資料結構與程式設計(Data Structure and Programming)

Homework #5 Report

資管三 **B06705027** 黃柏叡

一、資料結構實作

1. Array/ Dlist/ BST 之異同

Array

ARRAY為連續之記憶體，能以index快速存取資料。在實作中，array的CAPACITY

為2的次方數，當size = array後仍需加入資料時才做必要的CAPACITY擴展

優點：

Random access(只要利用index即可在O(1)時間對資料做存取)

節省記憶體空間，不需利用pointer記錄其他node的記憶體位置

缺點：

在三者ADT中新增及刪除資料的方式最麻煩，因為會需要挪移資料

調整capacity大小時，需要先用一temp指標站存資料，搬動上較麻煩

Dlist(doubly linked list)

以pointer串接不連續的記憶體。每個node除了有data，還分別有指向前一個與後一個的pointer；沒有Array中需要調整capacity的問題。

優點：

新增與刪除資料相較其他兩者較容易，透過更動node附近的pointer指向位址即可

因沒有capacity的限制，資料數量是動態的

缺點：

若要存取資料便只能從_head循序存取而不能隨機存取

因透過pointer維持其運行，因此需要額外的記憶體空間存pointer

BST(binary search tree)

Binary tree 中，每個node有兩個child，左child的data值比其parent小，右child的data值比其parent大。從_root開始向下存取資料，進行data大小的比較(n筆資料時最多n比較n次)，以判斷node的歸屬。

優點：

搜尋與插入資料的時間複雜度較低

資料在新增時即完成了排序，結構因比照樹狀而條理分明

缺點：

需要額外的記憶體空間存 pointer

實作上困難且和前兩者相比較不直觀

2. 程式說明

Array：

Data structure & member

1. 在class Array內有一個指向T的pointer _data，指到陣列第一個資料的位置。

   另外還有_size跟_capacity，_size紀錄資料目前的確切數量，而_capacity則是

   資料目前可以儲存的最大數量。

2. 當欲加入的資料數量超過_capacity時，再進行expand，從0開始，之後依序為

   2的次方數(1,2,4,8,...)

3. 在iterator中的++及--：

   因為是連續的記憶體空間，所以在++與--時只需要將pointer加減1即可。

Functions:

1. begin()，end()，empty()

三者分別記錄array的頭、尾、以及是否為空

2. push_back

先判斷需不需要expand capacity。若需要，則必須重新**new**一個大小為新capacity為原capacity2倍的array(empty則為1)；將原有資料存在暫存指標，之後原指標指向新位址之後搬過去，刪除暫存指標，最後size++。

3. pop_front

判斷是否為空，之後將最前面的資料pop出去；要注意若資料數大於二，則將最後面資料拿來補到最前面，最後size--。

4. pop_back

拿掉最後面資料，size--；若empty則不做任何事。

5. erase

將最後一個資料移到欲刪除資料的位置，size--。

6. find

從頭開始依次向下一個搜尋相符的資料。

7. sort

使用內建的sort function將陣列data進行排序。

8. clear

刪除所有資料，將size設為0。

Dlist (doubly linked list)

Data structure & member

1. Dlist最大的特色便是每個記憶體擁有指向前一個與下一個資料的pointer
2. 在class DList內有指向第一個node的pointer _head，也因此linked list**只能從頭開始循序存取**資料。
3. 除了每個資料的空間外有一個dummy node，是_head的前一個(_prev)。在沒有資料(empty)的時候，dummy node就是head；在empty的狀況時，dummy node的prev跟next都指向自己；一個node時，兩者互指；一個以上時則循環
4. 在iterator中的++及--：

   ++為next，指到現有node的下一個，--即為prev，指到現有node的上一個

Functions:

1. begin()，end()，empty()

三者分別記錄dlist的頭、dlist的prev、以及dlist是否為空(只有dummy)

2. push_back

首先new出一個node，再來判斷dlist是否為empty，若empty則加入node並令其為head，兩者next、prev互指；若非empty則加到最後，並改變相鄰node的next prev

3. pop_front

判斷是否為空，之後將最前面的資料pop出去；若剩一個被pop，則dummy的next prev皆指自己，且dummy變成新的head。

4. pop_back

判斷是否為空，之後將最後面的資料pop出去；若剩一個被pop，則dummy的next prev皆指自己，且dummy變成新的head。

5. erase

刪除相對位置或名稱的資料。

6. find

找尋相對位置或名稱的資料。

7. sort

在這邊我使用最直觀的bubble sort($O(n^2)$)將list data進行排序。

8. clear

刪除所有資料，dummy的next和prev指向自己並head = dummy。

BST (binary search tree) (程式部分未完成)

Data structure & member

1. TREE的最大的特色便是每個記憶體擁有一個指向上方的parent pointer以及兩個指向下方的child pointer(left right)
2. 在class BSTree內有三個pointer，**_root/ _min/ _dummy**。_root就是整個binary search tree的root，_min就是最左邊(data最小)的node，_dummy則是最右邊(data最大) node的right child
3. 在iterator中的++及--：

和前兩者不同，tree因為在insert時就考慮到了排序，因此++和—的實作上較複雜。++為找尋successor，及排序後比這個node大的下一個node，方法為找到right subtree的最小值；--為找尋desuccessor，及排序後比這個node小的下一個node，方法為找到left subtree的最大值。另外須注意++/--在_min或_dummy的邊界情形。當current node是_min(最左)或_dummy的上一個(最右)時，進行上述演算法一定會回到_root，因此只要在迴圈內加一個(_parent == 0)的判斷，就可以確定是否為最左或最右node，這時候就不再進行++/--了(也就是回傳原本的node)。

Functions: (因未全部寫完，故只列出寫出的FUNCTION原理)

1. **begin()，end()，empty()**

三者分別記錄Tree的min、max、以及是否為空(沒root)

2. **Insert**

先處理empty的情形(_root == _dummy)。

在_root **new**一個node，把_right指向_dummy。

把_min指到_root，再把_dummy的_parent指到_root。

其餘的情形，就是從_root開始向下比較data大小，若insert的data比current node的data小，則往left child走，反之則往right child走。直到沒辦法再走下去(即將走到null pointer或是dummy node)，這時候就找到要插入的位置(_temp)。

接著連接pointer，將_ins的_parent指到_temp，再把_temp的_right或_left指到_ins。須特別注意是否需調整_min或_dummy。

3. 程式實作及比較

因在這次作業中，我並沒有成功實作出bst，因此在以下比較中只針對array跟dlist進行adta，不同的adtd進行測試並記錄結果；不紀錄adts的原因為array的sort為系統內建sort function而並非自己寫出；下方表格中，最左行為ADT，最上列為測資筆數。

Adta：

	50000	100000	200000
Array	TIME: 0.04s MEM: 4.148M	TIME: 0.09s MEM: 7.203M	TIME: 0.17s MEM: 13.38M
Dlist	TIME: 0.02s MEM: 4.23M	TIME: 0.03s MEM: 7.205M	TIME: 0.06s MEM: 13.17M

由實測可以發現，兩者速度上 Dlist 較 Array 快，而 memory 用量兩者差不多

奇怪之處為 array 沒有存取 pointer 卻和 dlist 有雷同的記憶耗費，估計原因為

我的程式問題。

Adtd -f：

	50000	100000	200000
Array	TIME: 0.00s	TIME: 0.01s	TIME: 0.02s
Dlist	TIME: 0.00s	TIME: 0.00s	TIME: 0.01s

由實測可以發現，兩者在從前方開始 delete 的速度差不多；對於 dlist 為何能

和 array 在此有相同之表現估計為從前方刪除之故

Adtd -b：

	50000	100000	200000
Array	TIME: 0.00s	TIME: 0.00s	TIME: 0.01s
Dlist	TIME: 0.00s	TIME: 0.00s	TIME: 0.01s

由實測可以發現，兩者在從後方開始 delete 的速度差不多；對於 dlist 為何能和 array 在此有相同之表現估計和 Adtd -f 之原因雷同，為有固定刪除程序。

Adtd -r：

	50000	100000	200000
Array	TIME: 0.01s	TIME: 0.00s	TIME: 0.03s
Dlist	TIME: 5.71s	TIME: 23.51s	TIME: 106.1s

由實測可以發現，兩者在隨機 delete 上的速度相差甚遠；array 一如前者而 dlist 則耗費了相當大的時間，估計為雖機刪除下 next 及 prev 的不斷變換所致