



聽說要會很
多 喔!!

真討厭!



推廣教育資料結構與演算法

Topic 8 堆積與雜湊

Kuan-Teng Liao (廖冠登)

2021/07/10

大綱

.....

.....

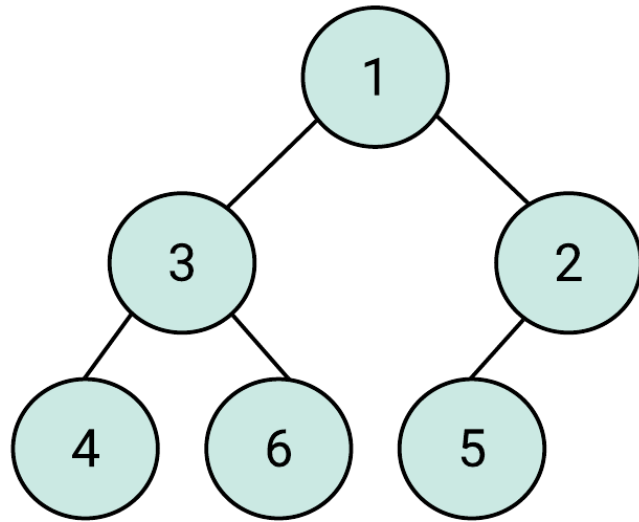
.....

.....

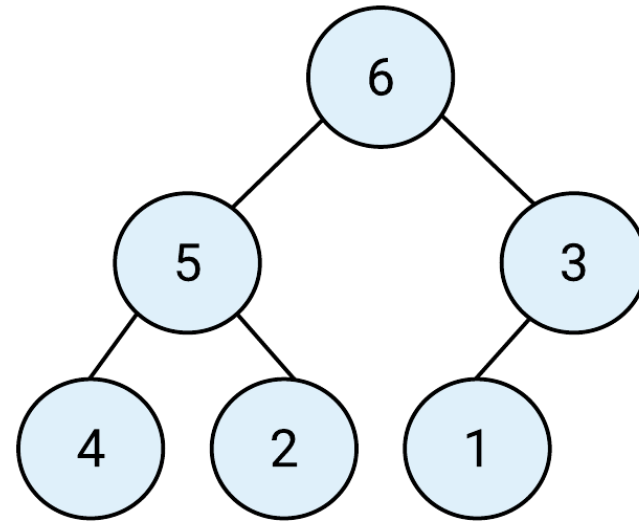
堆積(Heap) (1)

- 是一結構，其所產生的樹會是二元樹(binary tree)，即max degree = 2
 - ✓ 代表一個父結點最多會有兩個子節點
 - 就只有父節點與子節點關聯
 - ✓ 但在堆積內有兩種特別的機制
 - 最大堆積
 - 父節點永遠大於子節點與子樹
 - 最小堆積
 - 父節點永遠小於子節點與子樹

堆積(Heap) (2)



Min heap



Max Heap

堆積(Heap) (3)

- 一般來說建立方法可以用串列法與陣列法

✓注意

- 不管串列或陣列法，於建樹時只會進行放置，不會執行任何的機制
 - 就是於建樹時不會執行最大或最小機制
- 全部將樹建完後，才會執行機制的排序稱為heapsort，並在每回合的heapsort中執行heapify(堆積化)

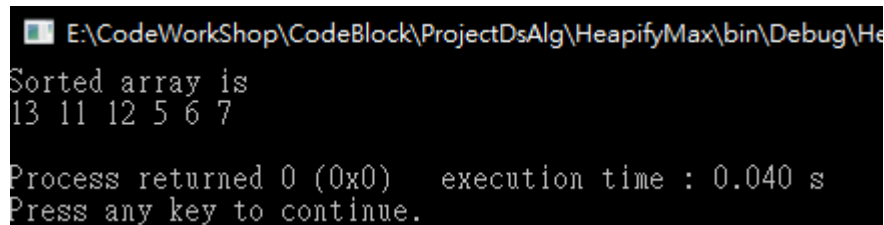
堆積(Heap) (4)

main.cpp

```
1 #include <iostream>
2 #include <utility>
3
4 class Heap{
5     public: void fn_MaxHeapify(int* ip_Arr, int i_Size,
6                               int i_Ind){
7         int i_Par = i_Ind;
8         int i_L = 2 * i_Ind + 1;
9         int i_R = 2 * i_Ind + 2;
10
11         if (i_L < i_Size && ip_Arr[i_L] > ip_Arr[i_Par])
12             i_Par = i_L;
13
14         if (i_R < i_Size && ip_Arr[i_R] > ip_Arr[i_Par])
15             i_Par = i_R;
16
17         if (i_Par != i_Ind) {
18             std::swap(ip_Arr[i_Ind], ip_Arr[i_Par]);
19             fn_MaxHeapify(ip_Arr, i_Size, i_Par);
20         }
21     }
22
23     public: void fn_HeapSort(int* ip_Arr, int i_Size){
24         for (int i_Ct = i_Size / 2 - 1;
25             i_Ct >= 0; i_Ct--){
26
27             fn_MaxHeapify(ip_Arr, i_Size, i_Ct);
28         }
29     }
30     public: void fn_GetResult(int* ip_Arr, int i_Size){
31         for (int i_Ct = 0; i_Ct < i_Size; i_Ct++){
32             std::cout << ip_Arr[i_Ct] << " ";
33         }
34         std::cout << "\n";
35     }
36 };
```

main.cpp

```
38 int main() {
39     int ia_Arr[] = { 12, 11, 13, 5, 6, 7 };
40     int i_Size = sizeof(ia_Arr) / sizeof(int);
41     Heap o_Heap;
42     o_Heap.fn_HeapSort(ia_Arr, i_Size);
43
44     std::cout << "Sorted array is \n";
45     o_Heap.fn_GetResult(ia_Arr, i_Size);
46 }
47 }
```

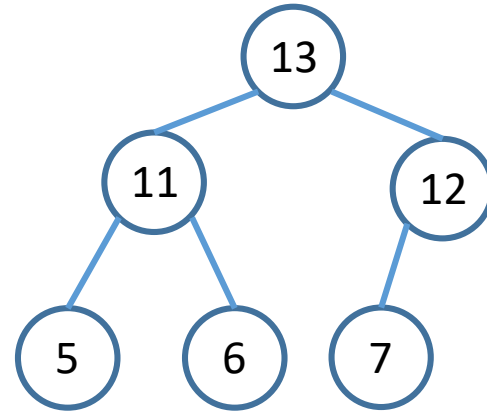
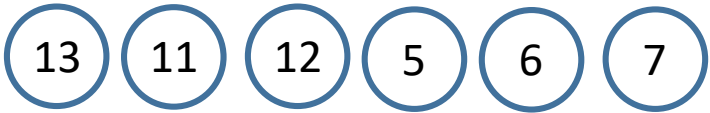


```
E:\CodeWorkShop\CodeBlock\ProjectDsAlg\HeapifyMax\bin\Debug\He
Sorted array is
13 11 12 5 6 7
Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

程式碼網址：<https://github.com/altoliaw2/HeapifyMax>

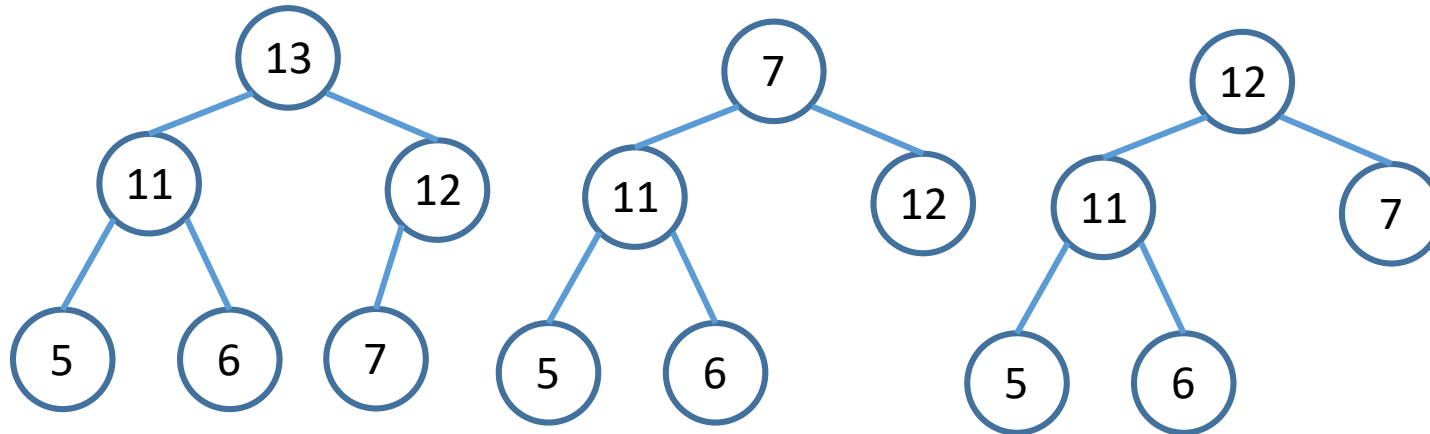
堆積(Heap) (5)

- 根據上頁程式碼可得知，其圖為



堆積(Heap) (6)

- 因此建樹只需以父節點進行比較即可
 - ✓時間複雜度 $O(n) + O(n\log n) = O(n\log n)$
- 刪除最大/小元素所需時間
 - ✓時間複雜度
 - 前者為直接丟出根節點的值，並將陣列中最後一個節點拷貝至根節點後，最後一個節點刪除
 - 後者為需對其進行heapify從根節點至最下面重新所需時間



堆積(Heap) (7)

- 插入一元素所需時間
 - ✓ 時間複雜度 $O(1) + O(\log n) = O(\log n)$
 - 前者為於陣列後面加入一個元素
 - 後者為該元素至根節點的半邊heapify所需時間

堆積(Heap) (8)

- 以此得知，當資料量較大，只需要找出第 i 個大或小的元素時，或是邊插入邊刪除邊找第 i 個，Heap結構會比直接執行完之前排序方法 (e.g., 插入排序、合併排序)再取第 i 個大或小的元素來的快速
- 有同學可能想問
 - ✓ 好像跟優先權佇列(priority queue，請見Topic 2 p. 26)有點像
 - ✓ 恩若是只抓最大最小值，是有點像，但是heap的效能會比priority queue來的好
 - 主要再移除根點時，heap需要 $O(\log(n))$
 - priority queue移除最上面節點時，需要 $O(n)$

堆積(Heap) (9)

- C++ 有沒有heap的容器
 - ✓ 沒有
 - ✓ 他只能vector + make_heap來實作(需#include <algorithm>)
 - 只能每次重新建立heap
 - 沒辦法直接呼叫heapify
 - 通常heapify要自己寫

練習1 (1)

- 給定一個陣列，裡面共 m 個元素，每個元素代表繩索長度，當欲將兩繩索連結時，需耗費兩元素長度之合費用。假設今日每次皆由**最短**兩組進行連結(因為想要耗費最少)，請問合成完整一條需耗費多少錢？

✓輸入共兩列

- 第一列為陣列中有幾個元素，即為值 m
- 第二列為 m 個元素

✓輸出耗費價錢，共一列

練習1 (2)

Sample	Output
4 1 8 7 4	37
4 5 4 2 8	36

練習2 (1)

- 給定一個陣列，裡面共 m 個元素，請對其數值找出中位數
 - ✓輸入共兩列
 - 第一列為陣列中有幾個元素，即為值 m
 - 第二列為 m 個元素
 - ✓輸出請輸出中位數的值

練習2 (2)

Sample	Output
4 1 8 7 4	5.5
5 1 6 3 2 8	3

雜湊(Hash) (1)

- 藉由一組hash function(雜湊函式)將樹入映射對應至一組固定的記憶體，並讓其可以快速找到儲存記憶體空間的索引與實體儲存的方式
 - ✓E.g., 當輸入為一個一個的a-z字元時
 - 其雜湊函式為hashing function, $f(X)$ 可為 $X - 'a'$ ，其中X代表輸入的字元，因此若宣告一陣列為26個空間，則 $f(X)$ 的值，稱為雜湊值(hash value)，將會與陣列某個元素產生關聯，此關聯可稱為雜湊
- 若輸入變得更為複雜時，這樣的函式對應方式將會需要配合極大的空間才能對應，因此常用建立關聯的方法的雜湊函式則讓自行選擇或是預設
 - ✓循環冗餘校驗(Cyclic redundancy check, CRC)
 - ✓核對和(Checksum)等
 - https://en.wikipedia.org/wiki/Hash_function

雜湊(Hash) (2)

✓ 不管是循環冗餘校驗(Cyclic redundancy check, CRC)或是核對和(Checksum)等雜湊方法

- 其hash function即要產生一個獨一無二的雜湊值，使Hash的方法可以使輸入與儲存記憶體彼此映射產生「單向」關聯
 - 這意味著只有拿到最後的雜湊值無法推算出原來的輸入
- 因此透過雜湊函式的特性，可應用於加密學中「非對稱加密」(Asymmetric cryptography)作為使用

• 因此一般來說，使用雜湊函式的input和產生的雜湊值最好的結果必然是要1對1關聯

✓ 所以簡單說雜湊就是一門

- 把蘋果香蕉你個芭樂(input)都丟進去打一打、攪一攪，全部變得爛爛產生「獨一無二」的產物(雜湊值)，以便可以快速「找尋出欲搜尋的資料」

雜湊(Hash) (2)

- 但是在一般使用的雜湊函式結果不會是一對一，因此會產生衝突/碰撞(collision)
 - ✓兩種不同的input會有相同的雜湊值
- 若要使用雜湊技術時，必須讓儲存的空間指向一個串列(Linked list)即可解決同樣輸入卻有不同input問題
 - ✓E.g., 如輸入目前有6筆資料：即Joe:'M', Sue:'F', Dan:'M', Nell:'F', Ally:'F', Bob:'M'
 - 若使用第一個字母設計陣列，再利用每個元素指向一個串列
 - 需要宣告陣列大小為26
 - 若是有很多空間並沒用到
 - 怎麼辦？

雜湊(Hash) (3)

✓E.g., 如輸入目前有6筆資料：即Joe:'M', Sue:'F', Dan:'M', Nell:'F', Ally:'F', Bob:'M' (cont'd)

- 就對每個名字字串對其取出字元的整數值利用雜湊函式得到雜湊值
- 再將雜湊值除以合適的陣列大小(e.g., 假設為5)，就可以生成下面的規律

Name	Hash value	Index
Joe	1846231176	1
Sue	2575795571	1
Dan	1293296397	2
Nell	174981062	2
Ally	2263042623	3
Bob	1961192743	3

0-

1- Sue:F Joe:M

2- Nell:F Dan:M

3- Bob:M Ally:F

4-

雜湊(Hash) (4)

- 所以根據Hash結果可以得知
 - ✓ 若要於hash結構中搜尋某筆資料其時間複雜度
 - 最好 $O(1) = O(c)$
 - 最差 $O(n)$
 - 平均 $O(1) = O(c)$ ，推導為利用平攤分析(amortized analysis)求得
 - 可以想成若有無限大陣列空間(m)，其每個串列個數為(n)， n 遠小於 m ，因此平均時間複雜度為 n/m ，其值為一常數，因此可得 $O(c)$
- 使用Hash的優點，於搜尋資料時只需常數時間就可找到該筆資料
- 會盡量讓每個儲存空間資料平均分配
- 語法
 - ✓ 需引用 `#include <functional>`，並且用 `std::hash<型態>` 物件名稱; 來創立物件

雜湊(Hash) (5)

main.cpp

```
1 #include <algorithm>
2 #include <functional>
3 #include <iostream>
4 #include <string>
5
6 struct Man{
7     public: std::string s_Name;
8     public: char c_Gender;
9     public: Man* op_Next;
10    public: Man(std::string s_Name, char c_Gender){
11        this->s_Name = s_Name;
12        this->c_Gender = c_Gender;
13        op_Next = nullptr;
14    }
15 };
```

main.cpp

```
17 struct Hash{
18     public: int i_Size;
19     public: Man** o2p_ManSet;
20     //COM: Functional
21     public: std::hash<std::string> str_Hash;
22
23     public: Hash(int i_Size){
24         this->i_Size = i_Size;
25         o2p_ManSet = new Man*[i_Size];
26         std::fill(o2p_ManSet, o2p_ManSet+ i_Size, nullptr);
27     }
28     public: ~Hash(){
29         for(int i_Ct=0; i_Ct< i_Size; i_Ct++){
30             if(o2p_ManSet[i_Ct] == nullptr){
31                 continue;
32             }
33
34             for(Man* op_Tmp = o2p_ManSet[i_Ct];
35                 op_Tmp != nullptr;){
36                 Man* op_Tmp2 = op_Tmp->op_Next;
37                 delete op_Tmp;
38                 op_Tmp = op_Tmp2;
39             }
40         }
41         delete [] o2p_ManSet;
42     }
```

雜湊(Hash) (6)

main.cpp

```

44 public: void fn_InsItem(Man* op_Man){
45     int i_Ind = str_Hash(op_Man->s_Name) % i_Size;
46     Man* op_TmpMan = o2p_ManSet[i_Ind];
47     op_Man->op_Next = op_TmpMan;
48     o2p_ManSet[i_Ind] = op_Man;
49 }
50
51 public: void fn_GetRes(){
52     for(int i_Ct=0; i_Ct< i_Size; i_Ct++){
53         std::cout<< i_Ct << "-\t";
54         for(Man* op_Tmp =o2p_ManSet[i_Ct];
55             op_Tmp != nullptr;
56             op_Tmp= op_Tmp->op_Next){
57             std::cout<< op_Tmp->s_Name << " : "
58                 << op_Tmp->c_Gender << " ";
59         }
60         std::cout<< "\n";
61     }
62 }
63
64 public: void fn_ShowTmp(std::string s_Name){
65     std::cout<< str_Hash(s_Name) << "\n";
66 }
67 };

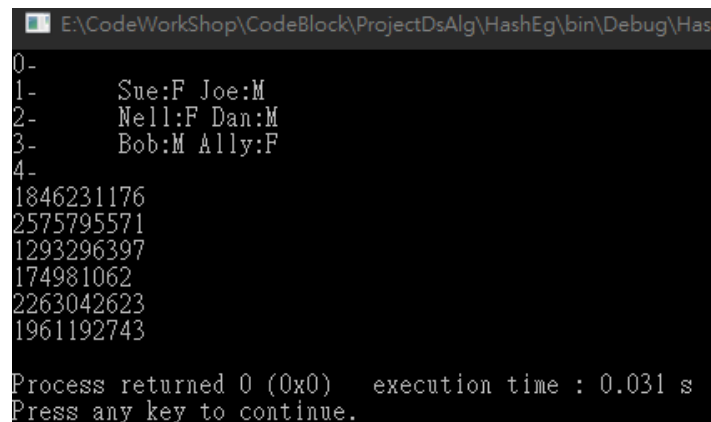
```

main.cpp

```

69 int main(){
70     Hash o_HMap(5);
71     o_HMap.fn_InsItem(new Man("Joe", 'M'));
72     o_HMap.fn_InsItem(new Man("Sue", 'F'));
73     o_HMap.fn_InsItem(new Man("Dan", 'M'));
74     o_HMap.fn_InsItem(new Man("Nell", 'F'));
75     o_HMap.fn_InsItem(new Man("Ally", 'F'));
76     o_HMap.fn_InsItem(new Man("Bob", 'M'));
77
78     o_HMap.fn_GetRes();
79
80     o_HMap.fn_ShowTmp("Joe");
81     o_HMap.fn_ShowTmp("Sue");
82     o_HMap.fn_ShowTmp("Dan");
83     o_HMap.fn_ShowTmp("Nell");
84     o_HMap.fn_ShowTmp("Ally");
85     o_HMap.fn_ShowTmp("Bob");
86     return 0;
87 }

```



```

E:\CodeWorkShop\CodeBlock\ProjectDsAlg\HashEg\bin\Debug\Has
0-
1- Sue:F Joe:M
2- Nell:F Dan:M
3- Bob:M Ally:F
4-
1846231176
2575795571
1293296397
174981062
2263042623
1961192743

Process returned 0 (0x0) execution time : 0.031 s
Press any key to continue.

```


雜湊(Hash) (7)

- 好像很難寫?有沒有現成的Hash可以用
 - ✓一樣手刻，但是陣列改用map
 - 稍微好寫一點
 - ✓hash_map
 - 但是他並非標準函式庫，所以標準考試沒法用
 - ✓其他方法
 - 請參考unordered_map
 - https://www.cplusplus.com/reference/unordered_map/unordered_map/at/
 - 與容器中的Map很像，但是unordered代表該容器不會排序
 - 只需要把#include <map> 改成 #include <unordered_map>即可

映射(Map)(1)

- 與陣列和向量相似，也是儲存同一型態類別資料的集合
 - ✓ 因此和陣列相同，可單獨宣告與個別定義，也可同時宣告與定義
- 擺脫用「含零的正整數」，作為索引，作為取代，用主鍵(key)當作索引，因此一個元素包含了主鍵+值
 - ✓ 主鍵：作為映射內作為鑑別整個元素的「**唯一值**」，可以於宣告時定義其形態(系統型態或是自訂義型態)
 - ✓ 值：即為原本要放入映射裡的值。

映射(Map)(2)

- 與陣列相比最主要優點
 - ✓ 可彈性調整集合元素個數(意味著，可不必於一開始決定映射裡面元素個數)
 - 由於陣列只能使用固定大小，若要變更陣列大小只能重新新創一大/小陣列在將其拷貝進新陣列中
 - ✓ 用主鍵(key)當作索引，且會於插入元素時，會透過主鍵於映射中將元素自行排序(升冪排序)
- 與陣列相比之缺點
 - ✓ 若儲存皆屬於同樣型態，映射耗費較大記憶體空間與時間

映射(Map)(2)

- 與向量相比最主要優點
 - ✓ 用主鍵(key)當作索引，且會於插入元素時，會透過主鍵於映射中將元素自行排序(升冪排序)
- 與向量相比之缺點
 - ✓ 插入元素會自動排序，因此較向量耗費時間

映射(Map)(2)

- 宣告注意事項

- ✓ 需引用 `#include <map>` 函式庫

- ✓ 語法

- `std::map <主鍵型態, 值之型態>` 物件名稱;

main.cpp

```
1 #include <iostream>
2 #include <map>
3
4 int main(){
5     std::map<char, int> o_NumSet;
6
7     return 0;
8 }
```

映射(Map)(3)

- 定義注意事項

- ✓ 語法

- 物件名稱.insert(std::pair<主鍵型態, 值之型態>(主鍵的值, 值));

main.cpp

```
1 #include <iostream>
2 #include <map>
3
4 int main(){
5     std::map<char, int> o_NumSet;
6
7     for(int i_C=0; i_C < 10; i_C++){
8         int i_Ascii= static_cast<int>('A') + i_C;
9         char c_Ascii= static_cast<int>(i_Ascii);
10        o_NumSet.insert(std::pair<char, int>(c_Ascii, i_Ascii));
11    }
12
13    return 0;
14 }
```

映射(Map)(4)

- 常見用法，物件後加上以下方法，即可達到以下效果
 - ✓ `.insert(std::pair<主鍵型態,值型態>(主鍵,值))` - 加入一筆元素於映射中
 - ✓ `.empty()` - 判斷映射內是否為空
 - ✓ `.size()` - 判斷映射內元素的個數
 - ✓ `.find(主鍵)` - 於映射中找出是否含有主鍵的元素，有的話回傳該元素的位址；若無則會回傳映射內最後一個元素的結束位址 (即`.end()`)
 - ✓ `.erase(主鍵)` - 將符合主鍵的元素於映射中刪除
 - ✓ `.clear()` - 清空所有元素
 - ✓ `.begin()` - 回傳映射內第一個元素的起始位址
 - ✓ `.end()` - 回傳映射內最後一個元素的結束位址(即最後一個元素之欲加入下一個元素的起始位址)
- 範例請見[p18](#)

映射(Map)(5)

main.cpp

```

1 #include <iostream>
2 #include <map>
3
4 int main(){
5     std::map<char, int> o_NumSet;
6
7     for(int i_C=0; i_C < 10; i_C++){
8         //Key A-J and theirs decimal values
9         int i_Ascii= static_cast<int>('A') + i_C;
10        char c_Ascii= static_cast<char>(i_Ascii);
11        o_NumSet.insert(std::pair<char, int>(c_Ascii, i_Ascii));
12    }
13
14    std::cout<< o_NumSet.size()<< ", " << o_NumSet.empty() << "\n";
15
16    for(std::map<char, int>::iterator o_1L= o_NumSet.begin();
17        o_1L!= o_NumSet.end();
18        o_1L++){
19        //(*o_1L) is an element and a pair
20        // For obtaining the key, using the term "first";
21        // for obtaining the value using the term "second"
22        std::cout<< (*o_1L).first << ", " << (*o_1L).second << "\n";
23    }

```

main.cpp (conti')

```

24
25 std::map<char, int>::iterator o_1L = o_NumSet.find('C');
26 if(o_1L != o_NumSet.end()){
27     std::cout<< (*o_1L).first << ", " << (*o_1L).second << "\n";
28     o_NumSet.erase(o_1L);
29 }
30 else{
31     std::cout<< "No element with the key—" << 'C' << "\n";
32 }
33
34 o_1L = o_NumSet.find('C');
35 if(o_1L != o_NumSet.end()){
36     std::cout<< (*o_1L).first << ", " << (*o_1L).second << "\n";
37 }
38 else{
39     std::cout<< "No element with the key—" << 'C' << "\n";
40 }
41
42 o_NumSet.clear();
43 std::cout<< o_NumSet.size()<< ", " << o_NumSet.empty() << "\n";
44
45 return 0;
46 }

```

練習3 (1)

- 給定一陣列，裡面共有M個元素。裡面任兩個元素是否有等於和 *sum*
 - ✓輸入，共兩列
 - 第一列為輸入值M 與 *sum* (該兩個值中間有空白)
 - 第二列為陣列中M個值，中間將藉由空白隔開
 - ✓輸出
 - 印出所有符合的一對，該一對需照陣列中元素的順序

練習3 (2)

Sample	Output
6 10 8 2 3 5 7 1	(8, 2) (3, 7)
6 10 8 2 3 5 7 3	(8, 2) (3, 7) (7, 3)