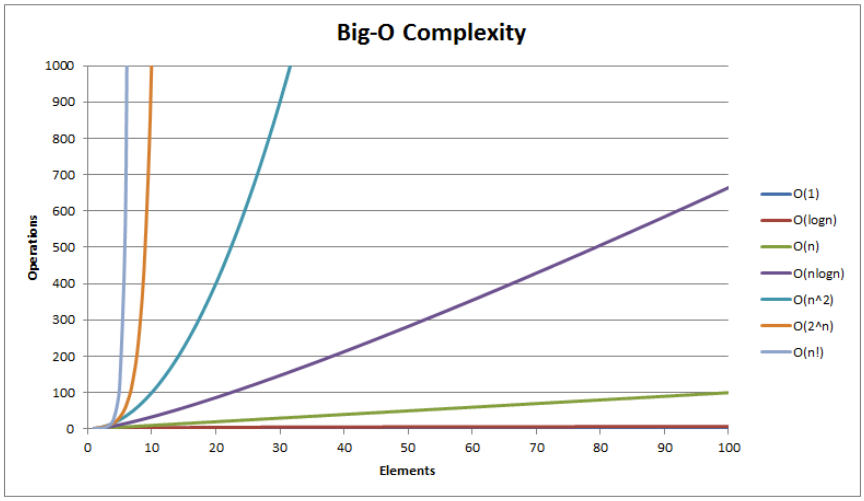
DSnP hw5 r07921001 李尚倫

1. Various abstract data types (ADTs) theoretical comparison:

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

如上圖，比較 Dynamic Array, Double-Linked List 和 Binary Search Tree 後可以發現，並沒有一個 ADT 是完美的，它們各有所長，應視使用情況來調整使用的 ADT 類型，因為 O(1), O(lgn), O(n)計算所需的時間在資料量大時會差距很大(如下圖)。而在接下來幾點我會就這三個資料結構做 Indexing, searching, insertion, deletion 的實驗，比較他們的 time complexity 和 space complexity。

Big-O Complexity Chart

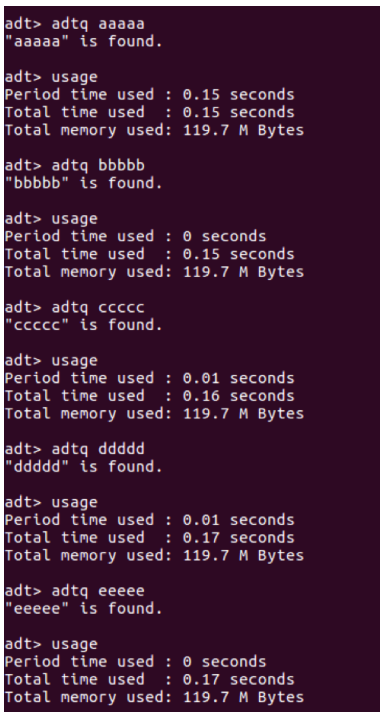


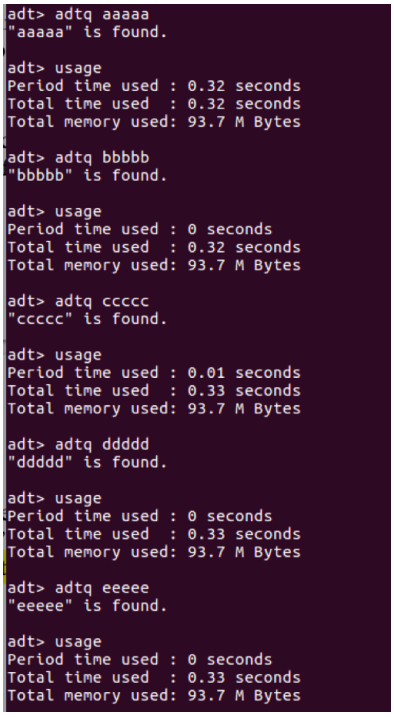
2. Searching

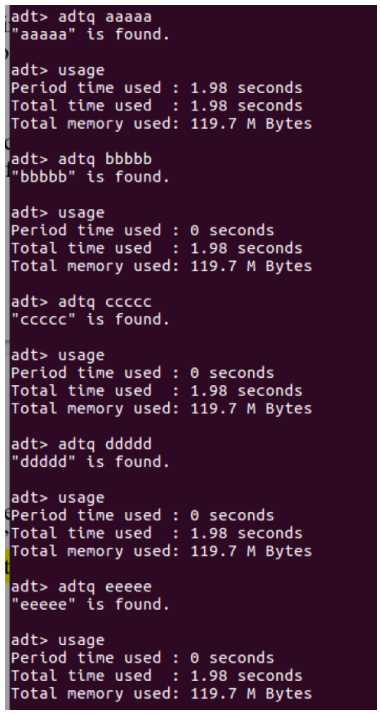
實驗結果依序為 dlist, array, bst，實驗的資料庫為 500004 筆資料，並穿插已知資料在其中，由下圖實驗結果可以得知，dlist 和 array 的計算複雜度理論上是 O(n)，但實際跑起來的速度其實很快，因為在 traversal 的時候他們皆只需要 O(1)，而且在平均的情況下，目標並不會都 traversal 到最後一個才被找到。反觀 bst 雖然理論上平均是 O(lgn)，但因為實作的不是 balanced tree，所以最差有

```
adta -s aaaaa
adta -r 500000
adta -s fffff
adta -r 500000
adta -s mmmmm
adta -r 500000
adta -s rrrrr
adta -r 500000
adta -s zzzzz
```

可能是 O(n)，而在 traversal 時，所花的計算時間也會跟 tree 的深度有關，並不比 dlist 和 array 直接 access 來得快，因此實驗結果中它比其他兩個花了更多的時間。另外空間複雜度方面，三者皆不需用到額外的記憶體，為 O(1)。

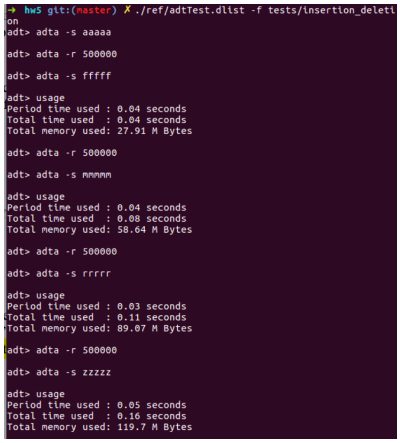


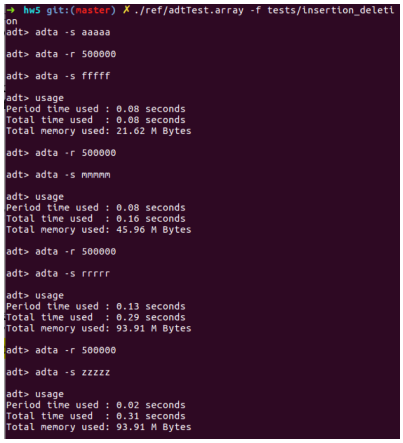


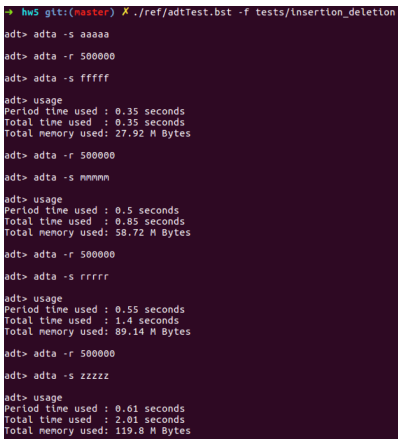


3. Insertion (push_back)

實驗結果依序為 dlist, array, bst，實驗的資料庫為加 500000 筆資料後額外 insert 一筆，由下圖為實驗結果。dlist 的計算複雜度理論上是 O(1)，每 insert 一筆都只需處理 pointer 的關係，把前後的 node 重新串起來即可，實際跑起來的速度是三者中最快的。而 array 的計算複雜度理論上是 O(n)，但因為我們的實作方式只是把資料 push_back 在最後，因此平常只需要 O(1)即可達到，只有在 capacity 裝滿時要把 capacity 變大，會需要 new 一個新的空間並把舊的資料複製過去，計算複雜度才會到 O(n)，速度僅次於 dlist。最後 bst，每次 insert 時都需要維持資料庫的順序，要先找到可以 insert 的位置等步驟最差會需要花到 O(n) (如果不是 balanced tree 的話)，最好也要 O(lgn)，insert 時也需要花 O(1)調整 pointer 之間的關係來維持 bst 的架構，因此會是三者中最慢的，但始終維持順序這點會有利於他之後的使用。



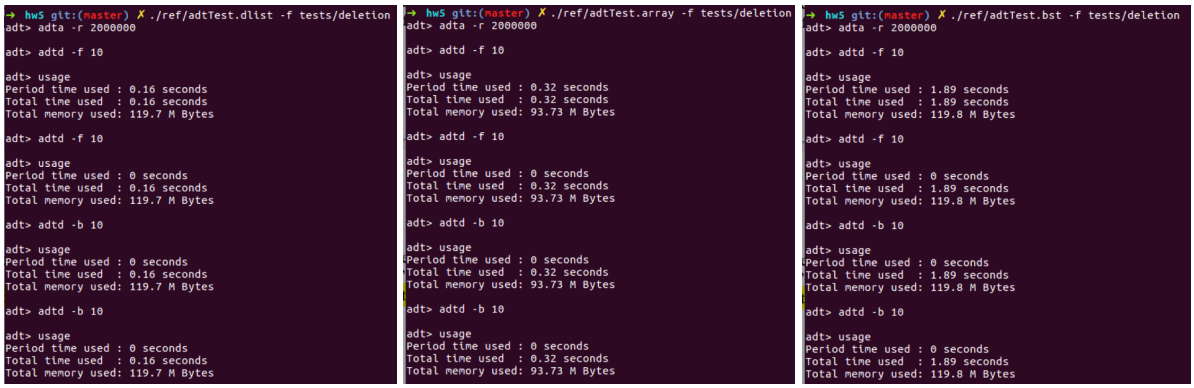




另外空間複雜度方面，三者皆不需用到額外的記憶體，為 O(1)。

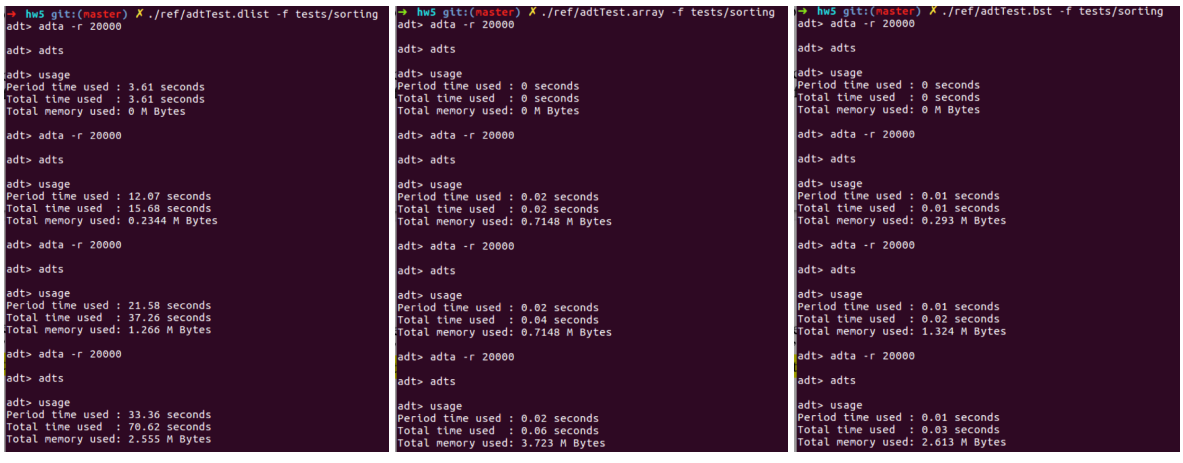
4. Deletion (pop_front, pop_back)

實驗結果依序為 dlist, array, bst，實驗的資料庫為加 2000000 筆資料，由下圖為實驗結果。dlist 的計算複雜度理論上是 O(1)，每 delete 一筆都只需處理 pointer 的關係，把前後的 node 重新串起來即可，實際跑起來的速度是三者中最快的。而 array 的計算複雜度理論上是 O(n)，但因為我們的實作方式，如果是 pop_back 就只是把 size -1，pop_front 則是把頭尾的資料 swap 後再 size -1，並且不變動(縮小) capacity，所以只需要 O(1)即可達到，速度和 dlist 相當。最後 bst，每次 delete 時都需要維持資料庫的順序，因此先找到頭或尾要進行 pop_front, pop_back 時，可能就已經會需要花到 O(n)(如果不是 balanced tree 的話)，最好也要 O(lgn)，然後再進行 deletion，維持好 bst 的架構，因此會是三者中最慢的，但始終維持順序這點會有利於他之後的使用。另外空間複雜度方面，三者皆不需用到額外的記憶體，為 O(1)。



4. Sorting

實驗結果依序為 dlist, array, bst，實驗的資料庫為每增加 20000 筆資料 sort 一次，總共做了四次，由下圖為實驗結果。dlist 因為 linear access 的特性，使其無法套用大部分計算複雜度 nlgn 的 sorter，計算複雜度會比 O(nlgn)來得大，為三者中花最久時間的。而 array 可以 random access，所以可以套用 c++內建的::sort，計算複雜度理論上是 O(nlgn)，速度與 dlist 相差甚大。最後 bst，因為每次對資料庫進行加減的時候都會維持架構，隨時都是處於 sorted 的狀態，不需要額外花時間進行 sort。



5. Indexing

因為本次報告為使用 ref 的程式進行，因此沒有特別再去實作一個 indexing 的 command，直接使用 deletion 裡面的 random delete，因為他的實作方式是先拿到一個 random 的數當 index 後，去刪掉那個 index 位置的資料，所以計算複雜度會同時包含 indexing 和特定位置的 deletion (非 pop_front or pop_back)。實驗結果依序為 dlist, array, bst，實驗的資料庫為 200000 筆資料，總共做了四次 random deletion，由下圖為實驗結果。其中 array 和 dlist 的 deletion 前面有分析過，而這裡兩種資料結構的 deletion 也是同理，實作上僅需 O(1)，indexing 上 array 理論上僅需 O(1)，可以 random access，而 dlist 理論上需 O(n)，因為是 linear access，要拿到想要的 index 必須一個一個往前找，不過實驗起來的速度兩者其實差不多，只有 bst 因為要找到想要的 index 必須要 traversal tree，可能需要花到 O(n) (如果不是 balanced tree 的話)，最好也要 O(lgn)，因此是三者中最慢的。