陣列裝資料，另外處理資料的加減我們不會去管順序，只管陣列大小受影響，需要再用 sort 把資料順序整理過。

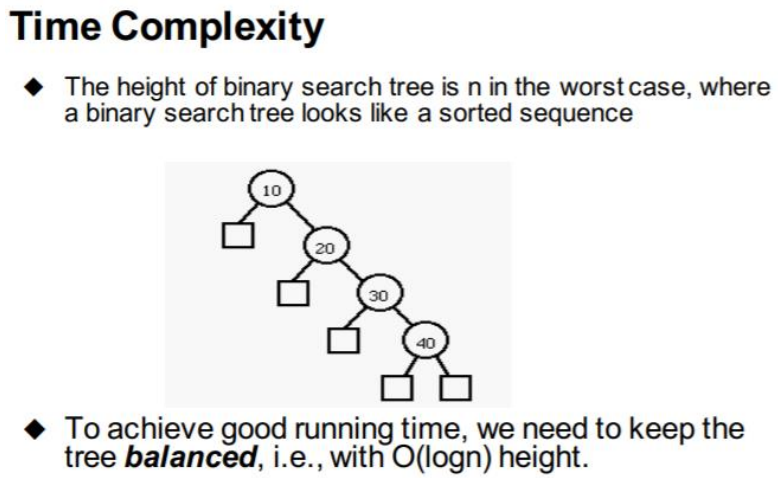
Complexity Analysis (Dynamic Array)

- push_front() O(n) or O(1) // if order not matters
  push_back() O(1)
  pop_front() O(n) or O(1) // if order not matters
  pop_back() O(1)
  size() O(1) // not O(n), why?
  empty() O(1)
  insert(pos, data) O(n) or O(1) // if order not matters
  erase(pos) O(n) or O(1) // if order not matters
  find(data) O(n) or O(log n) // why?

If order does not matter, almost all operations are O(1)!!

<Binary Search Tree>

資料以像是一棵每個點都兩個分支的樹，從根部的點分比較大往右長，比較小往左長，這樣的分法可以使得資料都是整理依照 order 整理過，它的每個 node 之間都有 leftchild 和 rightchild，很簡單可以 access 到需要使用的資料，這個型態好處是 find、insert 與 delete 相對時間複雜度較低約為O(logn)。



2. 程式方式：

<Double linked list> (pointer、next、prev、_head)

用 pointer 的方式連結每一個 node 之間的關係，每一個 node 會存它本身的資料以及 next node 及 prev node 的位置，這些東西用 class DListNode 包裝起來。接下來整串資料像是一個環一樣，可以從每點找到前一格及後一個，方便起見寫了 class iterator 來更簡單描述，但為了設立終止條件，我們加入一個_head 的 dummy node，使得整個資料環可以有既是起點也是終點的 node(同一點)，最後 class DList 寫一下會需要的函式，其中為了 sort 資料，我使用 bubble sort 的方法，可惜時間複雜度就會是$O(n^2)$。

<Dynamic Array>(pointer、new array、std::sort()、size & capacity)

用 pointer 指向可以存 capacity 大小的 array，若存取資料的 size 量超過 capacity，就會 delete array 然後 new array 兩倍大(2、4、8……)的 capacity 存回原本舊資料還有新的資料，這樣可以不用每次輸入資料都要索取新的記憶體浪費時間，但可能會因為資料量大浪費過多記憶體空間。另外資料 pop front 需將尾巴資料取代第一個，pop back 都不用只需讓 size 變小，erase 不用管 order 時間複雜度會大概是O(1)。

<Binary Search Tree>(pointer、left、right、parent、successor、tail)

每個 node 都存其 data 及 leftchild 位置、rightchild 位置和其 parent 位置，有 parent 可以使得 traversal 非常方便不用從 root 找下來浪費時間。另外為了 iterator 的 end，我加入了 tail 一個 dummy node 在最大 data 的 rightchild，但是因為這樣 iterator 的 ++和—不太好寫，而且 erase 及 insert 都要額外考慮，對寫 code 不太方便。size、clear、insert、erase 這些 function 我都使用遞迴的方法，原因是比較好懂，而且讓我練習遞迴的程式，尤其在 erase 找 successor 的時候會是將原資料以 successor node 取代，再讓 success node call 自己 erase。最後因為 tree 本身已 sort 不需另外特別想。

二、實驗比較

1. 實驗設計：

將三個資料結構以下列步驟測試 runtime

- ✓ add random 100,000 筆資料
- ✓ delete random 20,000 筆資料
- ✓ sort
- ✓ add random 100,000 筆資料
- ✓ pop back 10,000 筆資料
- ✓ pop front 10,000 筆資料
- ✓ delete random 20,000 筆資料
- ✓ sort

2. 實驗預期：

Add/Insert	Array($\boldsymbol{O(1)}$)<DList($\boldsymbol{O(1)}$)<BST($\boldsymbol{O(\log n)}$)
Delete/erase	Array($\boldsymbol{O(1)}$)<DList($\boldsymbol{O(1)}$)<BST($\boldsymbol{O(\log n)}$)
Sort	BST(0)<Array($\boldsymbol{O(n\log n)}$)<DList($\boldsymbol{O(n^2)}$)

3. 結果比較與討論：

-g/-O3	DList	Array	BST
Add -r 100000	0.01/0.0	0.04/0.02	0.09/0.06
Delete -r 20000	8.87/6.76	1.53/0.0	109/84.02
Sort	220.6/76.47	0.05/0.03	0.0/0.0
Add -r 100000	0.02/0.01	0.04/0.02	0.12/0.11
Popback 10000	0.0/0.0	0.0/0.0	0.01/0.01
Popfront 10000	0.0/0.0	0.0/0.0	0.02/0.0
Delete -r 20000	25.48/24.97	2.63/0.01	193.4/175.4
Sort	738.9/280.3	0.09/0.04	0.0/0.0
Total run time	993.9/388.5	4.38/0.12	302.6/259.6
Memory used(MB)	12.38/12.3	12.99/13.04	12.04/11.96

實驗結果與預期大小關係差不多，然而在 add 時；DList 時間比 array 少。總分析三者，array 因為沒有 order 幾乎時間都是$\boldsymbol{O(1)}$；DList 在本實驗除了 sort 時間是$\boldsymbol{O(n^2)}$，在 delete 的部分雖然預測時間是$\boldsymbol{O(1)}$，但並不像 array 來的少，可能是寫法不太好；BST 多取決於 tree 的高度，運算時間大多為$\boldsymbol{O(\log n)}$相對高於其他兩者。