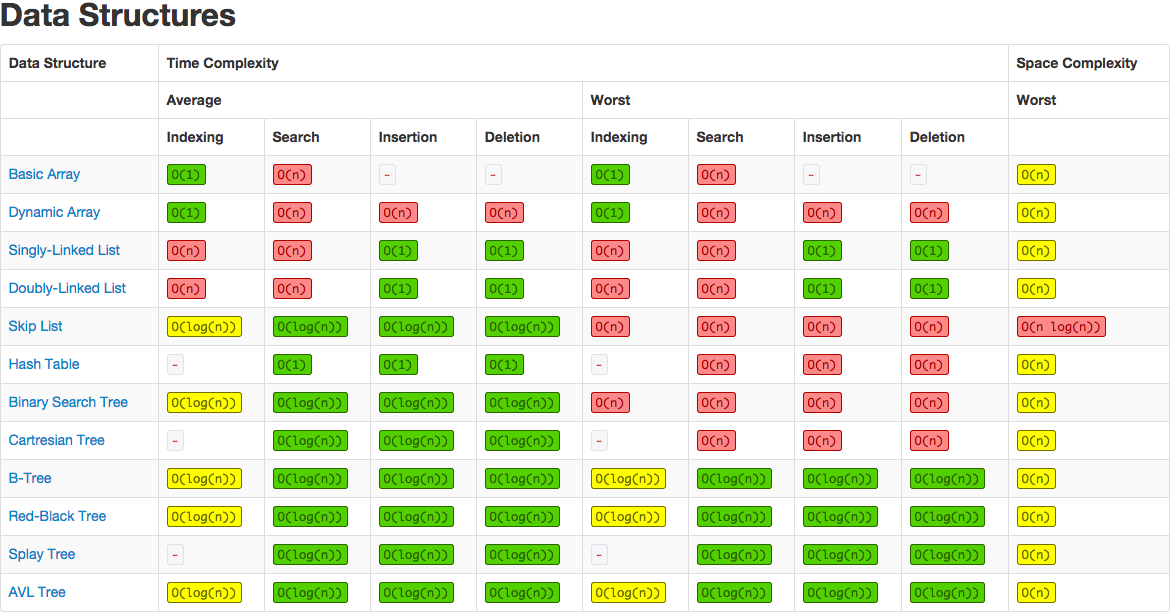
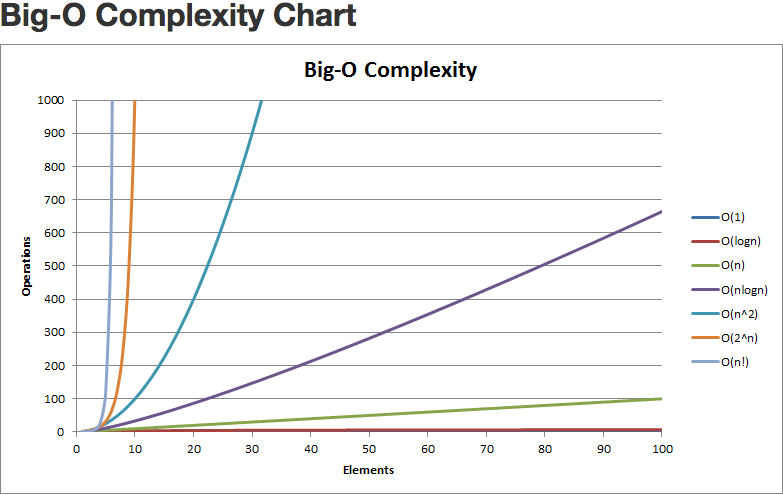
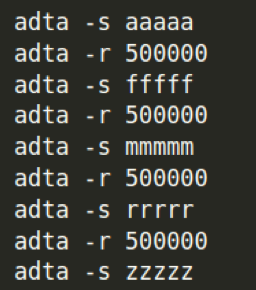
**DSnP hw5 r07921001 李尚倫**

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



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。



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

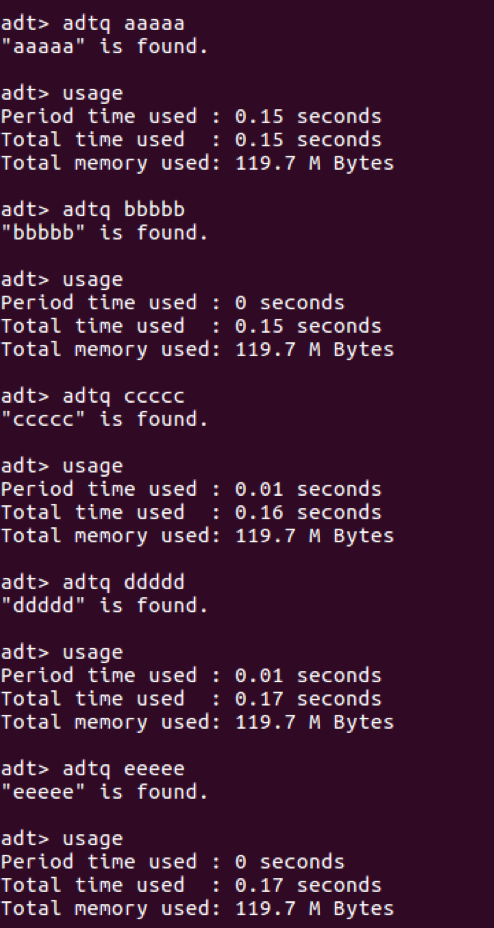
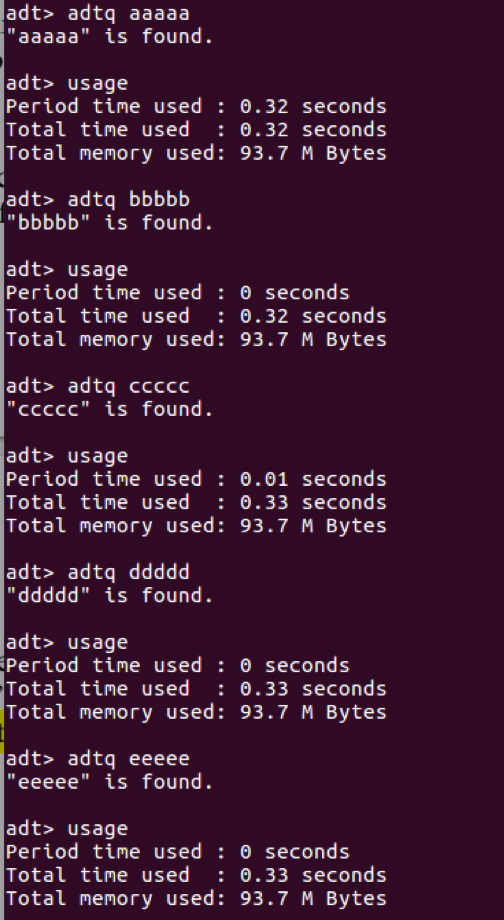
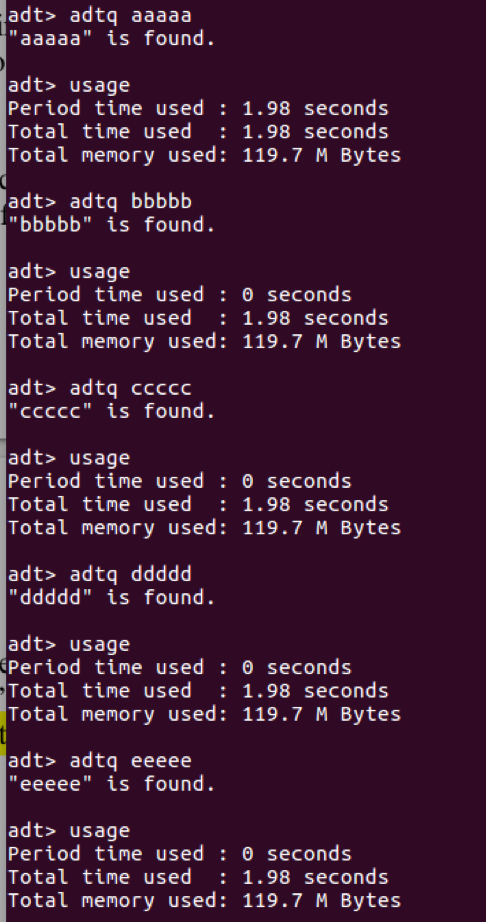
的計算複雜度理論上是O(n)，但實際跑起來的速度其實很快，因

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

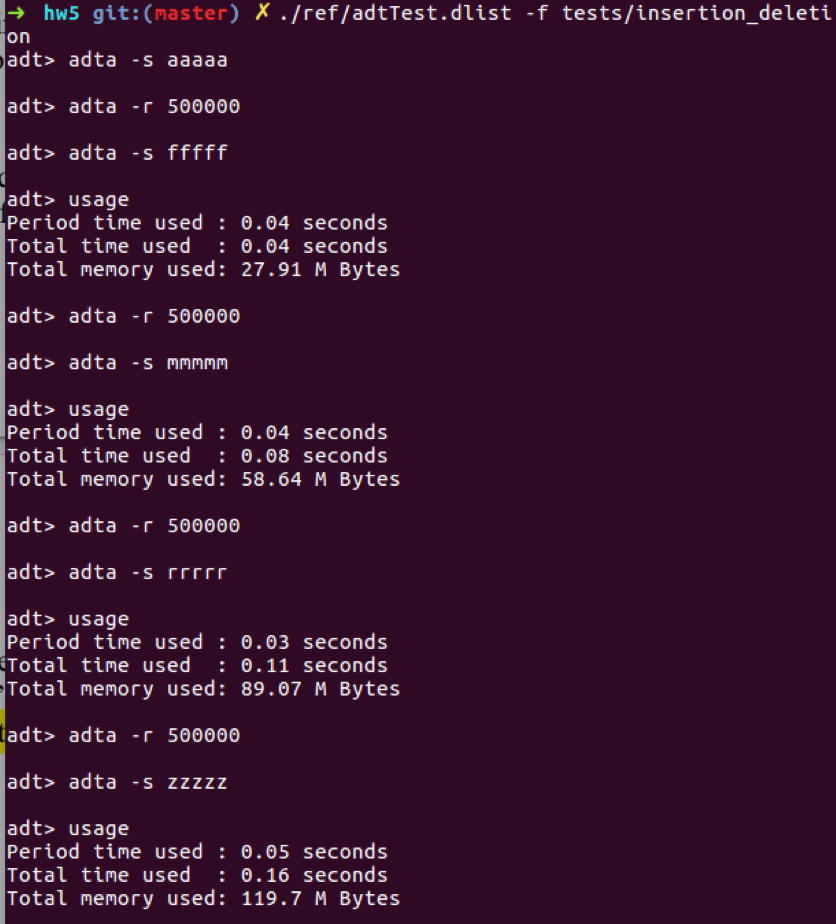
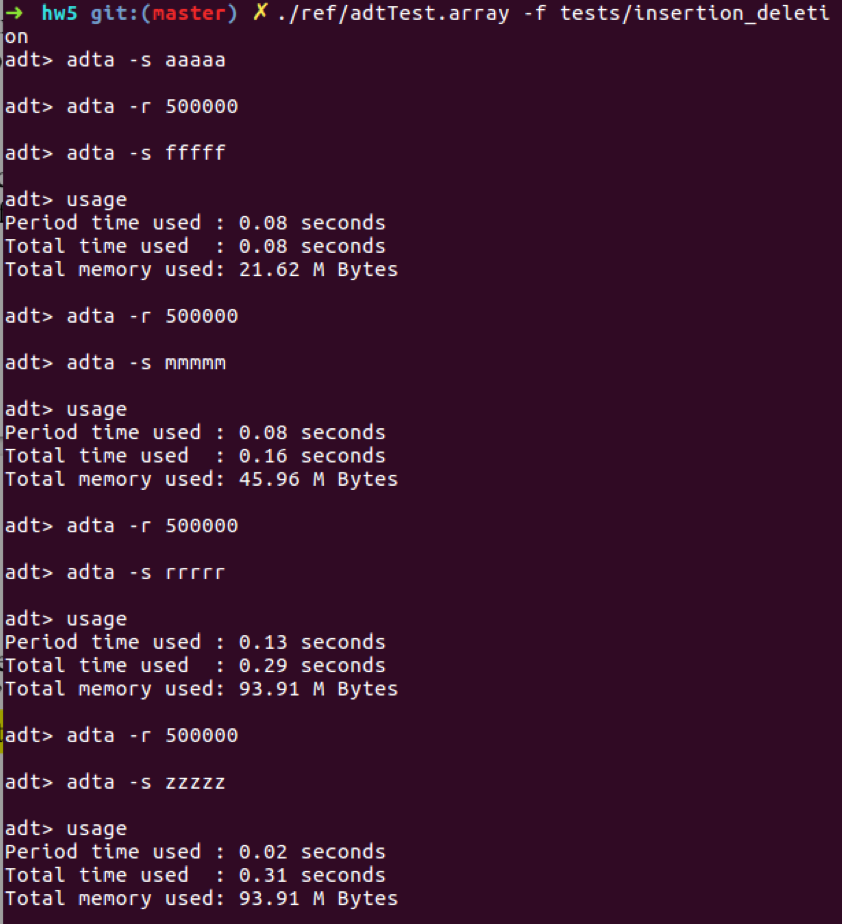
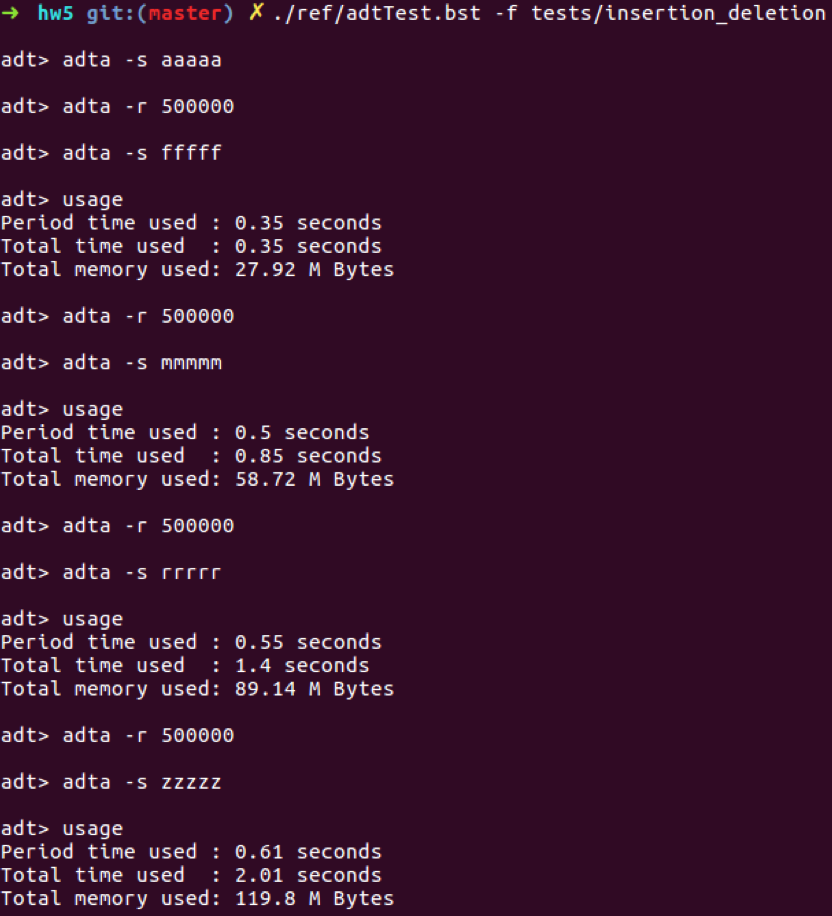
上平均是O(lgn)，但因為實作的不是balanced tree，所以最差有

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)

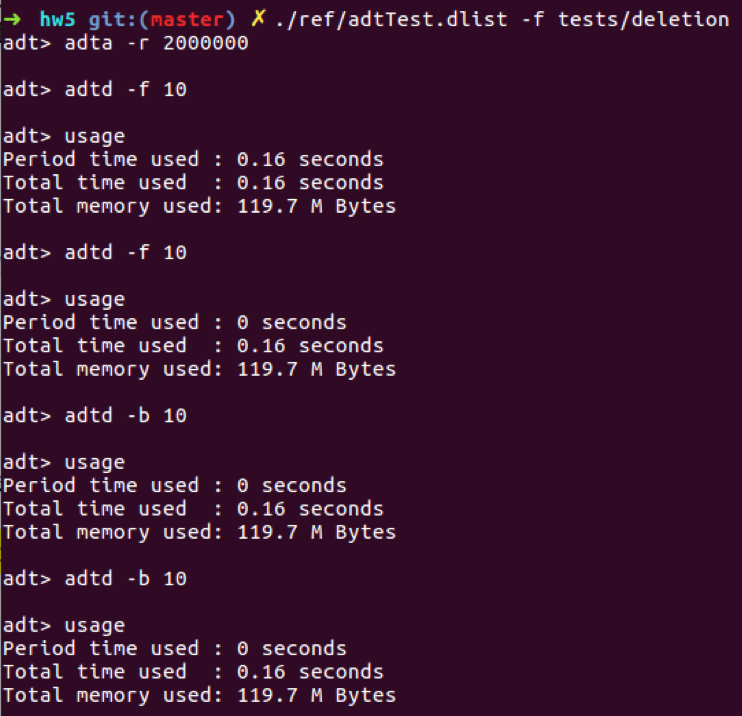
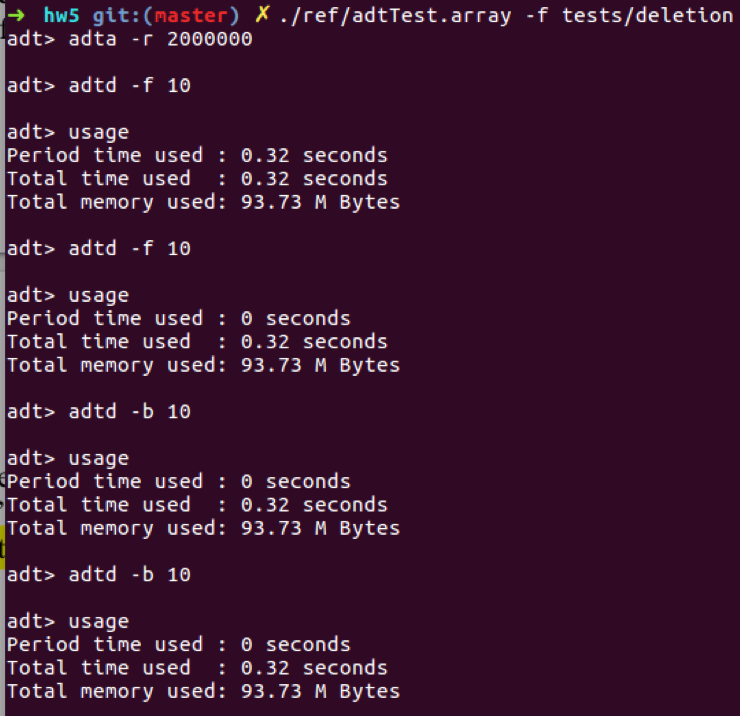
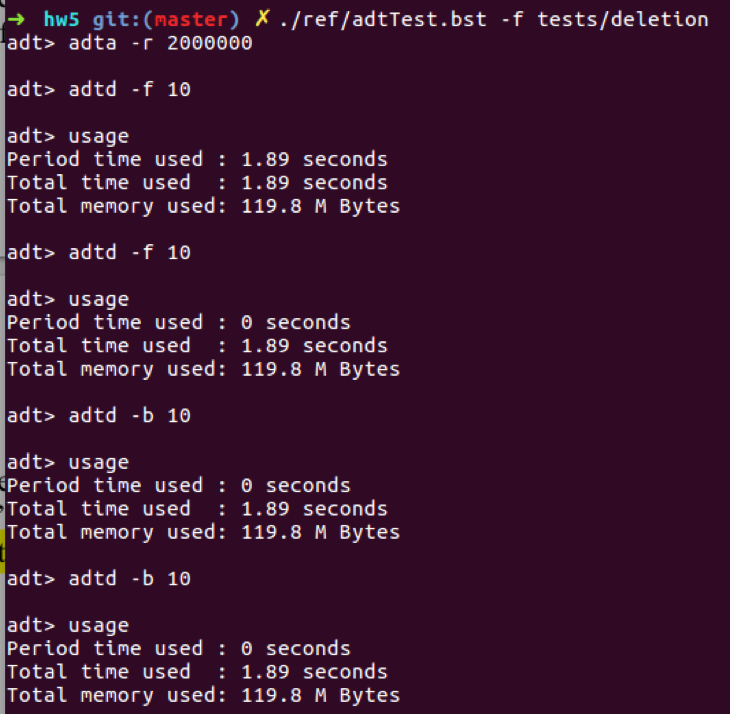
實驗結果依序為dlist, array, bst，實驗的資料庫為加500000筆資料後額外insert一筆，由下圖為實驗結果。dlist的計算複雜度理論上是O(1)，每insert一筆都只需處理pointer的關係，把前後的node重新串起來即可，實際跑起來的速度是三者中最快的。而array的計算複雜度理論上是O(n)，但因為我們的實作方式只是把資料push\_back在最後，因此平常只需要O(1)即可達到，只有在capacity 裝滿時要把capacity變大，會需要new一個新的空間並把舊的資料複製過去，計算複雜度才會到O(n)，速度僅次於dlist。最後bst，每次insert時都需要維持資料庫的順序，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)。

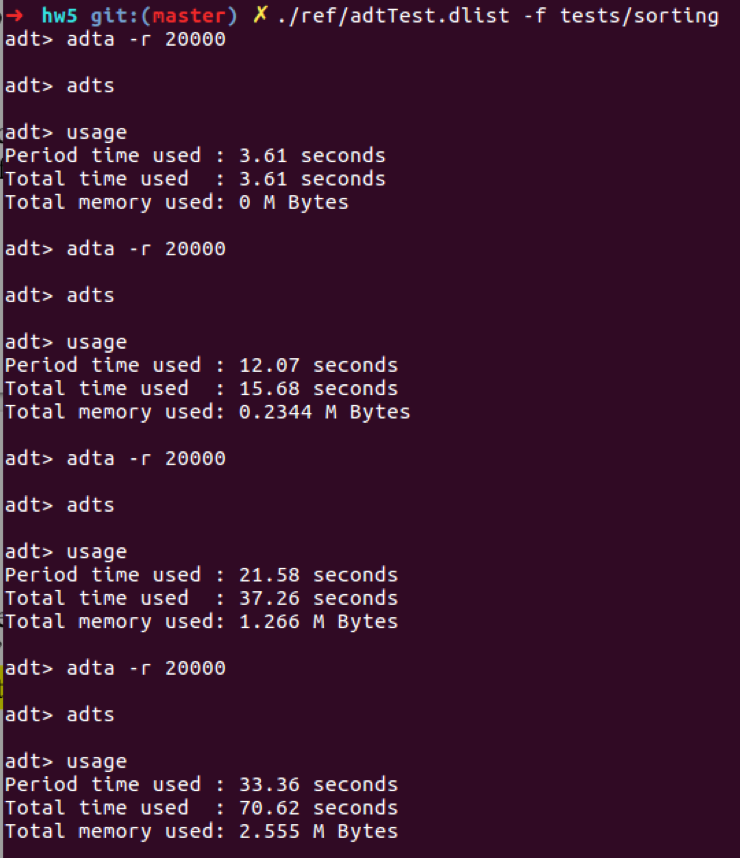
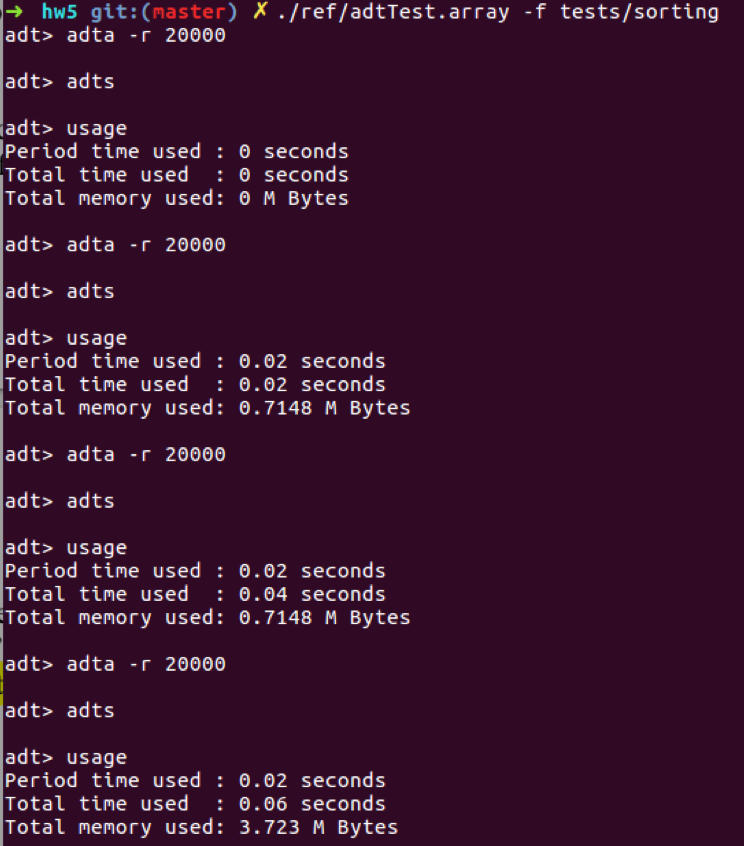
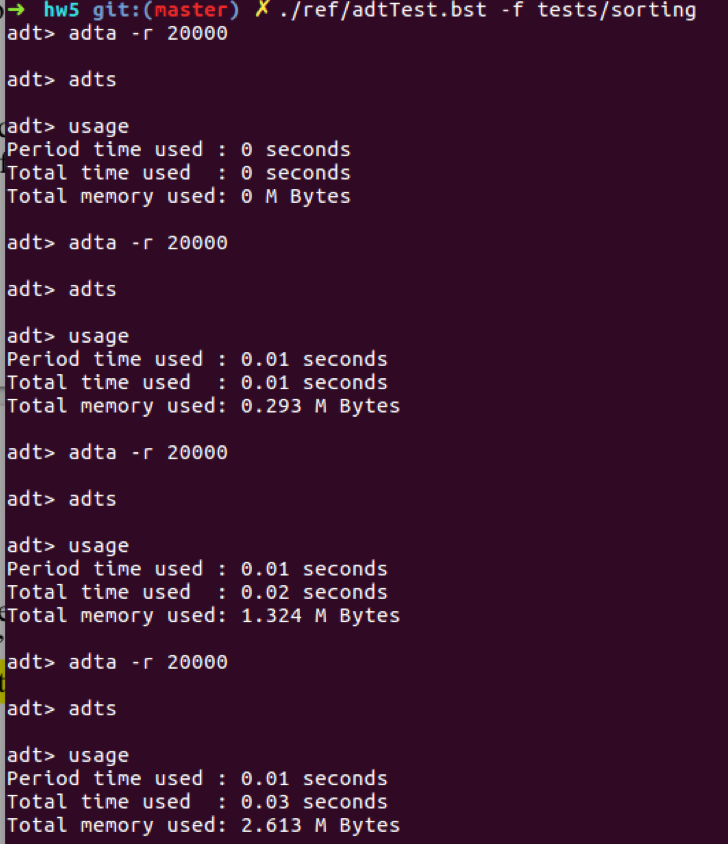
4. Deletion (pop\_front, pop\_back)

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。

