資料結構實作說明

一.實作說明:

Array 為最傳統的使用方式也是初學者接觸最容易入手的 ADT，動態宣告為以 heap 存取記憶體每個元素有固定 byte 差距。

DList 精神在於其 insert 以及做記憶體連結的彈性很高，並且能夠做到和 array 一樣的功能，美中不足的地方是其消耗時間複雜度較高，比較適合在元素較少的時候操作。

BSTree 透過一些演算的過程來縮短從 root 找到其對應的 node，其方便處在於耗費時間較 DList 短，因為需要經過的 node 不像 DList 那麼多，並可以實踐其 array 的操作。

二.實驗比較:

Array.h :

三種 ADT 裡面個人認為是最簡易好寫的，function 的內容就如同以往在陣列處理，在一開始的時候用動態宣告的方式從 heap 要一個名字為_data 的 Array，並使用變數_capacity 紀錄_data 陣列目前的容量。

在 pushpack(T &x)這個 funtion 功能裡面設定判斷 size 是否足夠，若不足夠，則用一 temp 動態宣告為原本容量兩倍的陣列然後將原本_data 的值 copy 至 temp 內後將原本_data 的內容還給 heap 後將其指到的位址指到 temp 讓 temp 所指的記憶體取代成為_data 原本的記憶體。

而 popback() function 而因為作業規定 adtdelete 的功能規定不能歸還記憶體，所以僅做 size 減少的動作而其真實記憶體容量不變且值亦存在其 size 外的記憶體內，只是在 adtprint 時並不會印出其內容。

popfront() function 先將紀錄的_size 的最後一個 element 的值輸入至第一個後將_size 減一達到跟 popback()一樣的動作

erase (iterator pos)和 erase (T &x)分別在指定 pos 的_node 指到的值和_data 中與 T &x 相符的陣列元素將其元素輸入最後一元素的值最後將最後一個元素 pop 掉。

而 clear 的動作發生時會將_data 歸還給 heap 並將本來的_capacity 設回 0，如同一開始建構子的狀態一樣。

Sort()則使用 STL 的標準函式並無其他更動。

begin()/end()則分別回傳 node 為陣列頭以及最後一元素後一位的 iterator 的 value。

其 operator overloading 就如同一般陣列的運作一樣，但一開始比較需要注意的是，改內部的「值」還是整個 class 的 object 改掉，像是"="這個運算子，一開始在寫的時候自己很猶豫要直接回傳 input 還是將*(this).node 的值給換掉。

而"++/--"皆會改變參數的值，其他別在於若需要 return 值至等號左邊的變數則在變數置於前與至於後，若欲執行加減動作的 iterator 置於加減號前會回傳本來沒有加減動作的值給左邊等號的變數，置於後則回傳經過++/--後的結果，而+=則類似++ i。

+則是回傳 iterator(_node + i)，即回傳一_node 值為_node + i 的 iterator。

而*()則回傳 iterator 本身所帶有的_node 的值。

Array 部分因為怎麼寫好像都差異不大，所以為何要這樣「實作」在此就不多加描述，三種 ADT 比較下來 Array 所耗費的時間複雜度遠低於其他兩個。

DList.h :

begin()回傳的是_node 值為_head 的 iterator 的 value。

end()因為沒有 pointer 紀錄 double link list 的 tail，所以使用 while 迴圈用一 temp 初始為 _head 然後 temp = temp->_next 疊代至_next 指向 null 時即為 tail。(tail 的_data 值為 T()即為亂數，實際上會 print 出的值只有到 end()._node 的_prev 指到的_data)

至於判斷是否 empty()其實自己覺得用 size()==0 或是 begin()==end()的時間複雜度都差不多是一個迴圈，我的 code 採用 begin()==end()來判斷。

push_back(T& x) function 先設一 temp 指到最後 end._node()的指標，再將原本 T()的_data 更新成有意義的值，並在 new 出一個新的 node 作為新的 end()然後將 isSorted 設為 false(因為新輸入的值未必是最大的)

pop_back()設一 temp 的 pointer 指向最後一個元素另再設一個 node_end 指向 end()的_node 來 delete 掉最後一個_node 並將 temp 指到的記憶體中的_data 改成 T()並且令其_next 指向 NULL。

pop_front()一開始本來是指標代換才實作這個功能，後來發現使用 iterator 指代換掉_data 的內容而非改掉指標指的記憶體意外的更快而且也比較不怕出現記憶體佔存錯誤的風險，在代換掉前一個_data 的內容後再使用 pop_back()將最後一個元素給 pop 掉。

erase(T& X)則是在使用 iterator 的 for each 找到對應的 X 值後將其 iterator 值作為參數置入 erase(iterator pos)中，而 erase(iterator pos)則是將 pos 後面的_data 值全部置入前一個元素內，後 pop_back()將最後一個元素給 pop 掉。(一開始也是打算用指標代換的方式，後發現這個方式比較快而且比較不容易在記憶體控管上出錯)

clear() function 是一開始令一指標 temp 指向 end()_node 然後再用 while 迴圈一個一個 delete 值到 temp 最後只到_head，後再將_head 的值設為一開始初始化 dummy node 的狀態。

Sort()當初設計時的想法是如果是採用其他 sort 的方式其實在 iterator ++或是—的時候也是透過迴圈來實作，這樣貌似跟直接用 insertion sort 的時間複雜度差異不大，所以這個就直接很乾脆的使用了 insertion sort。

Operator overloading 的部分和 Array 不同的地方是++跟—內部 node 的跳動是以_node = _node->_next 和_node = _node->_prev 來移動，剩下的部分與 Array 相同。

bst.h :

在 BSTree class 中設置的變數_root 為一開始樹的第一個 node，我在變數中加了一個 vector _trace 來存入排序好大小的 node 以及一個整數變數_size 來記錄我存入的 node 多寡，而 tail 則是作為 end()的 node 讓 iterator 在 for each 的時候可以做到==end()終止的動作。

而 iterator 內部亦有一個_trace 的 vector，我在 constructor overloading 中設計一個參數具有 vector 可以讓 BSTree 的_trace 可以 copy value 至 iterator 裡面，再用 count 這個變數

去計算我 vector 的 index 而為了判斷我是要 print reverse 還是 forward 在建構子當中也有一個 bool 可以去判斷。

在++/--運算子中做的動作僅是 count++與 count—然後再令_node = _trace[count]再 return 回來。

而在這邊"="是回傳整個 i 不是改變內部_node 值，為確保 print 出的值正確。

begin() function 跟 end() funtion 在_size 為 0 時 return iterator(_root)而在非 0 時則回傳 iterator(_trace[0],_trace,true)及 iterator(_trace[_size],_trace,false)其中的 true 跟 false 則是告訴我一開始的 iterator 中的 count 初始值是 0 還是_size。

empty()則是直接判斷_size 是否為 0。

Insert(T x)的部分是先用 temp 指向_root 後 while 迴圈移動，塞選機制即為比大小如果 x<temp 當下的_data 則判斷它有沒有 leftchild 如果有則 temp 繼續移動(temp = temp->leftchild)直到 temp 沒有 leftchild 為止，然後 x 就可以成為 temp 新長出來的左下 node 的值，而 x>temp 也是一樣的方式，讓 temp 一直往下走直到 temp 沒有 child 為止，則表示該 node 可以生長新的 leaf，而又找出 temp 中對應到的_trace 元素插入其中使得_trace 中的元素按照大小排序，會這樣插入是因為這樣就可以省掉在將 node 在計算 travel 路線推入_trace 的運算時間。

自己在 class 中有新增一個 function :isleftchild(node)來判斷其 node 為左 child 還是右 child。

而 popback 與 popfront 類似，主要概念為最大(最小)的 node 在函式內 pop 掉的時候需要判斷它是否還有 leftchild(popfront 為 rightchild)如果還有，則需要將其 child 的部分與 parent 做連結，而在 pop 的過程中還有一個需要特別注意的項目是"popnode 是否為_root"因為其 pop 動作與別的 node 不同，需要特別去注意。

而 erase(iterator pos) function 應該是其中最難寫的函式，為了省掉時間複雜度所以我做了很多的條件判斷，主要分為先從是否為_root 開始，那如果不是則判斷它為 leftchild 還是 rightchild(因為要做的指標代換動作不太一樣)，分好三類後在判斷其 popnode 是否同時具備 leftchild 與 rightchild，如只具備其一或是兩者不具備則處理的指標代換比兩者都具備簡單很多，因為其條件判斷是有效率的，而其連接點為何則仰賴_trace 中比大小的機制，如果 popnode 對應到的_trace 元素其 idx+1 的 node 會是其_data 大小會最接近 popnode 的 leftchild。

而 erase(T &x)的部分則是用 iterator for each 找出其對應到的 iterator 值傳入已經寫好的 erase(iterator pos)做處理。

本來的預期是可能 BSTree 可能在時間上不會差 array 太多，且沒想到 BSTree 需要使用 iterator，為了將其_trace 置入 iterator 這個問題想了一些時間。

總結來說，Array 在 ADT 的實作上個人認為最優，省掉許多記憶體也省掉很多時間，同時減少了空間與時間的複雜度，而 DList 就像老師上課說的，不知道為什麼而存在，耗時耗記憶體，想到些許的好處可能是在實作其 code 上比較沒有那麼困難，而且在操作記憶體連結上比較有彈性，相較於二元樹上比較單純，而二元樹的好處就在於省下了大量的時間，但始終還是沒有 array 快，且二元樹的缺點在於其結構如果在實作時並不整密很容易造成記憶體出問題，或是沒辦法連結到本來應該還存在的 node，造成 memory leak 的問題。