Performance study report

電機二 B06901063 黃士豪

一、 資料結構的實作

1. Array

(1) 資料結構與操作：

在記憶體中取一段連續、固定大小的記憶體空間，當遇到空間不夠的情況時，就再配置一塊更大的空間，並將原本位置儲存的資料複製過去，刪除原資料儲存區。

(2) iterator：

在 iterator 中存入目前指向位置的 pointer。因為是連續的空間，因此其中的 operator++與 operator--只要直接操作_node 即可。

```
iterator& operator ++ () { ++_node; return (*this); }
iterator operator ++ (int) { iterator result(*this); ++_node; return result; }
iterator& operator -- () { --_node; return (*this); }
iterator operator -- (int) { iterator result(*this); --_node; return result; }
```

(3) 實作方式：

在 array 剛開始要被存入資料時，取一塊_capacity 大小的記憶體，並以_data 這個 pointer 去指向這塊空間的開頭。

```
_data = new T[_capacity];
```

隨著資料的存入，前面分配到的大小可能會用完，因此在用完後要如(1)所說的，擴大_capacity 再重新取一塊記憶體。此時要記得將原本的資料複製過去，並刪除不用的記憶體。

```
if(_capacity == 0) ++_capacity;
else _capacity = _capacity << 1;
T* tmp = _data;
_data = new T[_capacity];
for(size_t i = 0; i < _size; ++i)
    _data[i] = tmp[i];
delete []tmp;
```

因為這塊空間是連續的，因此我們可以將_data 當作一個原先系統就有支援的 array 去操作，並藉此 overloading[]這個 operator。

```
T& operator [] (size_t i) { return _data[i]; }
const T& operator [] (size_t i) const { return _data[i]; }
```

因為在 array 中，每個位置都是可以 random access 的，且在此的 pop_front 並不考慮資料排序，因此做的各函式操作皆直接用 operator[]完成，影響的僅只有_size 的大小。

另外，sort 的部分直接使用 std::sort，不再贅述。

(4) 原因：

優：dynamic array 是個容易取得一塊連續記憶體的辦法，且藉此取得的記憶體空間可以直接使用預設的 operator 操作，操作較為方便也快速。

缺：若一開始取的空間不夠，要再要一塊更大的空間時，需要將原先空間內的資料複製到新的位置，花費額外的時間去進行操作。

2. Double Linked List

(1) 資料結構與操作：

將資料儲存在一個叫做 DListNode 的 class 中，另外在這 class 中還儲存了_prev、_next 兩個指向其他 DListNode 的 pointer，藉此紀錄每個 node 的前後關係，將大量不連續記憶體串在一起，藉由 DList 這個 class 去將這些散落的記憶體串接並操作。

(2) iterator：

與 array 相同，dlist 的 iterator 也只存了一個 pointer，但是是 DListNode 的 pointer。因為是不連續的記憶體，進行 operator++與 operator—時不能簡單進行_node 的++--，改為++時將_node 指向當前_node 的_next，--時則為_prev。另外，也不再支援 operator+，因為 call operator+相當於一次呼叫數個 operator++，再額外建立一個 function 繁冗而不必要。

```
iterator& operator ++ () { _node = _node->_next; return *(this); }
iterator operator ++ (int) { iterator result(*(this)); _node = _node->_next; return result; }
iterator& operator -- () { _node = _node->_prev; return *(this); }
iterator operator -- (int) { iterator result(*(this)); _node = _node->_prev; return result; }
```

(3) 實作方式：

在 Dlist 建立的同時，於 constructor 中先建立一個 DListNode 作為 dummy node，令這個 node 的_prev 和_next 皆為自己本身，將_head 指向這個 node。之所以要建立這個 dummy node，是要方便定位沒有資料時_head 的指向位置，並將整個資料串成一個環狀，方便取得整個資料尾端的位置。(因為沒有另建立一個指向_tail 的 pointer，因此要取得尾端的資料只能利用 dummy node 將資料頭尾串接，dummy node 的_next 為頭，_prev 即為尾。)

```
DList() {
    _head = new DListNode<T>(T());
    _head->_prev = _head->_next = _head;
    _isSorted = false;
}
```

此時，每做一次資料的變更就是 new/delet 一個 DListNode，並改變插入位置前後儲存的_next、_prev 指標位置。在此舉 erase 為例(因為 erase 可以套用到所有 pop 的操作，而 push 則相反即可)。

```
bool erase(iterator pos)
{
    if(empty()) return false;
    if(pos._node == _head) _head = pos._node->_next;
    pos._node->_prev->_next = pos._node->_next;
    pos._node->_next->_prev = pos._node->_prev;
    delete pos._node;
    return true;
}
```

其他 function 的操作皆藉由 iterator 來操作，在此不多贅述。值得一提的是，因為在此的 iterator 只有++、--兩個移動的功能，沒有 random access 的操作，因此 sort 無法使用 std::sort，必

須自己重新實作 sort function。因為每個 node 的重新連結牽扯到前後 node 內存的 pointer 值變更，而且還有_head 位置改變的問題，相對比較麻煩，因此在資料大小不大的前提下，我所實作的 sort 選擇直接交換 node 裡面_data 的值。

在此我實作了三種 sort：Insertion Sort, Merge Sort, Quick Sort。一開始實作完 Insertion Sort 後，發現因為複雜度是 O(n²)，執行教授給的 do2 檔速度十分緩慢，甚至有時候不確定是程式當掉還是還沒跑完，因此我決定來實作 Merge Sort。

我的 Merge Sort 分成 recursion 和 iteration 兩種寫法，原本以為 recursion 因為要重複 function call 速度會比較慢，沒想到其實沒差多少。不過實行上因為要取得正中間資料的 iterator，不斷進行++iterator 應該消耗了許多時間。

```
void mergeSort(iterator first, size_t size) const
void mergeSort_iteration(iterator first, iterator last, size_t size) const
```

此外，因為教授建議不要另外開 container 來存裡面的資料，因此我 merge 時採用 in place 做法，不另外增加記憶體消耗量。不過因為在 merge 的過程中需要不斷的 shift，因此速度明顯被拖慢，沒有達到一般 O(nlogn)的水平。(下圖為 shift 過程中最耗時的操作)

```
for(; c != a; --c) c._node->_data = c._node->_prev->_data;
```

因為 Merge Sort 的操作不便與不理想的操作結果，我最後採用 Quick Sort 來做為 dlist 的 Sort。iteration 版本的 Quick Sort 雖然多開了一個 array 作為堆疊儲存當前比較區間頭尾的 iterator，不過因為 in place 的 partition 操作沒有 shift 的動作，只有資料的 swap，速度比起 Merge Sort 根本是快得飛起，do2 的時間直接壓在 3 秒內解決。

```
pair<iterator, iterator> stack[size];
stack[top++] = make_pair(first, last);
```

(4) 原因：

優：若將資料在頭尾做操作的話速度非常快(雖然還是跟 array 差不多)，只要再連接上一個 node 即可，不用再多花時間移動到頭尾。另外，記憶體空間不必連續，可以善用記憶體的零碎空間。

缺：沒辦法 random access 到中段的資料，且不管需要哪個位置的資料都必須要從頭或尾開始進行++或--的操作。

3. Binary Search Tree

(1) 資料結構與操作：

bst 的 node 與 dlist 大同小異，但最大的差異在 BSTreeNode 中存的 pointer 從_prev, _next 改成了_leftChild, _rightChild, _parent 三種。如此的資料結構會使資料形成一顆樹的型態，在_leftChild 下所有的 node 都會比自身的_data 小，相對的_rightChild 下的會比自身大，而 node 本身又接在自己的 _parent 的左/右接腳上。因此此操作能在資料 insert 時就已經先排序好，比起 dlist 還要再進行 sort 相對方便。

(2) iterator：

相比起 array 和 dlist，bst 的 operator++與--麻煩許多。因為 bst 左小右大的特性，operator++的實作方式我選擇取以當前 node 的_rightChild 為 root node 的樹最小值作為++後的 node，若沒有右樹的話，則不斷返回前一層的_parent，直到不再是右支為止(即比當前 node 大)。同樣的，operator--的情況也相同，不過皆改為 _leftChild。

(3) 實作方式：

在實作 bst 時，為了方便 call end()這個 function，我有多存了一個 dummy node 作為_tail，並設了一個_root 指標指到整棵樹的 root node。

與 array、dlist 不同的是，bst 沒有另外存一個 pointer 指向整個樹的最小值，因此 begin()這個 function 需要額外的時間從 _root traverse 到最小值，需要花費額外的時間。

此外，bst 使用的也不是 push_back()這個 function，而是使用 insert()，因為 bst 在加入新的資料同時已經將新的資料依照大小插入了，沒有所謂的"加在後面"，也省去了 sort 所需的時間。我的 insert 用 recursion 來做，假如輸入的資料比當前 node 大的話，就加在這個 node 的_rightChild，若已經有_rightChild 了，就再去呼叫此 node 的右腳的 insert，反之亦然。比較麻煩的是若加入的資料是最大的，要另外再處理 dummy node。

```
if(!node->_rightChild)
{
   BSTreeNode<T>* newNode = new BSTreeNode<T>(x, 0, 0, node);
   node->_rightChild = newNode;
}
else if(node->_rightChild == _tail)
{
   BSTreeNode<T>* newNode = new BSTreeNode<T>(x, 0, _tail, node);
   node->_rightChild = newNode;
   _tail->_parent = newNode;
}
else insert(x, node->_rightChild);
```

為了方便起見，我將 pop_front、pop_back、erase 全部利用一個 private function : removeNode 來實作。在這個 function 中，我將移除資料的情形分成三種 case。第一種，移除的這個 node 有其中一邊的 Child 是 0，這種情形只要將非 0 的 Child 直接取代當前 node 或直接將此 node 移除即可。第二種，_leftChild 有 node，_rightChild 是_tail，這種情形我把_tail 先移除，當成前一個 case，再把_tail 接到這個新的 node 下的 tree 的最大值。最後一種 case 是兩隻腳都是別的 node，那麼只好把這個 node 的_rightChild 這棵樹上最小值移過來作為新的 node，並移除這個最小值。如此這個 case 就變成第一種 case 了。

(4) 原因：

優：因為在 insert 的時候已經按照順序插入了，因此省去了 sort 的時間，也因此在執行 find()函式的時候速度比 dlist 和 array 的速度快。

缺：每次插入都要尋找對的位置插入，因此速度較慢，在刪除的時候也因為要移動 node 因此減慢速度。

二、 實驗比較

1. 實驗設計

以下紀錄各指令執行時間與消耗記憶體

(1) 隨機生成 100000 筆資料(adta -r 100000)

(2) 印出資料(adtp)

(3) 排序資料(adts)

(4) 隨機刪除 10000 筆資料(adtd -r 10000)

(5) 從前方刪除 20000 筆資料(adtd -f 20000)

(6) 從後方刪除 20000 筆資料(adtd -b 20000)

(7) 清空並將資料長度改為 5(adtr 5)

(8) 前置：

- 生成 49999 筆資料(adta -r 49999)
- 生成資料"ric" (adta -s ric)
- 生成 50000 筆資料(adta -r 50000)

尋找資料"ric"(adtq ric)

2. 實驗預期

速度：

(1) 生成資料的速度為 Dlist > Array >> Bst

原因：Array 還需要取得空間的時間；Bst 還需要排序的時間。

(2) 印出資料的速度為 Array = Dlist > Bst

原因：Bst 在 traverse 時有時候會有回到_parent 的情形，消耗額外的時間。

(3) 排序資料的速度為 Bst >> Array > Dlist

原因：Bst 原先就是排序好的資料，而因為 Array 可以 random access，會比 Dlist 快一點。

(4) 隨機刪除資料的速度為 Array > Bst > Dlist

原因：Array 不注重排序，因此速度極快；Bst 還需要重新改變 node 的位置；Dlist 光隨機 access 到資料就要額外的時間，因此最慢。

(5) 從前刪除資料的速度為 Array = Dlist > Bst

原因：只有 Bst 需要多花 traverse 的時間。

(6) 從後刪除資料的速度為 Array = Dlist > Bst

原因：只有 Bst 需要多花 traverse 的時間。

(7) 重置資料的速度為 Array = Dlist ≧ Bst

原因：可以視為從前方或後方刪除全部資料，而 Bst 需要多花時間 traverse 取得全部資料的位置。

(8) 搜尋資料的速度為 Bst > Array = Dlist

原因：Bst 的 Worst case 為 O(logn)，因為每經過一個節點都能刪掉一半的可能性；而 Array 和 Dlist 都是 O(n)因為要全部走完。

使用空間：

Array 使用大小應會略大於 Dlist 和 Bst，因為會有多開的空間存在。但 Dlist 因為無法 random access，因此 sorting 需要另外的空間儲存指標資訊。

3. 實驗結果

	Array		Dlist		Bst	
指令	時間	記憶體	時間	記憶體	時間	記憶體
adta -r 100000	0.03s	5.133MB	0.01s	5.102MB	0.07s	5.121MB
adtp(100000)	0.04s	0MB	0.06s	0MB	0.06s	0MB
adts(100000)	0.03s	0MB	0.03s	1.836MB	0s	0MB
adtd -r 10000	0.03s	0MB	2.23s	0MB	0.01s	0.742MB
adtd -f 20000	0s	0MB	0.01s	0MB	0s	0MB
adtd -b 20000	0s	0MB	0s	0MB	0s	0MB
adtr 5(100000)	0s	0MB	0.01s	0MB	0.02s	0MB
adtq ric	0.01s	0MB	0s	0MB	0.09s	0MB

*指令後的括號代表執行時資料量

4. 實驗比較

與預期不符的實驗有(2)(3)(5)(8)。

(1) 實驗(2) Dlist 印出資料的時間比預期的慢，推測可能是因為重複進行 operator++，而改變 node 的 copy 行為減慢了速度。

(2) 實驗(4) Bst 隨機刪除的速度意外的比 Array 快，推測是在 erase 時我 Array 的 find()用的是從頭搜尋，導致花費更多時間。

(3) 實驗(5) Dlist 從前刪除的速度比預期慢，推測因為需要移動 _head 的位置，因而減慢程式速度。

(4) 實驗(8) Bst 的搜尋速度異常的慢，推測是因為此 Bst 並非平衡樹，使得若某一分支深度特別深，搜尋的速度就會被拖慢。