資結 HW5 報告

Array, DList, BST

B07901010

范詠為

一、 實作部分

1. array.h

a. class iterator

這部分因為 array 是連續的記憶體空間，所以 overloading++/--時基本起只要 _node++/--即可。

b. begin(), end(), empty(), size()

這部分也簡單，begin()/end()分別回傳含有_data/_data+_size 的 iterator，empty()/size()都可用 data member _size 完成。

c. push_back(const T& x)

如果_size ==_capacity，就呼叫 expand()去要記憶體空間，並++_size 和把 _isSorted 設回 false。

Helper Function：

void expand() //要記憶體空間

分成_capacity 是 0 和不是 0 的情況討論，前者要一個記憶體空間後 _capacity 設成一，後者_capacity 乘二，並將之前的資料複製過去再將原本要的空間歸還給系統。

d. pop_front(), pop_back(), clear()

pop_back()和 clear()基本上就是將陣列大小(_size)減一和設為零，只要不要取到資料就好，後面的資料在 push_back 後會被蓋掉。但 pop_front()在 _size>=2 時不能這麼做，要將最後一筆資料換到第一個（不在意資料順序的做法）後才能--_size，在這個情況下因為有動到資料順序，_isSorted 設回 false。

e. erase(iterator pos)

跟 pop_front()類似，在_size>=2 時，將最後一筆資料換到 pos 位置再--_size，_isSorted 設回 false。

f. erase(const T& x), find(const T& x)

前者只要 find 不是回傳 end()，就呼叫 erase(iterator pos)，pos 為 find 回傳的 iterator。至於 find()就是將 0~_size － 1 跑一遍看有沒有跟 x 相等，沒有找到的話就回傳 end()。

g. sort()

為內建的 sort()函式，有增加判斷_isSorted 為 true 時就不 sort()直接 return 和 sort()完後將_isSorted 設為 true 兩行。

2. dlist.h

基本上 DList 的結構重點就只有三個：

_head -> _prev is dummy

dummy -> _next is _head

dummy -> _prev is the last element

前兩個任何情況（list 不管是不是空的）都必成立。

a. DList constructor

一開始_head 是 dummy，故_head -> _prev、_head -> _next 都等於_head。

b. class iterator

overloading++/--時是將_node 指到下/上一個，也就是_node = _node -> _next/_prev。

c. begin(), end(), empty()

因為符合上述的三個原則，begin()為_head，end()為 dummy 即是_head -> _prev，節省走一遍資料的時間。empty()也只要判斷_head -> _prev 是不是等於_head 就可，不用記住_size。

d. size()

既然沒有記住_size，勢必得從 begin 到 end 前一個跑過一遍來計算。

e. push_back(const T& x)

先 new 一個 DListNode<T>*，它的_prev 是 dummy 的_prev，_next 是 dummy。然後我分成兩種討論，如果 list 是空的（增加第一個元素），那麼 _head、dummy 的_prev、dummy 的_next 都要指到它。如果不是空的，那麼要指到它的有 dummy 的_prev -> _next 和 dummy 的_prev。最後將_isSorted 設為 false。接下來的 delete 功能都不用改變順序。

f. pop_front(), pop_back(), clear()

前兩個就是把_node 的連結拆掉再重接，pop_front()要注意_head 的位置會變，跟 array 不一樣，最後要把_node delete。clear()從 begin 到 end 前一個跑一遍 pop_front()即可。

g. erase(iterator pos)

把_node 的連結拆掉再重接即可。

h. erase(const T& x), find(const T& x)

前者只要 find 不是回傳 end()，就呼叫 erase(iterator pos)，pos 為 find 回傳的 iterator。至於 find()就是從 begin 到 end 前一個跑一遍看有沒有跟 x 相等，沒有找到的話就回傳 end()。

i. sort()

一樣先判斷是否_isSorted 來決定要不要 sort()，並且 sort()完後將_isSorted 設為 true。sort()本身我是用 bubble sort。

Helper Function：

void swapData(DListNode<T>* &a, DListNode<T>* &b) const () //交換 data

3. bst.h

BSTreeNodedata member：

```
T                   _data;
BSTreeNode<T>* _left;
BSTreeNode<T>* _right;
BSTreeNode<T>* _parent;
```

BSTree data member：

```
BSTreeNode<T>* _root;
```

size_t _size;

基本規則如下：

_root -> _parent is dummy

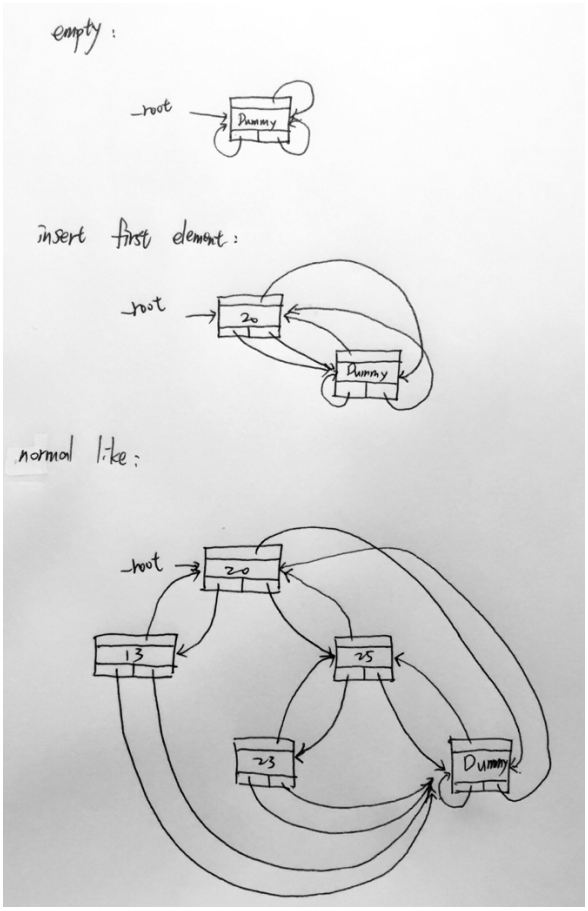
dummy -> _left is dummy

dummy -> _right is _root

dummy -> _parent is biggest element

leaf -> _left & _right are both dummy

架構圖大概長這樣：



但是寫到最後其實覺得沒必要這麼複雜，有試著寫另一版本但因時間來不及而作罷。

a. BSTree constructor

先 new 一個 BSTreeNode 即為 dummy node 亦是_root、_root -> _left、_root -> _right、_root -> _parent 共同所指到的地方，參見上圖 empty 時的樣子。

b. class iterator

++/--是三種資料結構中最為複雜的，因為意思分別是指到下一個大小的 _node，我的做法是舉++為例，大概是先判有沒有 rightchild，有的話就忘 rightchild 過去再往 leftchild 一直走到底（dummy 前）即為下一個大小的 _node，但如果沒有 rightchild，則必須往上走直到自己是別人的 leftchild 才停下。--的做法就相反而已。那因為要判許多東西所以我在 class iterator 的 private 也加了幾個 inline function，事後也覺得有點冗。

Helper Function：

inline bool isDum() //是不是 dummy

inline bool isLDum() //_left 是不是 dummy

inline bool isRDum() //_right 是不是 dummy

inline bool isPDum() //_parent 是不是 dummy

inline bool isLeaf() // isLDum()&&isRDum

inline bool isLChild() //是不是別人的_left

inline bool isRChild() //是不是別人的_right

c. begin(), end(), empty(), size()

begin()就是從_root 一直往左走直到 dummy 的前一個。end()就是回傳 dummy 也就是 iterator(_root -> _parent)。empty()則是檢查_root -> _parent 是不是 _root。size()因為我有記起來所以就直接回傳_size。

d. insert(const T& x)

三種情況：

1. empty：_root、_root -> _parent、_root -> _right 指到它，它自己的三個接點都指到 dummy(Helper Function：toDum(BSTreeNode<T>* & n))。

2. insert 的東西最大：跟下一種不一樣的地方在因為 dummy -> _parent 要指到最大的 node，在 insert 最大的東西時要把 dummy -> _parent 換上去。

3. 其他：insert(_root, x)。

Helper Function：

void toDum(BSTreeNode<T>* & n) // n = _root -> _parent //dummy

void insert(BSTreeNode<T>* node, const T& x) // function overloading

這個 insert 的做法是用 recursive，也就是判斷 x 跟 node -> _data 的大小往左或往右再呼叫 insert，直到到 dummy 前，就 new 一個_data 為 x 的 node

e. pop_front(), pop_back(), clear(), erase(iterator pos)

這些都靠 erase(iterator pos)來達成，而 erase(iterator pos)又是去判斷 leaf case 或 onechild case 或 twochild case 三種人然後分別呼叫 Helper Function 中的 delLeaf(pos)、delOneChild(pos)、delTwoChild(pos)。

Helper Function：

void delLeaf(iterator pos) //_left&&_right 都是 dummy 時

void delOneChild(iterator pos) //_left&&_right 其中之一是 dummy

void delTwoChild(iterator pos) //_left&&_right 都不是 dummy，交換 successor（twochild case 所以一定有），變成 onechild case 或 leaf case（呼叫前二者）。

f. erase(const T& x), find(const T& x)

前者也是靠 find 後 erase(iterator pos)完成，find 的做法是 recursive，寫在 private Helper Function。

Helper Function：

iterator find(const T& x, iterator& pos) //function overloading

recursive 作法，從_root 開始，如果 x == *pos 就回傳 iterator，不然就依大小 find 左右下去直到 dummy 就回傳 end()。

二、 實驗部分

針對不同指令做執行時間上的比較如下：

1. ADTSort

BST 因為 add 完就已經 sort 好，故採用 add 的時間。

a. 10000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.01	1.06	0

b. 30000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.01	9.66	0.01

c. 50000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.02	28.09	0.02

d. 100000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.04	108.5	0.04

e. 200000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.07	436.1	0.1

從上面五個的結果和每種資料結構本身的運作方式可推知：

Array 應為 O(nlogn)（網路上寫的(std::sort())）

DList 應為 O(n^2)（bubble sort）

BST 應為 O(nlogn)（平均情況下, worst case insert n 次應為 n^2）

而 array 和 dlist 用 isSorted 可大幅減少沒改動到資料順序情況下，第二次 sort 的時間。

2. ADTDelete

(1) -a

a. 50000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0	0	0.02

b. 100000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0	0.01	0.03

c. 1000000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0	0.07	0.44

從上面三個的結果和每種資料結構本身的運作方式可推知：

Array 應為 O(1)（_size 設為 0）

DList 應為 O(n)（跑一遍 pop_front()）

BST 應為 O(n)（跑一遍 pop_front()）

(2) -f , -b

a. 50000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
-f 秒數(s)	0	0.01	0.02
-b 秒數(s)	0	0	0.02

b. 100000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
-f 秒數(s)	0	0.02	0.04
-b 秒數(s)	0	0.02	0.02

c. 1000000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
-f 秒數(s)	0.01	0.07	0.43
-b 秒數(s)	0	0.08	0.39

從上面三個的結果和每種資料結構本身的運作方式可推知：

Array 應為 O(n)（pop O(1) n 次每次_size--或交換再_size--(不管順序)所以幾乎是 0）

DList 應為O(n)（pop O(1) n 次）

BST 應為 O(nlogn)（pop O(logn)(找 begin) n 次）

(3) -r

a. 30000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0	0.79	4.01

b. 50000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0	2.58	11.66

c. 100000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.01	9.66	56.68

從上面三個的結果和每種資料結構本身的運作方式可推知：

Array 應為 O(n)（erase(pos) O(1) n 次還是很快）

DList 應為O(n^2)（erase(pos) O(1) n 次，但每次 iterator 要移過去 O(n)）

BST 應為 O(nlogn)（erase(pos) O(1) n 次，但每次 iterator 要移過去 O(n)，但平均是 logn）

(4) -s （用自己寫的 gendo file 去跑多筆資料的結果，不然一次只能手動刪一個）

a. 200000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	9.1	18.78	4.44

從上面的結果和每種資料結構本身的運作方式可推知：

Array 應為 O(n^2 + n)=O(n^2) （find 為 O(n)，find n 次，erase(pos)亦需 O(n)）

DList 應為O(n^2 + n^2))=O(n^2)（find 為 O(n)，find n 次，erase(pos)亦需 O(n^2)）

BST 快很多應為 O(nlogn + nlogn)=O(nlogn)（find 為 O(logn)，find n 次，erase(pos)亦需 O(nlogn)）

這個結果跟 find 有關故不測 ADTQuery。

3. ADTAdd

a. 1000000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.13	0.09	0.98

b. 5000000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	0.84	0.45	7.81

c. 10000000 筆資料

	./adtTest.array	./adtTest.dlist	./adtTest.bst
秒數(s)	1.66	0.9	18.32

從上面三個的結果和每種資料結構本身的運作方式可推知：

Array 應為 O(n)

DList 應為O(n)

BST 應為 O(nlogn)（insert 為 O(n)，但可看成平均 logn，insert n 次）