B04901060 電機二 黃文璁

一、 資料結構實作

（一）Dynamic Array

初始化時 _capacity 為 0。

在有新元素加入 Dynamic Array 時，_size 也會隨之增加，但若 _capacity 不夠時，會自動呼叫 reserve() 函數，使得有足夠的空間放入新元素。pop_back(), pop_front() 等實作中操作也和一般陣列操作方法類似，故不多加描述。

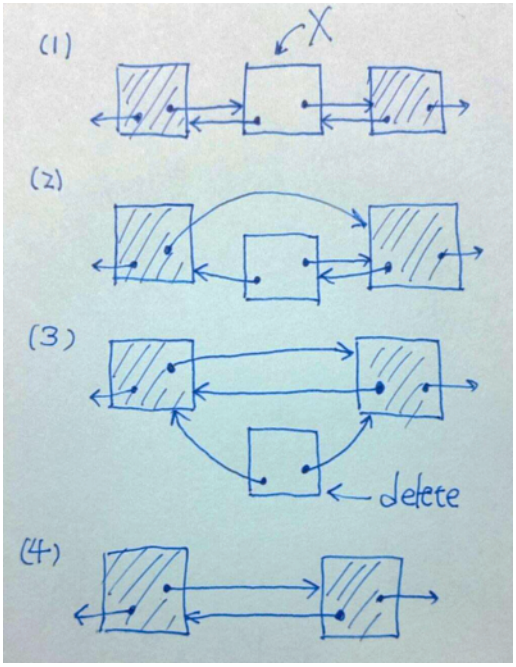
1. Iterator 的 operator++, operator--, operator[] 等操作
   由於 Dynamic Array 實際上和一般陣列類似，但可以不用一開始就宣告一個固定大小的陣列。Dynamic Array 的記憶體配置也和一般陣列類似，故這些 operator 的存取方式也和一般陣列類似。
2. reserve(n)
   若 n 比當前 _capacity 小則不做動作。若比 _capacity 大則：若 _capacity 為 0，則將 _capacity 增加到 1；若 _capacity 不為 0，則將 _capacity 乘上 2。接著 new 一塊新的 _capacity 大小的記憶體，再利用 std::move 將原先記憶體內容移動到新的記憶體，最後 delete[] 原先的記憶體即可。
3. push_back()
   先檢查 _capacity 是否足夠，若不夠的話呼叫 reserve(_capacity+1)。

（二）Double Linked List

實作細節：

DList 中有 _head、_end 兩個節點，分別對應到 STL 的 begin() 和 end() 的概念，也就是 _end 代表的是最後一個元素的「下一個」。

push_back(), pop_front(), pop_back(), erase() 等操作的重點即在於 _prev, _next 的操作，也就是要如何重新連接節點。舉 erase() 為例，如右圖：



1. 刪除元素的操作：erase(), pop_back(), pop_front()
   刪除時都要注意一些特例或邊界條件。如：
   　刪除 _head、DList 的 _size 為 1 時…等情況，以免刪除不該刪除的節點。
2. operator++, operator-- 等操作
   每個節點上有_prev、_next 指標，++ 時往 _next 移動、-- 時往 _prev 移動即可。由於
3. sort()
   使用 quicksort 演算法實作 DList::sort()。
   quicksort 在隨機資料下的期望複雜度為 O(n log n)

（三）**Binary Search Tree**

- BSTreeNode
  節點中存有：1. 該節點的 data,　2. Left child,
  　3. Right child,　4. Parent,　5. 重複次數 _repeat
  關於重複次數的功能將在下節說明。
  此外 BSTreeNode 中還宣告了一些 helper function，例如 minChild(), maxChild()，可以回傳該節點 child 中 data 最小的 child，以及 nextNode(), prevNode()，分別代表的是 iterator 的 ++, -- 操作。

- BSTree
  1. 重複元素
     本次作業中採用隨機測資，可能會有重複的元素。其中兩種處理方法為：
     (1) 每個重複元素都當作不同節點
     (2) 在節點上紀錄該 data 的重複次數
     可以列出幾個方法 2 相對於方法 1 的優點:
     - erase 時，若刪除的是重複元素，則只要將節點的 _repeat 減 1 即可。
     - iterator++, iterator-- 時若 iterator 在該節點還沒有移動超過 _repeat 次，則 iterator 就不用實際呼叫 nextNode(), prevNode() 計算下個節點。

     當然缺點也是顯然的，就是需要多一個 int 儲存 _repeat，不過若重複次數較多的情況下，方法 2 在空間、時間上都有很大優勢。

2. iterator++, iterator--

   對於這兩個操作而言，若不是在 begin(), end() 附近，則兩個操作基本上是對稱的，故以下只討論 iterator++ 的情況。

   iterator++

   (1) 重複元素

   在 iterator 中也存有一個 _repeat 代表停留在此節點的次數，若 iterator 的 _repeat 小於當前節點的 _repeat 數，則不移動 iterator，僅做 ++_repeat。

   (2) iterator 所在的 node 有 left child

   由於是 ++ 操作，但左子樹中所有元素都小於此節點，故不考慮此情況

   (3) iterator 所在的 node 有 right child

   將 iterator 移動到右子樹中最小的節點，可呼叫前述的 nextNode() 達成

   (4) iterator 所在的 node 沒有 right child

   將 iterator 不斷往 parent 移動，直到 node 為 parent 的 left child。

3. end()

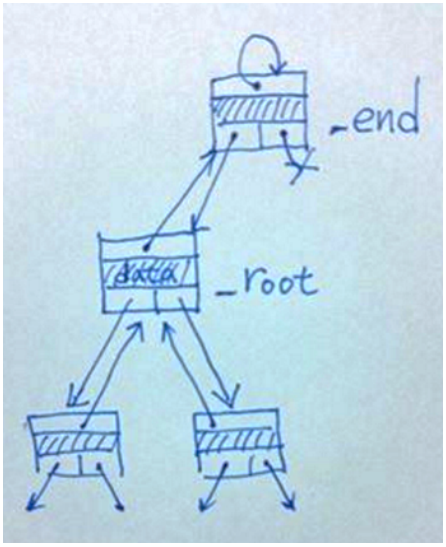
   關於 end() 的實作，其實可從前述 iterator++ 的討論中看出端倪。

   在剛剛狀況 (3) 中的這句

   直到 node 為 parent 的 left child

   其實給了一個啟發就是若將 _root 的 parent 設為另一個節點 _end，那當 ++ 到最後時，iterator 就會自動跑到 _end，接著再判斷該節點是不是 _end 即可。

   可參考右圖。



4. erase(), pop_back(), pop_front()

   關於 BST 的這些操作，就分為四個情況

   (1) 節點沒有 children

   (2) 節點只有 left child

   (3) 節點只有 right child

   (4) 節點有兩個 children

   另外還要判斷被刪除的節點是否為 _root，對 _end 作出更新。

   最後呼叫 helper function: updateBegin() 對此 BST 的 begin iterator 作出更新。最後將該節點 delete 掉。

5. insert()

   注意到 insert 操作也可能導致 begin iterator 的更新，但情況較為簡單：

   若 insert 的元素是原先 begin iterator 的 left child，那 insert 的節點即為新的 begin iterator 的位置。

二、 實驗比較

1. 設計

這次作業基本上是要比較：插入、刪除、排序等操作的效率，故可藉由不同數量級的測試資料大小來進行比較。除了時間外也可比較記憶體用量。

2. 插入

使用隨機數據測試 3 次的平均值，利用指令：adta –r x。

以下為利用 push_back/insert 插入。

次數	Dynamic Array	DList	BST
10000	0s /0.848M	0s /0.531M	0s /0.691M
100000	0.02s /6.109M	0.01s /4.715M	0.05s /6.328M
200000	0.04s /12.12M	0.02s /9.367M	0.13s /12.36M
500000	0.09s /24.12M	0.05s /23.31M	0.44s /30.42M
1000000	0.16s /48.09M	0.11s /46.56M	1.07s /59.52M
5000000	1.18s /384.1M	0.53s /232.6M	7.4s /253.1M

可以注意到 DList 插入的速度甚至比 Array 還要來得快，這可能是由於 DList 沒有 Array 重新 new 一塊記憶體然後移動原始資料這樣的動作。

本實驗中 Array 和 DList 都是用 push_back，複雜度為 O(1)，故速度顯然較 BST 的 O(logn) 的 insert 來得快。但值得注意的是若插入在中間元素的話：

Array 的複雜度為 O(n)

DList 的複雜度為 O(1)

BST 的複雜度期望為 O(logn)

實際上 DList 插入仍為最快。故 DList 適用在插入、刪除頻繁的應用上，例如遊戲場景的渲染。

3. 刪除

先插入 200000 次，再進行隨機刪除，利用：adtd –r x。

次數	Dynamic Array	DList	BST
100	0s	0.05s	0.73s
200	0s	0.09s	1.52s
500	0s	0.24s	3.97s
1000	0s	0.52s	7.70s
2000	0s	0.97s	15.67s
5000	0s	2.39s	39.09s
10000	0.04s	4.73s	89.74s

本實驗中 Array 的 erase 和 STL 的 erase 規則不同，會影響元素的順序。
若要不影響順序的話，在中間刪除元素的複雜度分別為：

Array: O(n)
DList: O(1)
BST: 期望 O(logn)

故 DList 實際上最適用於刪除元素頻繁的應用。
值得注意的是一般如 std::vector 的 dynamic array 刪除尾端元素的複雜度仍為 O(1)，故若應用時只需刪除尾端元素的話使用 dynamic array 是更好的選擇。

4. 排序

先隨機插入 x 筆數據，再進行排序，利用：adts。

次數	Dynamic Array	DList	BST
50000	0.02s	0.02s	0s
100000	0.04s	0.04s	0s
200000	0.07s	0.09s	0s
500000	0.19s	0.23s	0s
1000000	0.41s	0.49s	0s
2000000	0.85s	1.05s	0s
5000000	2.24s	2.85s	0s

值得注意的是 BST 的遍歷本來就是有序的，故排序不需要時間。
而本次實作使用在 DList::sort() 上的 quicksort 演算法，期望複雜度和 dynamic array 使用的 std::sort() 都是 O(n log n)，故可以發現所花的時間成長速度相似。