# DSnP—Homework 5 Report

R08921053

- 以下實驗的 BST 部分參考 BST-linux16.ref 檔案，謝謝 Ric 老師的提供

**1. 實作設計方法和複雜度理論分析**

首先，在抽象資料結構(ADT)的設計上，扣除掉只與”AdtTestObj”成員函數相關的 reset，以及印出資料的 print，我們的重點放在 add, delete, find, sort 等四個功能上.

**a. 動態陣列(dynamic array)**

add: 如果_size==_capacity，則把_capacity 變成兩倍或是 0->1. 因此需要使用 new 呼叫一個兩倍大的陣列(或 0->1)，再把資料複製過去. 如果_size<_capacity，則不需要處理. 最後，再把_size 加 1，並在空出的格子放入一筆資料.

值得一提的是，在呼叫兩倍大的陣列(或 0->1)並且複製資料後，可以把舊陣列還給系統，達成優化系統與防止記憶體溢出(overflow)的目標，而後再把_data 指標指向新的陣列.

理論上，加入 n 筆資料，整體新增陣列的時間大約是 2n，每筆資料填入的時間是 O(1)，平均而言，每筆資料的時間還是 O(1)，當然這裡限制從末端加入資料.

delete: 實作上分成 pop_front(移除第一格), pop_back(移除最後一格), erase(iterator), erase(const &T)等四種功能. 為了簡化設計，我利用 find 功能把 erase(const &T)簡化成 erase(iterator). 而其餘的三種功能，都是把指定的格子用最後一格覆蓋掉，再把 size 減 1 就好.

值得注意的是，必須時時刻刻小心 size==0 和 size==1 的特例. 都有可能造成系統的崩潰. 而_capacity 則是都不需要改變.

特別地，在隨機刪除的過程中，需要讓 iterator 指定到第 x 個位置，再把 iterator 丟給 erase 運算. 由於 array 是隨機存取，此操作為 O(1)時間，所以整體還是 O(1)時間.

理論上, pop_front, pop_back, erase(iterator)單次都是 O(1)時間, erase(const &T)的時間複雜度取決於 find，所以是 O(n)，稍後介紹.

find: 我的方法是，直接遍歷一次 array 尋找鎖定對象. 如果沒找到就 return end.

順帶一提，如果確定是 sorted 的前提下，可以用二元搜索(binary search)把時間複雜度壓縮到 O(logn). 不過我並沒有實作 isSort 的功能.

理論上，遍歷一次的時間複雜度就是 O(n).

sort: 我直接套用 std 內建的排序函數(sort function). 也就是經過優化後的 quick sort 演算法.

理論上, 時間複雜度平均是 O(nlogn), 最差會到 O(n^2). 空間複雜度也是介於 O(logn)到 O(n)之間, 取決於遞迴所需要的 stack 空間.

**b. 雙向連結串列(double linked list)**

add: 一律把新增的 node 加到末端. 如果 size==0, 也就是容器內只有 dummy node, 那麼除了把使用 new 呼叫的 node 和 dummy 連接上, 還要讓_head 指向新的 node. 如果 size!=0, 那麼只要處理好 dummy node, dummy node's _prev, 新增的 node 這三者之間的指向關係就好了.

值得一提的是, 我這邊採取 Ric 老師的小技巧, 使用 bool size0 來判斷容器的狀況, 進而分類處理問題.

理論上, 每次添加資料到末端的時間複雜度為 O(1).

delete: 同樣分成 pop_back, pop_front, erase(iterator), erase(const &T)這四種功能. 在 pop_back 和 pop_front 部分, 如果 size=0, 什麼都別做. 如果 size=1, 則把唯一的點移除, 並讓 dummy 指向自己, _head 指向 dummy. 如果 size>=2, 那麼就把 dummy->_prev 的前後鄰居(dummy, second last)或是_head 的前後鄰居的指向關係處置好就行. 並且小心_head 新指向的對象. erase(const &T)同樣可用 find 簡化成 erase(iterator)的情況. 最後, erase(iterator), 同樣用 size 來分成三個情況. 如果 size==0, 則什麼都別做. 如果 size==1, 則移除唯一的點, 把 dummy 指向自己, _head 指向 dummy. 如果 size>=2 而且 iterator 就是_head, 那就是 pop_front, 如果 size>=2 而且 iterator 不是_head, 那麼把前後鄰居的指向關係處理好就行.

需要注意到, 可以使用 delete 刪除不會再用到的 node, 減少記憶體的使用. 而 size==0 的特例也是我們特別需要注意的情況.

特別地, 在隨機刪除的過程中, 需要讓 iterator 指定是第 x 個位置, 再把 iterator 丟給 erase 進行操作. 由於 dlist 是線性存取的, 所以需要 O(n)的時間.

理論上, 這四種功能, 都只有改變前後鄰居的指向關係, 頂多再改變_head, 或是多一次 delete, 所以單次操作都是 O(1)時間. 當然, 使用到 find 的 erase(const &T)的時間複雜度等同 find, 將會是 O(n)時間.

find: 基本上, 就是遍歷一次所有的 node, 若沒找到就 return end.

順帶一提, 由於 linked list 不是隨機存取, 所以就算 sorted, 也無法做二元搜索(binary search), 不可能達到 O(logn)的時間複雜度.

理論上, 平均的時間複雜度是 O(n), 因為大概要走過一半的 node.

sort: 我採取選擇演算法(selection sort), 也就是每次遍歷未排序的 node, 再把最大/最小值, 加入已經排序的部分. 剛開始, 我的方法是每次找到最小值, 再把它 push_back, 再 erase(iterator), 重複做 n 次這個操作, 就可以在現成的功能下, 最

不花費力氣地完成 sort 運算. 然而, 這樣會改變_head, 因此和 const 互相牴觸. 再後來, 參考老師提供的”只交換資料不刪除頂點”的方式, 我在每回合挑選最大值, 並且把未排序區域的尾端資料和最大值交換. 就我初步的測試, 兩種實作方法的花費時間差不多. 不過後者是在 const 要求下, 也不須刪除 node, 比較不容易出錯.

理論上, 時間複雜度是 O(n^2), 因為做了 n 次長度為 n, n-1, …, 1 的遍歷.

c. **二元搜尋樹(binary search tree)**

add: 我構想的方法是, 先把資料丟入 find, 找到他應該插入的位置, 接著再使用 new 呼叫一個新的 node, 接在這個位置上, 並且設定這筆資料的 right, left 指標都是 0.

因為這樣的需求, 所以我的 find 函數, 除了回傳 bool type, 還要再吃一個 iterator 變數, 把最後 find 函數中, 停留的位置存進 iterator 變數.

理論上, 呼叫一個新的 node 和設定指標都是 O(1)時間, 所以時間複雜度取決於 find, 也就是樹的高度, O(logn), 在通常樹不會長的太偏的前提下; 但最差也可能到 O(n).

delete: 同樣分成 pop_front, pop_back, erase(iterator), erase(const &T)這四種功能. 關於 pop_front, pop_back, 指定刪除的對象基本上就是 iterator 的 begin 和 end 前一格, 也就分別是最小值和最大值. 顯然地, 最小值和最大值都是 tree 的 edge node, 所以只要直接刪除就好, 不需要補上. 同樣地, 可以把 erase(const &T)利用 find 簡化成 erase(iterator). 至於 erase(iterator), 就要考慮三個情況. 如果 left=right=0(#child=0), 也就是 edge node, 直接刪除就好. 如果只有一個 child, 那麼就把該 node 刪除, 並且讓他唯一的 child 補上就好. 如果有兩個 child, 那麼從大於該 node 的右側, 所有大於他的對象中, 用 find 找出最小值, 再用這個最小值覆蓋想要刪除的目標(iterator)就好.

值得一提的是, find 功能的使用方法. 如果要找最小值, 就用 find(a); 如果要找最大值, 就用 find(zzzzz); 如果要找一個 node(content T)的右側最小值, 就用 find(T+1).

理論上, pop_front, pop_back, erae(iterator)都需要用到 find, 來尋找刪除的對象或是用來覆蓋的 node, 但是其餘的動作都是 O(1)時間, 所以時間複雜度取決於 find(), 大部分是 O(logn), 極端情況下是 O(n). 而 erase(const &T)則是用 find 來簡化成 erase(iterator)情況, 也就是兩次 find, 所以時間複雜度同 erase(iterator).

find: 基本上就是二元搜索(binary search). 從頂點(root)開始, 如果給定的資料比 node 小, 則往左走, 如果比 node 大就往右走, 直到到達 edge node 為止.

因此, 理論上時間複雜度取決於樹的高度. 通常大多為 O(logn), 極端情況下可能到 O(n).

sort: 因為二元搜索樹(binary search tree)本身結構的緣故, 特別是他 add 和 delete 的結構維護, 所以它在任何情況下都是已經排序好的.

所以, 時間複雜度是 O(1), 基本上甚麼事情都不用做.

**2. 實驗數據**

**a. 單項功能的效能和解釋**

首先, 分別對各個功能, 測試他們的時間複雜度.

| | Array[2.5*10^6] | Array[5*10^6] | Array[10^7] |
|---|---|---|---|
| Add –r 1000000 | 0.08 s | 0.08 s | 0.08 s |
| Delete –f 1000000 | 0.02 s | 0.02 s | 0.01 s |
| Delete –r 1000000 | 0.14 s | 0.15 s | 0.16 s |
| Quary (amy) | 0.04 s | 0.09 s | 0.17 s |
| sort | 1.85 s | 3.75 s | 7.77 s |

解釋: 在 array 部分, 顯然 add, delete –f, delete –r 這三個運算都是常數時間, O(1) time, 因為他們並沒有隨著規模成長而遞增, 頂多因為電腦硬體和作業系統的不規律性, 而有些許的時間差異而已, 但基本上都很小. 而且, 這三個常數十間的運算, 都非常地快, 要連續做一百萬次, 才能夠有>0.01s 的非零數值顯示出來.

再來, Quary 運算需要遍歷一次 array, 並且一一比較. 所以如同理論預測地, 隨著規模增加成兩倍, 運算時間大概增加成 189%~225%, 也就是兩倍左右. 些許的誤差同樣來自於硬體和作業系統的不規律, 以及四捨五入的影響.

最後, sort 運算是 c++內建的 quick sort, 理論上是 O(nlogn), 也就是比 O(n)再增長快一點. 在實驗裡, 每當規模增加成兩倍, 運算時間大概增加成 203%~207%, 姑且看不出來是 O(n)還是 O(nlogn), 但感覺應該是略大於 O(n). 事實上, 理論上的預測: (2n)log(2n)/(nlogn)=2*log(2n)/log(n)=2+log4/logn. 以 n=2.5*10^6 帶入, 應該會增長成 209%左右, 如果是四倍, 那應該增長成 414%左右.

換言之:

| | 兩倍(基底 2.5*10^6) | 四倍(基底 2.5*10^6) |
|---|---|---|
| 理論增長(nlogn) | 209% | 414% |
| 實際增長 | 203% | 420% |
| 常數增長 | 200% | 400% |

所以, 實驗數據顯示出, sort 運算是 O(nlogn)的現象, 吻合我們的理論.

| | Dlist[2.5*10^6] | Dlist[5*10^6] | Dlist[10^7] |
|---|---|---|---|
| Add –r 1000000 | 0.11 s | 0.04 s | 0.09 s |
| Delete –f 1000000 | 0.03 s | 0.01 s | <0.01 s |
| Delete –r 1 | 0.03 s | 0.06 s | 0.16 s |
| Quary (amy) | 0.03 s | 0.06 s | 0.14 s |
| Sort “in 1% size” | 3.88 s | 15.10 s | 61.49 s |

解釋:在 dlist 部分，顯然 add –r 和 delete –f 都是常數時間，幾乎不隨著時間而增長，只是因為時間數值實在太小，所以不規則誤差的影響偏大. 而且同樣地，常數時間的單次操作都很快，需要連續做一百萬次，才能夠有一點點的時間值.

另一方面，在這裡，理論上 delete 應該是 O(n)時間. 因為在隨機數 size_t x 決定後，必須要花費 O(n)的時間，才能把 iterator 移動到第 x 格，也就是非隨機存取的特性. 而後，才能把 iterator 傳給 erase(iterator)來進行 O(1)的操作. 根據實驗的數據，當規模增加成兩倍，運算時間分別增加成 200%和 233%，考慮到可能有務規則的誤差，影響到 0.01 的數值，確實吻合理論沒錯.

最後則是 sort 的運算. 由於 sort 實在跑太久了，所以我只好把規模縮小成 1%，也就是 2.5*10^4, 5*10^4, 10^5 這三個規模底下運算. 理論上，由於我是採取 selection sort，應該是 O(n^2)時間複雜度，而且常數還不會太小. 實驗數據上，隨著規模增長成兩倍，運算時間分別增加成 389%和 407%，確實非常接近理論上的四倍沒錯. 所以這部分的理論預測也是成功的.

| | BST[2.5*10^6] | BST[5*10^6] | BST[10^7] |
|---|---|---|---|
| Add –r 1000000 | 2.04 s | 2.07 s | 2.14 s |
| Delete –f 1000000 | 0.17 s | 0.21 s | 0 s |
| Delete –r 1 | 0.25 s | 0.63 s | 1.2 s |
| Quary (ijk) x 30 次 | 0.27 s | 0.31 s | 0 s |
| sort | 0 s | 0 s | 0 s |

解釋: 首先, add –r 應該要是一個 O(logn)的操作，因為他需要先 find()，才能進行 O(1)時間的插入，所以時間取決於 find()，應該是 O(logn). 然而，在這裡我連續做一百萬次，所以理論上的預測，假設規模從 2.5M 增長到 10M，時間應該是 107%，與實驗結果的 105%非常接近.

再來, delete –f 應該要是一個常數的運算. 要不他是 edge node 就直接刪除，要不他只有右小孩就讓小孩補上. 而實驗中，確實時間長度也幾乎是固定的，而且每次運算都非常快. 然而，不知為何地，在 10^7 規模底下，刪除所需的時間竟然是 0，或許是程式有一些問題吧.

然後, delete –r 理論上是一個 O(n)的操作，因為在隨機生成 size_t 後要轉換成 iterator 需要 O(n)時間，然後再丟給 erase(iterator)做 O(logn)的運算．而兩次規模倍增以後，時間分別增加成 252%和 190%，平均大約是 220%，接近我們的預測.

最後, Quary 的部分，由於單次運算太快，因此我重複做 30 次．理論上，這最多需要走樹高度個回合，才能結束，所以也是 O(logn)．預測的增長幅度，應該是 log(2n)/logn=1+log2/logn，當 n=2.5M 時，也就是 104.7%，與實驗數據上的 115%也有一定的準確度．然而，比較特別的觀察點再於，可能大部分的運算都會在到達 O(logn)時間以前結束，所以平均下來並未到達 115%的增長幅度．而且，在規模是一千萬的條件上，不知為何做 Quary 反而只需要常數時間．可能是結構有一些特殊的性質，或者是隨機數在太多次以後有一些特性，亦或者是程式內有進行一些優化吧.

**b. 思考，比較，以及反思**

| | array | dlist | BST |
|---|---|---|---|
| Add –r | O(1) | O(1) | O(logn) |
| Delete –f | O(1) | O(1) | O(1) |
| Delete –r | O(n) | O(n) | O(n) |
| Query | O(n) | O(n) | O(logn)<br>maybe not tight |
| Sort | O(nlogn) | O(n^2) | 0 |

從這個表格的比較，可以推論出，如果使用者需要做到 sort 運算，甚至是幾次，幾十次的 sort，那麼 array 和 dlist 的複雜度一定是 O(nlogn)以上，基本上 BST 幾乎可以確定是最佳的選擇了.

- ➔ 需要使用到 Sort: BST > array > dlist

那如果不需要使用 sort 呢？基本上就只剩下 add, delete 和 query 的功能，整個系統的功能會減弱很多，一般是很少發生的．不過在這個前提下，基本上 delete –r 在三個架構下都是 O(n)．然而 BST 其餘的功能都低於 O(n)，因此顯然還是 BST 最佳．而 array 和 dlist 雖然複雜度一樣，但是 array 在實作上顯然速度快了一截，我想應該是常數的差異，所以 array 還是略加.

- ➔ 不需要使用 Sort: BST > array > dlist
- ➔ 整體系統速度: BST > array > dlist

然而, dlist 真的是最差的選擇嗎？他真的沒有太多的優點嗎？或是他有什麼優勢呢？首先，如果考慮到設計的難易度和複雜度，我想很顯然地，程式碼的數量也多少的反映出，他們的關係:

➔ 設計複雜度: BST > array > dlist

再來，也是反思的主要部分，是整個系統的極限容量問題. 換句話說，整個系統最大能塞進多少的元素呢？ 這個疑問給出的另外一個小實驗: 我們單純做 add –r 操作，每次都加入一百萬的元素，看看我們的容器最多能夠塞入幾個元素:

| | capacity | Used memory |
|---|---|---|
| Array | 16M < capacity < 17M | 655M |
| Dlist | 19M < capacity < 20M | 654.4M |
| BST | 13M < capacity < 14M | 664.3M |

➔ 整體而言，三個容器的容量關係為: BST<array<dlist

比較特別的是，因為 BST 的 add 運算是 O(logn)，會隨著尺度增加而成長，因此在一千三百萬尺度的情況底下，需要非常長的時間才能夠長到一千四百萬，而且記憶體也接近極限了，所以我就只好先中止程式，他的容器最大容量取決於 add 的運算時間. 至於其他的兩個容器，由於 add()都是常數時間，所以很快就做到極限的記憶體量了，因為記憶體不夠而停止.

所以，這個實驗反映出，整體速度最快的 bst，反而受限於 add 的複雜度以及記憶體，只能裝最少的容量. 而實驗上，整體速度最滿的 dlist，不但 add 的速度非常快，而且記憶體使用量也很少，可以裝最多的容器.

這樣的結果，也讓我們反思，並沒有最好，最優越的容器，只有最適合你的需求的容器.