**HW5 adtComp**

生機所碩二 r05631044黃君灝

1. **資料結構實作**

這三種資料結構都有Node類別，在用一個大的ADT類別包起來，其中Node裡面含有資料訊息以及根據不同的資料結構而有不同的指標。

**Doubly linked list :**

Doubly linked list含有兩個指標，分別指向下一個節點以及上一個節點，iterator重載的部分++/--就分別將\_node指向下或上一個節點，當位置在頭跟尾時，--/++便會連到一個dummy node，形成一個loop。

Begin()的部分就直接指向第一個節點，end()及用第一個節點的iterator再--，找到dummy node的地方。

判斷是否為空的ADT就看是否為dummy node，是的話他的\_next、\_prev變分別會指向自己。

Size就從begin()開始數到end()前一個節點。

Push\_back()我分為兩種狀況，當資料為空的時候，產生一個新的節點並將其指標與dummy node互相鏈結，而非空的時候則是用--(end())找到最後一個節點，並將其\_next指標new一個新的節點，再將dummy node的\_prev指向新的節點。

Pop\_front() & pop\_back()就分別用\_head & --(end())找到第一個與最後一個節點，將其前後的節點指標連向彼此，再delete該位置節點。

Erase(iterator& pos)的部分一樣將pos前後節點的指標連向彼此，再將pos位置的節點刪除。

Find()是從頭開始數到尾，若有找到相應的節點則返回其位置，沒有的話返回dummy node。而erase(const T& x)我則是用find以及erase(iterator& pos)這兩個函式合併運用，找到的話就將它刪除。

Clear()判斷ADT是否為空，否的話則用pop\_back()開始慢慢刪除節點。

Sort()部分我是選用insertion sort，原因是它是stable sort，並且我認為其演算法是最好理解並好時做的，與bubble sort一樣都是O(n2)，因此選用它，而沒用更快的merge sort則是因為它需要更多的記憶體來實作。

**array :**

array的iterator有個指標\_node指向array的不同記憶體位置，在重載++/--的部分只需要將那個\_node做加減運算變可以得到對應的位置。

Begin() & end()也是用\_data，也就是整個array的頭的記憶體位置作運算，\_data開始dereference便是begin()，\_data += size便是end()。

Empty() & size()則是用整個類別裡面的\_size來判斷，在做資料處理的時候便會計算現在ADT裡面資料的量，而判斷是否為空的時候便是用size是否為0來判斷。

Push\_back()主要將資料丟置整個ADT末端位置，並將size+1，但要維護整個ADT的capacity，如果不夠時則將capacity變為兩倍以後再將資料複製過去新的array。

Pop\_front() & pop\_back()比較簡單，pop\_front()將第一比資料內容改成末端的資料再將size-1即可，pop\_back()只需直接將size-1，如此在讀取資料時只會讀到size這個範圍的內容。

Erase(iterator& pos)與上述方法一樣，將該位置資料內容改為尾端資料再將size-1即可達成目標。

Find()一樣是從頭開始讀取讀至尾，找到對應的返回其位置，而erase(const T& x)我一樣是用上面兩個函式合併，用find尋找在用前一個erase刪除。

Clear()只需將整個ADT size-1就好，而sort則是用stl裡面的函式。

**binary search tree :**

iterator的++/--部分還沒整個完成，因此在traversal時會有問題，所以無法跑出正確的結果，這部份還在努力。

Begin()我是從\_root開始走，因為在insert時是依據順序接上的，因此當有左分支時變往左走，便可以得到最小的那筆資料。

End()則是創造一個iterator，讓他的\_node為零，當資料在尾端，也就是最大的資料時++便會指向end()，而--end()則會指回最大那筆資料。

而empty()與size()跟array一樣，在做資料增減時變將資料大小做加減運算存起來，需要時直接return即可。

Insert()依據所要加入資料的內容大小做尋訪，資料較節點資料大時則往右走，小則反之，當沒有左右分支時便將該節點的pointer new一個新的節點接上，此節點內容便是所要加入的資料內容。

而erase(iterator& pos)是依據pos的資料內容大小作尋訪，找到其位置將\_left、\_right、\_parent做串接，又分幾種狀況，為external node時、只有左右分支時，以及同時有左右分支時所做的都有點不太一樣，找到對應的節點維護好這三個pointer即是。

Find()則是一樣從\_root作尋訪，找到的話返回指向該位置的iterator，沒有的話返回end()。

而pop\_front()、pop\_back()、erase(const T&)我都是用上面的erase來作運行，找到所要的位置的iterator再將其節點拿掉。

Clear()的部分與dlist一樣，當判斷ADT不為空時，pop\_back()慢慢將其刪除至沒有資料為止。

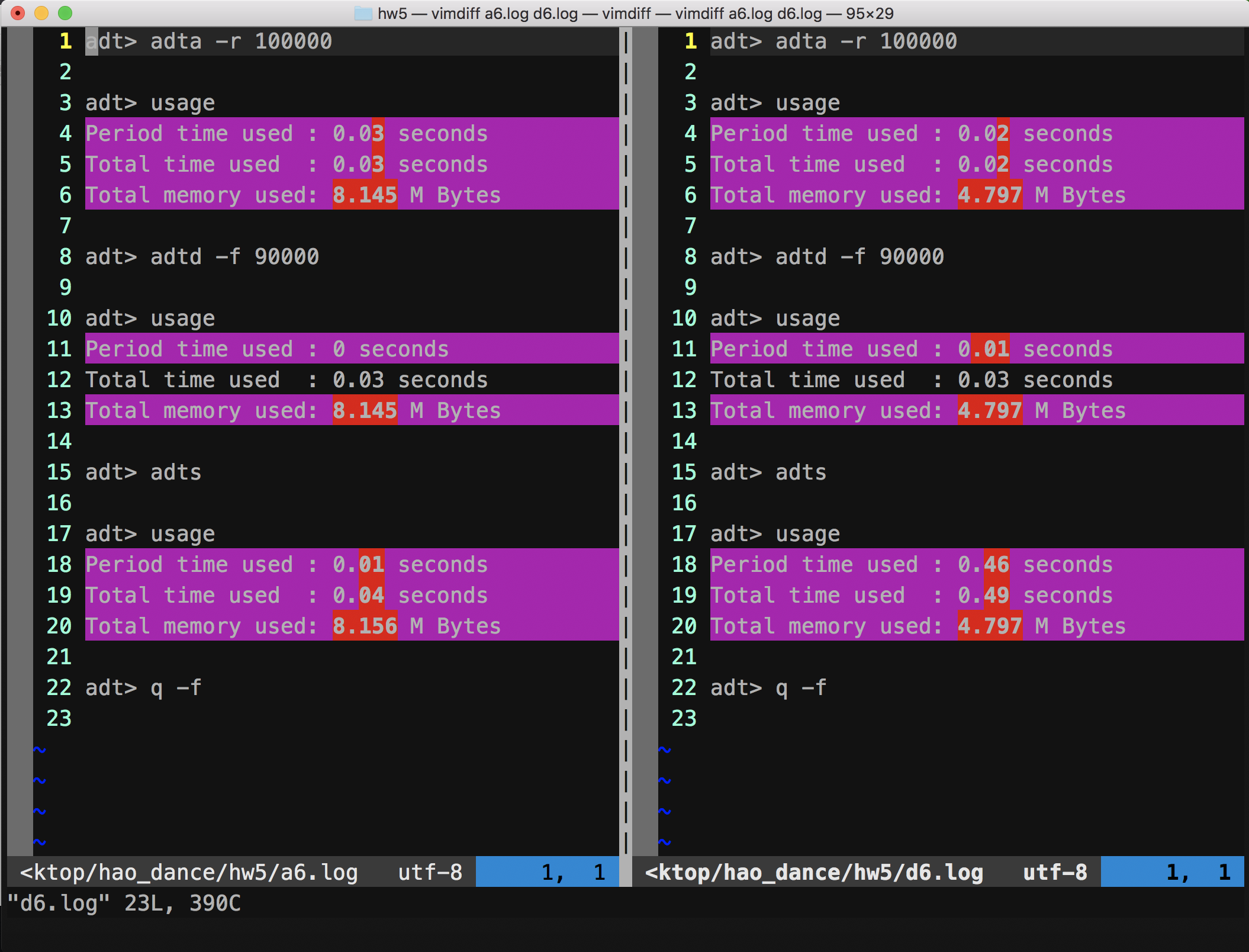
而sort()在bst不需要因為bst以是一個已經排序好的ADT。

1. **實驗結果與比較**

理論上來說，在增加大筆的資料時，array & dlist應該會相對來的快許多，因為不像bst需要維護整個ADT的排序，然而在做搜尋資料時，bst則會比上面兩者來的快，因為順序的關係可以做O(logn)的搜索。

而array在sorting時又會比dlist來的快，因為他的資料室連續的，不需要一直跟著指標只道不同區段的記憶體，因此在搬運資料時會相對來的有優勢，不過相對的在一開始就要限定好array的大小，不夠的時候就得開新的array再把資料複製過去，所以在新增資料時會相對慢一些。

寫了一個簡單的策資作測試，下圖分別為用array & dlist作運行的結果



左邊是array，右邊是dlist，在新增十萬筆資料時array相對慢了約0.01s，主要應該是capacity不夠另開新的array並搬運資料所花的時間，而在delete資料的時候變不會比較慢，只將size-1而已，反而dlist需要做pointer assigned和串接的動作，所以慢了一些些。在sorting的部分很明顯的array就比dlist快上許多，都與預期的差不多。

而因為bst最後沒在時限內完成因此沒有作結果討論。