ADT Report

一.各個 ADT 簡介

1. Double link list:

   這個資料結構就像是串珠,每一個新資料加入時就像把一個珠子串上去，因此每次新增資料都要記得在自己前面的資料(_prev)以及在自己後面的資料(_next)，而在最後面會加上一個沒用的單元(_dummy)當作結尾，這種資料結構麻煩在於每次加上資料或是刪減資料都要保持_prev 及_next 有接到該接的地方，也因為這樣在每次更動資料時都要重新 assign *prev* 和_next，花的時間也會比較多。

2. Array:

   這個資料結構是預先先保留一個 2 的次方倍的大小作為儲存空間(capacity)，然後再用 size 來表示在這個空間內有多少區域已經被使用，而在儲存資料時只要一個個像排隊一樣排在一起而已，不需要記得自己前後的資料，另外刪減資料時也只要把最後一個資料填補到刪減處，然後 size 減 1 就好，因此不論刪減還是增加都很快。

3. Binary search tree:

   這個資料型態像個樹，由_root 開始分成兩邊(_left , right)接上資料，這個資料結構在增加及減少時就會保持原本的順序，必須符合一個重點，任何資料的_left 資料一定比自己小,而任何資料的_right 資料一定比自己大，因此在每次增加及刪減資料時都要很小心，要注意有沒有符合這個規定。

二.我的 implement 方法：

1. Double link list:

   這個資料結構會有不同 implement 方法的應該是在於 sort 的部分，這邊我使用 insertion sort 來實作他,是一種 O(n^2)的演算法,因此跑得滿慢的，此外在這裡如果我要調換資料順序時，我一開始是把整個 object 調換，這樣要做很多組_prev 和_next 的 assignment，會超耗費時間，因此之後在調換時，我只會把 object 裡面的資料(_data)交換，這樣會快很多。

2. Array:

   這個資料結構沒有什麼特別不同的地方，因為我也沒有去更動 sort 的寫法，只有用一開始在 std 裡面的 sort。

3. Binary search tree:

   在 implement 這個 tree 時，我設定每個 node 都會有四個 data member，分別是_data、_parent、_left、_right，而 tree 的部分也是有四個 data member，分別是_root、_upperbound、_lowerbound、_dummy，其中_root 代表最開始加入的資料(node)，_upperbound 及_lowerbound 則分別是最大值及最小值，而_dummy 這個 node 則是會接在最大值(_upperbound)的_right，用來代表 tree 的結尾，也就是 end()的地方，而其中比較特別的是我在 iterator 這個 class 裡面要傳入 tree，也就是 iterator 這個 class 有兩個 data member，分別是 BSTreeNode<T>* _node 以及 BSTree<T>* tree，在我一開始的設計中我是沒有加入 BSTree<T>* tree 這個 data member 的，然而當我在設計 operator ++及 operator –時會出現一個大問題，我在處理 operator ++及 operator –時會用到 BSTree 的兩個 function，分別是 successor 及 presuccessor，但是若要這麼做我的 successor 和 presuccessor 這兩個 function 必須是 static function，然而我在這兩個 function 裡面有用到_upperbound、_lowerbound 及_dummy，因此如果要把 function 改成 static，那上述三個 data member 也要是 static，但這是不可行的，因為每個 tree 有不同的三個_upperbound、_lowerbound 及_dummy，不能把他們設成 static，因此我上網查到的辦法就是可以在 iterator 產生時就要傳入 tree，而如果我們再操作 tree 時要產生這個 tree 的 iterator，那就把 this(指向這個 tree 的 pointer)傳進去即可，以下來講解一下我的 insert 及 erase 的 implement 方法：

   (1) insert:

   我 insert 的方法很簡單，會造成一個非常不 balance 的 tree，其實已經有點像 dlist 了，當沒有 node 時就把資料加入並設為_root，之後若是(a)要加入的資料比_root 小或是一樣大：那就我_root 左邊往下一個個找(就是不斷往下一個_left 去找)直到找到比它小的 node(假設為 A，要加入的 node 為 B)，這時把 A 設為 B 的_left，A 的_parent 設為 B，B 的_parent 則指向原本 A 的_parent，原本 A 的_parent 的_left 則指向 B，這樣就 ok 了，而如果找到最尾端都沒有比 B 小的值，那就把 B 接在最尾端，也就是_lowerbound 的_left，然後 B 的_parent 則指向_lowerbound，最後再重設_lowerbound 為 B(_lowerbound = B;)即可；(b))要加入的資料比_root 大：做法跟(a)的做法一樣，但是要注意在重

設_upperbound 值時要記得將_upperbound 的_right 指到_dummy。

(2) erase:

在我的設計中不論是 pop-front()、pop-back()、earse 或是 claer，我都適用 deleteNode 這個 function 來刪除 node，差別只在於刪除的 node 不同而已，因此只解釋 void deleteNode(BSTreeNode<T>* d)這個 function，首先我以 d 這個 node 的 child 的數量分成三種情況：分別是有兩個 child(_left , _right 都不是 NULL)，有一個 child(_left , _right 有一個是 NULL)，以及沒有 child(_left , right 都是 NULL)狀況，其中要注意對於 _upperbound 右邊有接一個_dummy，但在這裡我不把_dummy 算入 child 中，其中只有一個 child 和沒有 child 的狀況都非常簡單，只是把要刪掉的 node 刪掉，然後把上下的 node 之間的_parent、_left、_right 之間的關係整理好即可，比較複雜的是有兩個 child 的狀況，這個時候我要去找 d 的 successor s，而這裡又可以分成兩種狀況：s 沒有 child 以及 s 有 child(在這裡 s 一定只有 1 個 child，而且一定是_right 這個 child) ，因此 implement 比較複雜，有要顧好很多 pointer 的指向，很容易會忘記。

三.為什麼選擇這種 implement 方式：

這裡主要討論 bst 這個資料結構，在這裡我選擇了一般的方式，每個 node 依然會記住自己的_parent，這純粹因為我覺得要去想 trace 要怎麼記錄太麻煩，如果有_parent 的話就只要每次增加或是刪減都很小心的顧好_parent 就好，但沒想到這比我想像的麻煩，尤其在 deleteNode 的時候這真的是會常常忘記要把 _parent 重新接好，因此出現了一大堆 bug，也花了滿多時間去把她一個個接好，而一開始我為了方便選擇去加上_dummy 這個 node 來當作結尾也造成我後續要保持_dummy 正確接到_upperbound 的_right 花了滿大的心力，但整題來說我採用的方法對我來說就是最直觀的，把該有的東西都把明確的設出來；此外我在 insert 的時候使用了非常不 balance 的方法，這其實不是我一開始的做法，我一開始的做法是盡量去讓他 balance，也因此幾乎每個 node 都有兩個 child，但之後我會了測試到底 balance 的 tree 效率好還是 unbalance 的 tree 效率好的時候，我去發現 unbalanced 的 insert 和 erase 都比較快，因此我最後選擇用 unbalancd 的方式去完成。

四.實際測驗

(一)實驗設計：

1. 各個 ADT 比較：

   隨機的增加及刪減大量的 data(50000~100000)，然後比較各個 ADT 增加及漸少 data 的速度，另外對於 array 及 dlist 兩個資料結構則多做大資料量(100000)時 sort 的速度比較。

2. balanced tree V.S. unbalanced tree：

   用三種資料測試：(1) 隨機的增加及刪減大量的 data(50000~100000) (2)增加的資料會越來越大(一直往_root 的右邊增加) (3)增加的資料會越來越小(一直往_root 的左邊增加) 。

(二)預期結果：

1. 各個 ADT 比較：

   不論增加還是減少 data，performance 應該都是 array > dlist > bst。

   而 sort 的 performance 則是是 array > dlist。

2. balanced tree V.S. unbalanced tree：

   預測結果應該是 balanced tree 的 performance 會比較好，尤其是在第二種及第三種狀況的時候更是明顯。

(三)實驗結果：

各個 ADT 比較：

在各個 ADT 互相比較時，我發現不論是增加新的 node(push_back() or insert())，抑或是減少 node(earse()、pop_front() or pop_back())，執行的性率都是：array > dlist > bst，這也跟我預測的是一樣的，因為在增加新的 node 還有減少 node 的時候只有 array 不需要去保持各種 pointer(dlist 要保持 _prev、_next 及 _head，而 bst 則要保持 _left、_right、_root 甚至最差的時候連_upperbound、_lowerbound 及_dummy 都要小心保持)，所以 array 一定比較快，而 bst 最慢的原因除了他要保持的 pointer 多以外，另一個重要原因是 bst 在新增 node 是還要保持大小順序，而 dlist 不用，另外 array 的 sort 會比 dlist 快，因為在這裡我 dlist 的 sort 是 insertion sort，時間複雜度是 $O(n^2)$，而 array 用的 sort 是 std 裡面的 sort，時間複雜度是 O(log(n))，因此 array 當然比較快。

balanced tree V.S. unbalanced tree：

這個實驗的結果完全不如我預期，本來預期 balanced 的效率會比較好，否則不會有人特地去設計演算法，讓自己的 tree 是永遠 balanced 的，然而我做出的狀況是不論是 inssert 或是 deleteNode，我可以理解的是以我的排法，在 deleteNode 的時候因為大部分的 node 都是只有一個 child 的狀況，幾乎不會出現兩個 child 的狀況(只有刪到_root 才會有兩個 child)，因此狀況變得非常簡單，要重新 assign 的 pointer 數量會大幅減少，也不需要找 successor，另外找 successor 的步驟也會很簡單，如果有_right 那 successor 就直接是_right，沒有_right 那 successor 就是_parent，因此根本不會跑到 findmin 的迴圈中，因此時間大幅下降，但是 insert 為什麼會比較快就一直想不通，理論來說如果新加入的 node 非常大或非常小，那按照我的設計就會一直往右(_right)或往左(_left)跑，這樣會很耗費時間，但我發現在這種狀況下 balanced tree 的花費時間卻還是比 unbalanced tree 多一點，這是我一直想不透的，或許是我一開始 balanced tree 的 implement 就沒有很好吧，或是 balanced tree 的好處並不是發揮在 insert 和 erase 的效率上。