ADT Report

ㄧ.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。

1. 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。

1. 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()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。

1. 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。

1. 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。

1. 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。

1. 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。

1. 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()，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。