DSnP 物理二 林展慶 b05202044

一、資料結構的實做

1.三種ADT的資料結構& 操作上的不同& function的performance。

	Array	BST	DList
資料結構	1.向系統直接要一整塊空間來存data。而data會是連續的放在一起。	1.每個node最多只能有兩個child與一個parent，並且左邊的child一定不會比自己大，右邊的child不會比自己小。 2.存data的方式是用BSTreeNode來將parent和child的位址以及data存起來。	1.使用DListNode來將前一個node和下一個node的地址以及data存起來。
操作的差別	1.簡單好寫直觀	1.若使用parent&child來暴力寫member function的話，會使整個code變得有點複雜，尤其是delete。	1.簡單好寫直觀 2.使用dummy node的概念，可以使紀錄更方便。
insert //push_back	O(1) //O(n) if expand.	O(log(n)) //in avg	O(1)
pop front/ back	O(1)	O(log(n)) //in avg	O(1)
sort	O(nlog(n))	O(1)	O(nlog(n))
find	O(n) //可以是O(log(n)) if is sorted.	O(log(n)) //in avg	O(n)
print	O(n)	O(nlog(n)) //since operator++ cost O(log(n))	O(n)
clear	O(1)	O(nlog(n)) //若長成一條直線則只須花O(n)	O(n)
評語 //優缺點	1.若排序不需要時常保持正確，效率	1.搜尋只需O(log(n)) 2.若需要常常知道順	1.insert/delete的速度很快。

	很高 2.可能會浪費些許記憶體 3.記憶體連續 -存取效率 -可隨機存取	序關係，可能不錯用 3.要確定整棵樹一直都是BST會需要花一些功夫。	2.delete不會影響sorted的狀態。

2.如何去實做？& 為什麼這麼做？

Array	1.向系統從動態記憶體一次要一大塊記憶體，若在insert過程中賽不下了，則再去向系統要一塊兩倍大的記憶體，並且將原先的記憶體還回去。 2.在push時，可直接將物件複製到最後面。 3.在pop時，可以直接將最後面的物件複製回來填補空缺，如此一來可以確保整個array都是連續的。 4.會這麼寫的原因就是因為很直觀，並且在也可以保持在最佳的時間複雜度。
BST	1.在BSTreeNode中存parent & leftchild & rightchild 的位址，這麼做就可以使用暴力法來進行find & insert & pop & print 2.為了在外面使用iterator時能夠更直觀，於private member中多紀錄一項BSTreeNode<T>* _end，以確保(end()--)++可以回來end() 3.原本會這麼做的原因是因為很直觀，並且可以保持在最佳的時間複雜度。但是寫到後面才發現需要紀錄一個_end，不然end()會寫不出來，結果寫到後面bug變得很多，de了很久，也許用其他方式來寫可以使code變得更乾淨。
DList	1.使用於DListNode中，使用_ next & _prev來紀錄前後Node的位址，如此一來就可以連接不同位址的記憶體了。 2.使用dummy node來將DList變成一個circle。 3.sorting 最直觀的方法就是 bubble sort，不過如果在資料很大的時候，也可以用quick sort之類的。 4.會這樣寫的原因是因為很直觀，並且除了sort其他function都可以保持在最佳的時間複雜度。而sort會寫bubble sort是因為老師在課堂上說只需要寫bubble sort就好。

二、實驗比較

實驗目的：
　　比較三種不同ADT在
1.insert
2.reset
3.insert(random)+sort
4.delete(random)+sort
上的performance。

實驗設計：
　　input不同的指令，比較不同情況下的usage為何：

1.insert大量資料時
input：adta -r 500000

2.reset
input：adtr 6

3.需要在insert時時常保持sorted狀態時
input：(adta -r 100 adts)重複10次

4.需要在delete時時常保持sorted狀態時
input：(adtd -r 100 adts)重複10次

實驗預期

	Array	BST	DList
insert大量資料時	因為在一開始建立array時，每insert 2的整數次方個data時，就需要進行一次copy，故需耗時O(nlog(n))	每個insert的資料都需要花費log(n)的時間，故總耗時為O(nlog(n))	每一個insert都耗時O(1)，故總耗時為O(n)
reset	只需要移動end的指標就好，故總耗時O(1)	每一個delete都需要花費O(log(n))的時間，但實際上所花費的時間會跟樹的形狀有很大的關聯性，但是在接近balace的情況下，平均花費應為O(nlog(n))	每一個delete都耗時O(1)，故總耗時為O(n)
insert+sort	主要會花費時間會在sort上，故總耗時為O(nlog(n))	sort不會花費時間，故總耗時應該為O(insert數量*log(n))	主要會花費在sort上，而在這次hw中使用的演算法為bubble sort

			，故總耗時為O(n^2)
delete+sort	同insert+sort，故總耗時為O(nlog(n))	同上，故總耗時為O(delete數量*log(n))	delete不會影響sorted的狀態，故總耗時為O(delete數量*n)

實驗結果：

	Array	BST	DList
insert大量資料時	Time: 2.54 secs memory: 383 MB	Time: 10.21 secs memory: 304 MB	Time: 0.66 secs memory: 304 MB
reset	Time: 0 secs	Time: 1.28 secs	Time: 0.07 secs
insert+sort	Time: 0.33 secs	Time: 0.01 secs	Time: 286 secs
delete+sort	Time: 0.41 secs	Time: 9.67secs	Time: 0.27 secs

討論：

1.在insert大量資料時效率為DList>Array>BST。
與預期結果一樣。可以看出，Array雖然跟BST的時間複雜度一樣，但是係數會小不少。但是Array會比其他兩個ADT多出了一些memory。並且在reset後Array並不會將空間還回去，這麼做空間會比其他兩個多至多一倍的使用，但是可使之後insert的速率變快（因為不需在這麼頻繁的跟系統要空間），因此考慮在空間成本不高的情況下，好用程度應該為DList>Array>BST
（NB:如果在一開始就知道需要使用的空間有多大，那麼Array的時間複雜度也可以變成O(n)）

2.在reset時效率為Array>BST>DList。
與預期結果一樣。而外可以討論的是，Array若在reset後知道不會再用到這多容量的話，可以多設計一個func來歸還記憶體。

3.時常需要insert並且要求要時常保持sorted狀態時的效率為BST>Array>DList。
與預期結果一樣。可以看出，若是需要進行大量data的排序時，十分不適合使用Bubble Sort的演算法。

4.時常需要delete並且要求要時常保持sorted狀態時的效率為DList>Array>BST。
與預期結果不一樣。這裡Array跟DList的performance跟預期的差不多，但是BST原本預期只會花費O(delete數量*log(n))但是卻花費了9.67秒，因此可以看出進行delete的時間複雜度雖然只有O(log(n))，但是係數卻非常的大。我猜測係數這麼大是因為delete後需要保持整棵樹持續為BST，而這個步驟會花費很多的時間。