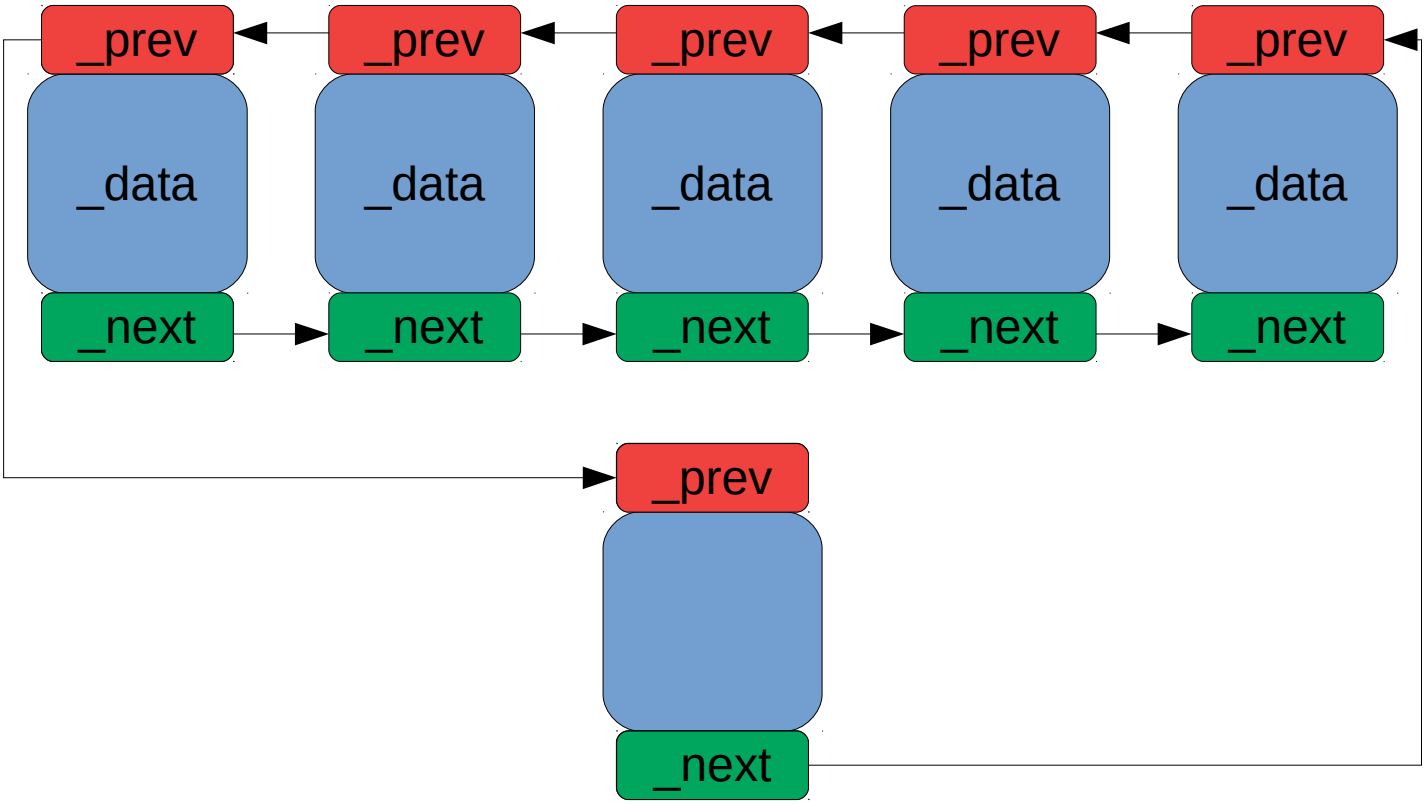
ADT Comparison Report

一、資料結構實做

1.簡介

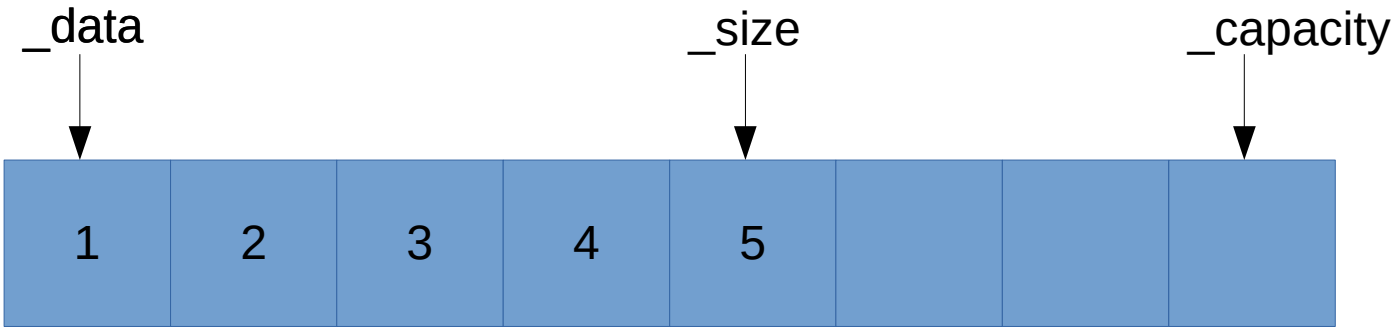
(1)Double-linked list

List 是一維的線性結構，其將資料存在 node 之中，再用指標以單一方向將 node 一個接一個串連起來，欲存取資料必須從第一個 node 開始沿著指標去尋找目標 node，而 double-linked list 則是用雙向的指標去串連 node，可以更快速的去找到最後一個 node，所以在某些操作上會更加快速。

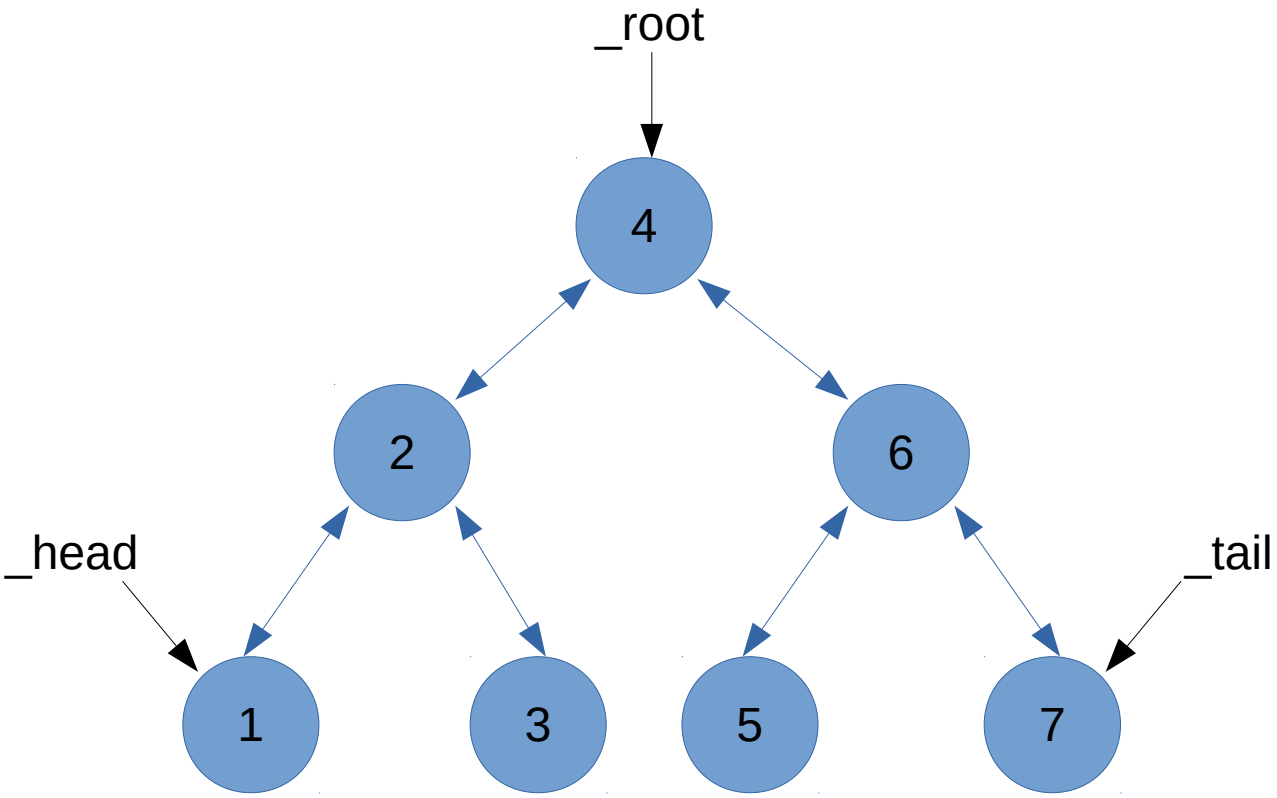


(2)Dynamic Array

Array 同樣也是一維的線性結構，不過使用的是一塊固定大小的連續記憶體區段，在操作時可以直接存取任一記憶體內的資料，而 dynamic Array 則是會隨資料量增加而自動增長的 Array，使用前便不需要預先設定容量，操作上更有彈性。



(3)Binary Search Tree
Tree 是階層結構，同樣以 node 的形式保存資料，以一個 node 為 root，向下連接多個 children node，這些 children node 底下分別再連接各自的 children node（稱作 subtree），依此類推，而從 root 走到任何一個 node 皆只有一種路徑。Binary Search Tree 則規定一個 node 最多只能有兩個 children node，且若資料之間可以進行比較，則 left subtree 裡任何一個 node 的資料都要比這個 node 小，right subtree 裡任何一個 node 的資料都要比這個 node 大。



2.程式實做方式
(1)Double-linked list
在 class DlistNode 中有三個 data member:
_data(存放資料)
_prev(指向前一個 node 的指標)
_next(指向後一個 node 的指標)
各個 DlistNode 由 class Dlist 進行管理，Dlist 裡面有兩個 data member:
_head（指向第一個 node 的指標）
_isSorted(用來檢查 Dlist 是否已排好序)
Dlist 包有一個 class iterator，iterator overloads *,++,--,=,!=,==，使用者可以利用 iterator access data。
實際操作 Double-linked list 則是透過 Dlist 中的 member function:
constructor:產生一個 dummy node，_head 指向它，並將它的_prev 跟 _next 都指向自己。
destructor:呼叫 clear 並 delete dummy node。
begin():回傳指向_head 的 iterator。
end():回傳指向 dummy node 的 iterator。

empty():檢查第一個 node 的_next 是不是自己，是自己的話就代表目前 Dlist 只有 dummy node，回傳 true。
size():從第一個開始數，直到數到 dummy node。
push_back():創建一個新的 node，它的_prev 是原本最後一個 node，它的 next 是 dummy node，同時也將原本最後一個 node 的_next 跟 dummy node 的_prev 指向它。若新的 node 是第一個加入 Dlist 的，則要把_head 指向它。
pop_front():把第二個 node 的_prev 指向 dummy node，把 dummy node 的_next 指向第二個 node，delete _head 並把_head 指向下一個 node。
pop_back():把 dummy node 的_prev 指向倒數第二個 node， 把倒數第二個 node 的_next 指向 dummy node，delete 最後一個 node。
erase(by pos):把 pos 下一個 node 的_prev 指向 pos 的前一個 node，把 pos 上一個 node 的_next 指向 pos 的後一個 node。如果 pos 是 begin 的話要把 _head 只向下一個 node。
erase(by data):一個一個找是否有符合的_data，找到的話呼叫 erase(by pos)。
find():用 iterator 一個一個找是否有符合的_data。
clear():反覆呼叫 pop_front 直到 empty。
sort():用 iterator 進行 bubble sort。

(2)Dynamic Array
在 class Array 中有四個 data member:
_data(指向第一筆資料)
_size(總共幾筆資料)
_capacity(可容納多少資料(以 2 的指數成長))
_isSorted(用來檢查 Array 是否已排好序)
Array 包有一個 class iterator，iterator overloads *,++,--,+,+=,=,!=,==，使用者可以利用 iterator access data。
class Array 中的 member function:
constructor:除_isSorted 外所有 data member 設為 0。
destructor:delete 所有 data。
begin():回傳指向第一筆資料的 iterator。
end():回傳指向最後一筆資料後一個位置的 iterator。
empty():!_size。
size():_size。
push_back():如果 Array 為 empty，把_capacity 設為 1，然後 new 記憶體; 如果 Array 已經滿了， 把_capacity 乘 2，new 一塊新的記憶體，delete 原本的記憶體。接著，把資料放在_size 的位置並把_size+1。
pop_front():用最後一筆資料取代第一筆資料然後 pop_back。
pop_back():_size-1。
erase(by pos):用最後一筆資料取代 pos 的資料然後 pop_back。
erase(by data):一個一個找是否有符合的_data，找到的話呼叫 erase(by pos)。

find():用 iterator 一個一個找是否有符合的_data。
clear():_size=0。
sort():使用老師寫好的演算法。

(3)Binary Search Tree
在 class BSTreeNode 中有四個 data member:
_data(存放資料)
_parent(指向 parent node)
_left(指向 left child node)
_right(指向 right child node)
各個 BSTreeNode 由 class BSTree 進行管理，BSTree 裡面有四個 data member:
_root(指向樹根)
_head(指向資料最小的 node)
_tail(指向資料最大的 node)
_size(總共幾個 node)
BSTree 包有一個 class iterator，iterator overloads *,++,--,=,!=,==，使用者可以利用 iterator access data。
比較特別的是 iterator 的 private data member 除了_node 外還多了 bool _OutOfRange，主要就是為了應付 end()這種超出現有 node 範圍之外的情況，在正常情況下_OutOfRange 都是 false，但是當_node→_right==NULL(亦即_node==_tail)卻還是要執行++時(即為 end())，便將_OutOfRange 設為 true，_node 不變，如此一來在做一end()時，只須將_OutOfRange 改回 false 即可得到 iterator(_tail)。不過由於多了這個 bool，所以在 overload != 及==時，要多考慮到_OutOfRange 的值，才不會將 iterator(_tail)跟 end() 當作一樣。
這個作法的限制就是無法應付像(end()++)++這樣的情況，不過考量到原本在操作的時候就不應該去使用這樣的 iterator，所以就不 handle 這樣的情況了。
實際操作 Binary Search Tree 則是透過 BSTree 中的 member function:
constructor:所有 data member 設為 0。
destructor:呼叫 clear。
begin():回傳指向_head 的 iterator。
end():回傳++iterator(_tail)。
empty():!_size。
size():_size。
insert():如果 BST 為 empty，新增 root node，否則將 insert data 與 root 進行比較，小於等於的話往左走，大於的話往右走，與下一個 node 進行比較，直到走到空位置為止，在該空位置新增一個 node 並將 data 存入。insert 完後再更新_head 及_tail。
pop_front():erase(iterator(_head))。
pop_back():erase(iterator(_tail))。
erase(by pos):如果要 erase 的 node 沒有 child，直接 delete 那個 node 即可；如果要 erase 的 node 有 1 個 child，delete 那個 node 並直接把它底下的

subtree 往上接；如果要 erase 的 node 有 2 個 child，先找到它的 successor(也就是在所有 data 比它大的 nodes 中，data 最小的那個 node)，然後用它的 successor 取代它，最後再 erase successor。在這過程中除了要調整_root,_head,_tail 之外，也要記得更新相關 node 的 _parent,_left,_right。
erase(by data):呼叫 find，找到的話呼叫 erase(by pos)。
find():我的做法是從_root 往下找，跟 insert 的方法相同，不過如果遇到要找的 data 就直接 return，如果走到了空位置便 return end()。
clear():pop back _size 次。
sort():空，因為 BST 的資料在 insert 的時候就已經排序好
print():以 in order 的方式 print BST
traverse():以遞迴的方式 traverse BST

3.優缺點
(1)Double-linked list
Double-linked list 之所以需要使用 dummy node 就是為了避免 end()的問題，而且在做 push_back,pop_front,pop_back 時也可以省去 empty 造成的困擾，減少 if...else...判斷式，code 自然會比較簡潔，唯一的缺點大概就是會需要多一個 node 的記憶體空間，還有在找_tail 的時候會需要多跑過一個 node，不過老實說成本真的很低，所以這樣的 trade-off 可謂非常划算。

Double-linked list 與 list 的比較，其優點必定是 push_back,pop_back 等操作的時間複雜度只需要 θ(1)，而 list 需要 θ(n)，快上許多，然而 trade-off 就是 node 都多了一個指標，吃的記憶體空間就會多上 8n。
Double-linked list 相較於 Array，它的優點是可以動態插入及刪除元素，而且也沒有容量限制，但是其缺點便是在 sort 的時候實在太慢了，每次移動元素都要重新做一次鍵結，著實浪費時間，但不鍵結的話資料便會遺失。
Double-linked list 相較於 Binary Search Tree，首先它的優點是 code 比較簡單，動態插入及刪除元素的速度也比 Binary Search Tree 快，但敗筆同樣在 sort 上面。

(2)Dynamic Array
Dynamic Array 其實除了這樣寫之外我也想不到其他寫法...，直接用指標去操作應該是最直接也最精簡的寫法，而且無論是在操作速度還是記憶體使用量上應該都算是最佳化了。我的寫法值得挑剃的地方大概是當_size>_capacity 時需要 new 另一塊全新的記憶體空間再把 data 都 copy 過去，這樣還蠻浪費時間的，更好的寫法應該是不需要 copy 過去，而是直接繼續跟記憶體要_size 之後的記憶體空間，但是這樣在 delete 上會變得麻煩，所以就還是保持原樣。

Dynamic Array 是我個人認為這三種中最棒的 ADT 了，首先它的 code 非常好寫，因為其實就只是指標相加減而已，第二，在跑所有 test dofile 時永遠是最快的，第三，它支援 random access，不像 list 或 tree 一定要從某個點開始走，所以除了 find 跟 sort 以外所有的操作都只需要 θ(1)(在 insert 及 erase 若

不需要維持 element 相對位置的情況下)，雖然 Dynamic Array sort 的速度比 Binary Search Tree 慢，但由於其記憶體連續又可 random access 的緣故，所以它 sort 的速度遠比 list 快上許多，同時它在 push_back,pop_back,traverse 的時候也不需要像 Binary Search Tree 一樣費時，故以綜合來看，Dynamic Array 是我個人認為最好用的 ADT 了。
不過它最大的缺點也就如上所說，在不改變相對位置的 insert 及 erase 上時間複雜度會達到 θ (n)。而且 sort 跟 find 也比 Binary Search Tree 還慢。還有 capacity 的限制，capacity 往往比實際需求大，導致 Dynamic Array 感覺起來並不節省記憶體空間。

(3)Binary Search Tree
我的寫法跟老師不同，我有使用到_parent,_head,_tail 等，在做 iterator 時也不是使用 trace 去 handle end()而是用一個 bool variable，然後還多存了一個 _size。
有 parent 的好處當然是在能夠自由 traverse tree，寫 code 時比較易於思考，但是在 insert 跟 erase node 的時候 code 就會變得比較複雜，因為除了 children 要重接之外，parent 也要重接，而且每個 node 的 memory usage 也會變大。
而我之所以會使用 head 跟 tail 的原因主要是覺得這樣 begin(),end(),pop_front(),push_back()這些操作就不需要每次都從 root 去找起，可以加快一點效率，不過同樣每次 insert 跟 erase node 的時候就要去更新 head 跟 tail 的位置。
至於在 iterator 中增添 bool variable 的原因在上文有說明，這樣做的優點就是比起在每個 iterator 中都多一個 trace，能夠省下不少記憶體空間，缺點可能就在 erase 的時候比起用 trace 會慢上許多。
多寫一個_size 的好處是每次呼叫的時候就不需要重數一次，而且也不過就一個 size_t 的大小，不用白不用。

Binary Search Tree 最大的優點便是它的 sort 時間複雜度為 θ (1)，而其他兩種 ADT sort 的時間複雜度最低也是 θ (nlogn)，而且在 find 的時候也較其他兩者為快。
但是 Binary Search Tree 的缺點也是顯而易見，除了 code 很難寫，需要使用到的記憶體較大之外，在 insert 跟 erase 時都需要比較多的時間，因為它在進行 insert 或 erase 的同時還要維持排序，也就是說，雖然它 sort 及 find 的時間複雜度低，但它就只是將時間轉移到了 insert 跟 erase 上面。

綜上所述，在實際使用這些 ADT 時，還是要根據實際的使用需求來決定；若是常常需要做資料的新增或刪除，且不太在意排序的話，那麼 Double-linked list 是不錯的選擇；若不常有資料的移入移出，且常常需要查找資料時，Binary Search Tree 便很好用；若對於上述兩者都沒有強烈的需求的話，Dynamic Array 絕對是首選。

二、實驗比較
1.實驗設計
針對 ADT 的各個 function 進行測試，包含:
push_back()/insert(),pop_front(),pop_back(),erase(),find(),sort(),print()
分別比較三種 ADT 在 memory usage 及 time usage 上的差異。

2.實驗預期
memory usage:Array<Dlist<BST
(1)push_back()/insert()/pop_front()/pop_back()/erase():
    time:Dlist~Array<BST
(2):find():
    time:BST<Array< Dlist
(3):sort():
    time:BST<Array<Dlist
(4):print():
    time:Dlist~Array~BST

3.結果比較與討論

	Dlist	Array	BST
memory usage (100000 筆 data)	7.109Mb	7.152Mb	7.234Mb
memory usage (65536 筆 data)	4.992Mb	3.934Mb	5.012Mb
push_back()/ insert() (100000 筆 data)	23.33s	0.11s	0.19s
pop_front() (1000 筆 data)	0s	0s	0s
pop_back() (1000 筆 data)	0s	0s	0s
erase() (隨機 1000 筆 data)	0.47s	0s	3.4s
find() (in 100000 筆 data)	0.67s	0.16s	0s
Sort() (10000 筆 data)	7.74s	0.02s	0s
Print() (100000 筆 data)	0.07s	0.13s	0.33s

討論:

(1)在 100000 筆 data 的情況下三種 ADT 的 memory usage 看似相差不大，但在 65536 筆 data 的情況下三者就出現了差距，且跟我的預期相同，原因是因為 Array 的 capacity 是以 2 的指數成長，當有 100000 筆 data 時，Array 的 capacity 為 131072，也就是說相較於其他 ADT，它足足多要了 31072 個 data 的大小，所以看上去的 memory usage 便會很大，然而實際上使用到的並沒有這麼多，為了突顯 capacity 的影響，刻意做了 65536 筆 data 這組數據，果然就跟我預期的相同了。仔細去計算的話，Array 每存一筆 data 需要 sizeof(T)，Dlis 需要 sizeof(T)+16bytes，BST 需要 sizeof(T)+24bytes，實驗結果與此吻合。

(2)出乎預料的是，Dlist 在 add data 上竟然如此緩慢，原本以為最慢的會是 BST，因為還要從根開始去找插入葉子的位置，而 Dlist 直接透過 dummy node 去插入最後的位置就好，但是結果顯然不是如此，我不知道為何會發生這種事情，而且就連 reference program 在我的電腦上跑也這麼慢，令人疑惑。

(3)三種 BST 的 pop_front 及 pop_back 幾乎都是瞬間完成，還蠻合理的，Dlist 本來就有存 head，去找 tail 也易如反掌，而 Array 可以 random access 自然不在話下，至於 BST 由於我有存 head 跟 tail 所以也能馬上 access。

(4)erase 最快的是 Array，因為不在乎順序，所以只要直接複製最後一個位置的 data 再 pop_back 就好，其次是 Dlist，因為要做指標的重新串連，最慢的是 BST，如果刪除的是 node degree<2 的話很簡單，但是若刪除的 node degree=2 的話就必須先找到 successor，把 successor 的 data 複製過來，再去刪除 successor，這個過程相當耗費時間。

(5)find 最快的是 BST，因為 time complexity 原本就介在 $\log_2(n)$及 n 之間，其他兩者的 time complexity 皆為 n，至於為何 Array find 的速度比 Dlist 快應該是實驗誤差。

(6)BST 因為不用 sort 所以最快，Array 其次，Dlist 最慢，主要的原因在於 Dlist 要 swap 兩個 node 的 data 比 Array 麻煩，而且由於 Dlist 是用 bubble sort 而 Array 是用 quick sort，所以在演算法上就有差異了。

(7)print 的速度就是 traverse 的速度，可發現三種 ADT traverse 的速度差不多，BST 因為結構較複雜所以可能稍慢一點。