目錄

03 複合式的資料結構之原理與實作

04 重要演算法之原理與應用

05 程式碼的除錯和執行時間

06 APCS 相關考題的實作與解析

03

CHAPTER

複合式的資料結構之原理與實作

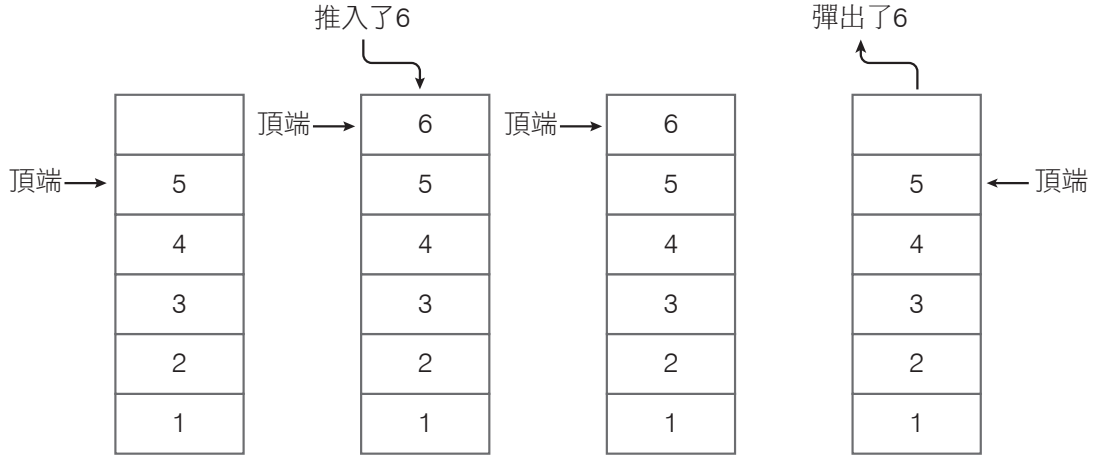
本章學習重點

- 3.1 堆疊
- 3.2 佇列
- 3.3 鏈結串列

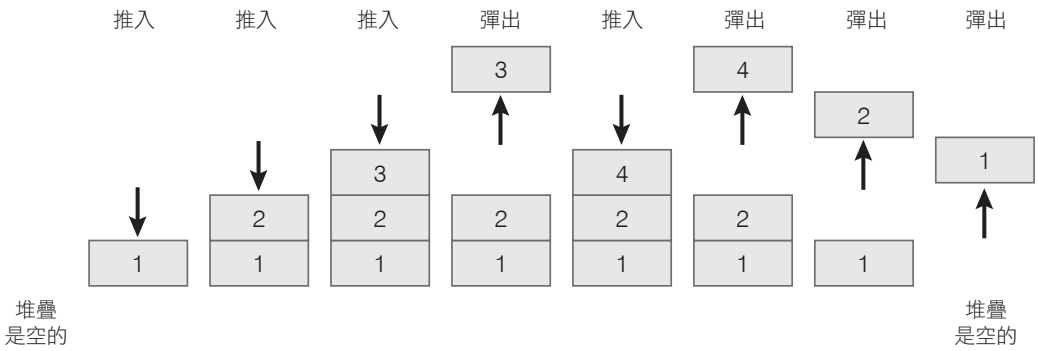
3.1 堆疊

在電腦科學的領域裡，堆疊 (stack) 在抽象概念上，猶如是縱向之連續的資料結構，並且具有以下特性：

- 目前之最後一個資料所在的位置，可稱為頂端 (top)。
- 資料的存取 (推入與彈出)，皆在同一個頂端出入口。
  - 推入 (push) 的動作，即是指存放的意義。
  - 彈出 (pop) 的動作，即是指取出的意義。
- 最後被推入的資料，則可最先被彈出！換言之，最先被推入的資料，最後才能被彈出。
  - 此特性可稱為後進先出 (LIFO: last in, first out)，或是先進後出 (FILO: first in, last out)。



▲ 堆疊頂端 (top) 的變化



▲ 堆疊資料的推入與彈出之順序

在 Python 與 C++ 程式語言當中，對於堆疊資料結構的支援，早就已經是預設內建的！不過，為了讓兩種程式語言之範例程式碼的實作方式儘量相似，筆者在本節，即將改寫兩種程式語言之相關內建的部分語法，以利讀者們的理解。

從如下範例，可看出在 Python 程式語言中，利用串列資料型態的變數，模擬出堆疊資料結構，並且利用自訂函數的定義，以模擬出簡易存取堆疊內部資料的動作。

範例：ch03-01-xx-01.py

```
01  color_stack = ['orange', 'red', 'green', 'blue', 'yellow', 'violet', 'indigo']
02
03  def push(data): color_stack.append(data)
04  def pop(): return color_stack.pop()
05  def top(): return color_stack[-1]
06  def size(): return len(color_stack)
07  def empty(): return len(color_stack) == 0
08
09  temp = pop()
10  print(temp)
11
12  temp = pop()
13  print(temp)
14
15  push('gold')
```

```
16  push('pink')
17
18  temp = top()
19  print(temp)
20
21  check = empty()
22  print(check)
23
24  temp = size()
25  print(temp)
```

輸出結果

```
indigo
violet
pink
False
7
```

說明

- 列 01 的語法，定義了變數 color_stack，並設定其初始資料為串列常數 ['orange', 'red', 'green', 'blue', 'yellow', 'violet', 'indigo']。
- 列 03 的語法，定義了新函數 push，並傳入「內含欲被推入堆疊的資料」之參數 data。
  - 因為變數 color_stack 目前的內含資料，係為一個串列！所以支援「.append()」的函數語法。
  - 「color_stack.append(data)」可將參數 data 的內含資料，新增 (append) 至變數 color_stack 所內含之串列的尾端，也就是 'indigo' 所在的那一端。本範例在此：
    - 透過串列，來模擬出堆疊的資料結構。
    - 串列的尾端，即用來模擬出堆疊的頂端。

- 列 04 的語法，定義了新函數 pop。
  - 因為變數 color_stack 目前的內含資料，係為一個串列！所以亦支援「.pop()」的函數語法。
  - 「color_stack.pop()」可用來彈出變數 color_stack 所內含之串列的尾端資料。
  - 「return color_stack.pop()」則會將被彈出的資料，傳回到執行函數 pop 的位置，例如：列 09 或列 12 的位置。
- 列 05 的語法，定義了新函數 top。
  - 因為變數 color_stack 目前的內含資料，係為一個串列！所以支援「color_stack[-1]」的索引語法。
  - 「color_stack[-1]」並不是彈出，而是僅僅傳回變數 color_stack 所內含之串列的尾端資料。換言之，目前在串列裡的尾端資料，並沒有被彈出！
  - 「return color_stack[-1]」則會將串列裡的尾端資料，傳回到執行函數 top 的位置，例如：列 18 的位置。
- 列 06 的語法，定義了新函數 size。
  - 因為變數 color_stack 目前的內含資料，係為可迭代的資料結構，例如：串列、值組、集合、字典等等；所以，「len(color_stack)」會傳回變數 color_stack 內含之串列裡的資料個數。
  - 「return len(color_stack)」則會將串列裡的資料個數，傳回到執行函數 size 的位置，例如：列 24 的位置。
- 列 07 的語法，定義了新函數 empty。
  - 「len(color_stack) == 0」是用來判斷變數 color_stack 所內含之串列裡的資料個數，是否為 0 ？！
    - 倘若是的話，則傳回 True (代表成立)，意謂著該串列目前是空的 (empty)，不存在任何資料；反之，則傳回 False (代表不成立)。

- 「return len(color_stack) == 0」則會將意謂著串列是否為「空的」之 True 或 False，傳回到執行函數 empty 的位置，例如：列 21 的位置。

- 列 09 的語法，先是定義了變數 temp；然後，列 09 與列 12 的語法，前後個別執行了函數 pop，並在「變數 color_stack 內含之串列」所模擬出來的堆疊裡，先後彈出並傳回其頂端的兩個資料 'indigo'、'violet'，成為了變數 temp 在不同時間點上的內含資料。
- 列 18 的語法，執行了函數 top，並在「變數 color_stack 內含之串列」所模擬出來的堆疊裡，傳回其頂端的資料，成為了變數 temp 在另一個時間點上的內含資料。
- 列 21 的語法，執行了函數 empty，並在「變數 color_stack 內含之串列」所模擬出來的堆疊裡，判斷是否已經不存在任何資料了？！最後傳回 True (代表成立) 或 False (代表不成立)。
- 列 24 的語法，執行了函數 size，並在「變數 color_stack 內含之串列」所模擬出來的堆疊裡，計算並傳回其資料個數，成為了變數 temp 在最後一個時間點上的內含資料。
  - 在此，函數 size 傳回了7，意謂著該堆疊內，目前尚有 7 個資料。

從如下範例，可看出在 C++ 程式語言中，利用其支援的堆疊資料**型態**，直接模擬出堆疊資料**結構**。並且利用自訂函數的定義，以模擬出簡易存取堆疊內部資料的動作。

範例：ch03-01-xx-02.cpp

```
01  #include <iostream>
02  #include <stack>
03
04  using namespace std;
05
06  stack<string> color_stack;
07
08  void initialize()
```

```
{
  for (string data: {"orange", "red", "green", "blue", "yellow", "violet",
   "indigo"})
  {
    color_stack.push(data);
  }
}

void push(string data) { color_stack.push(data); }

string pop()
{
  string data = color_stack.top();

  color_stack.pop();

  return data;
}

string top() { return color_stack.top(); }

int size() { return color_stack.size(); }

bool empty() { return color_stack.empty(); }

int main(void)
{
  initialize();

  cout << pop() << "\n" << pop() << "\n";

  push("gold");
  push("pink");

  cout << top() << "\n" << empty() << "\n" << size() << endl;

  return 0;
```

```
45  }
```

輸出結果

```
indigo
violet
pink
0
7
```

說明

- 列 02 的語法，載入了和類別 stack 相關的資源庫，以便如下語法可以被利用：
  - 列 06 之代表類別 stack 的關鍵字 stack。
  - 列 12 與列 16 之用來推入資料至堆疊裡的函數「.push」。
  - 列 20 與列 27 之用來僅僅「傳回」堆疊之頂端資料的函數「.top」。
  - 列 22 之用來「彈出」堆疊之頂端資料的函數「.pop」。
  - 列 29 之用來傳回堆疊的資料個數的函數「.size」。
  - 列 31 之用來判斷「在堆疊裡，是否已經不存在任何資料」的函數「.empty」，並傳回 1 (代表成立) 或 0 (代表不成立)。
- 列 06 的語法，定義了堆疊 (stack) 資料型態的變數 color_stack，並且只能內含字串 (string) 資料。
- 列 08 至列 14 的語法，定義了函數 initialize，以便一開始在變數 color_stack 所內含之空的堆疊裡，逐一推入 7 個字串資料。
  - 其中，列 10 的語法，係為相當特別的迴圈敘述！請留意其語法的細節。
- 列 16 的語法，定義了新函數 push，並傳入「內含欲被推入堆疊的資料」之參數 data。

- 因為 color_stack 被定義成為堆疊資料型態的變數！所以支援「.push()」的函數語法。
- 「color_stack.push(data)」可將參數 data 的內含資料，推入至變數 color_stack 所內含之堆疊的頂端，也就是 "indigo" 所在的那一端。本範例在此：
  - 改寫了 C++ 核心系統原本所支援之堆疊的資料結構和函數，進而成為了執行較為簡便的新函數。

◆ 列 18 至列 25 的語法，定義了新函數 pop。

- 因為 color_stack 被定義成為堆疊資料型態的變數！所以亦支援「.top()」與「.pop()」的函數語法。
- 在列 20 的語法裡，先是定義了區域變數 data；然後，「color_stack.top()」可用來僅僅傳回變數 color_stack 所內含之堆疊的頂端資料，並且成為了區域變數 data 的初始資料。
- 列 22 的語法，「color_stack.pop()」，可用來彈出並刪除變數 color_stack 所內含之堆疊的頂端資料。
- 列 24 的語法「return data」則會將區域變數 data 所內含的資料，傳回到執行函數 pop 的位置，例如：列 37 的位置。

◆ 列 27 的語法，定義了新函數 top。

- 因為 color_stack 被定義成為堆疊資料型態的變數！所以支援「color_stack.top()」的函數語法。
- 「color_stack.top()」並不是取出，而是僅僅傳回變數 color_stack 所內含之堆疊的頂端資料。換言之，目前在堆疊裡的頂端資料，並沒有被取出！
- 「return color_stack.top()」會將堆疊的頂端資料，傳回到執行函數 top 的位置，例如：列 42 的位置。

◆ 列 29 的語法，定義了新函數 size。

- 因為 color_stack 被定義成為堆疊資料型態的變數！所以支援「.size()」的函數語法。
- 「color_stack.size()」會傳回變數 color_stack 內含之堆疊裡的資料個數。
- 「return color_stack.size()」會將堆疊的資料個數，傳回到執行函數 size 的位置，例如：列 42 的位置。

◆ 列 31 的語法，定義了新函數 empty。

- 因為 color_stack 被定義成為堆疊資料型態的變數！所以支援「.empty()」的函數語法。
- 「color_stack.empty()」是用來判斷變數 color_stack 所內含之堆疊裡，是否已經不存在任何資料了？！並傳回 1 (代表成立) 或 0 (代表不成立)。
  - 倘若是的話，則傳回 1 (代表成立)，意謂著該串列目前是空的 (empty)，不存在任何資料；反之，則傳回 0 (代表不成立)。
- 「return color_stack.empty()」會將意謂著串列是否為「空的」之 1 或 0，傳回到執行函數 empty 的位置，例如：列 42 的位置。

◆ 列 35 的語法，執行了函數 initialize，使得變數 color_stack 的內含資料，成為了 {"orange", "red", "green", "blue", "yellow", "violet", "indigo"}。

◆ 列 37 的語法，前後個別執行了函數 pop，並在變數 color_stack 所內含的堆疊裡，先後彈出並傳回其頂端的兩個資料 "indigo" 與 "violet"，然後顯示在畫面上。

◆ 列 39 與列 40 的語法，前後個別執行了函數 push，進而在變數 color_stack 所內含之堆疊的頂端，先後推入了兩個資料 "gold" 與 "pink"。

◆ 列 42 的語法，前後執行了函數 top、empty 與 size，並個別傳回相關資料，然後顯示在畫面上。

- 「top()」傳回了變數 color_stack 所內含之堆疊裡的頂端資料。
- 「empty()」傳回了「意謂著在變數 color_stack 所內含之堆疊裡，是否已經不存在任何資料」的 1 (代表成立) 或 0 (代表不成立)。

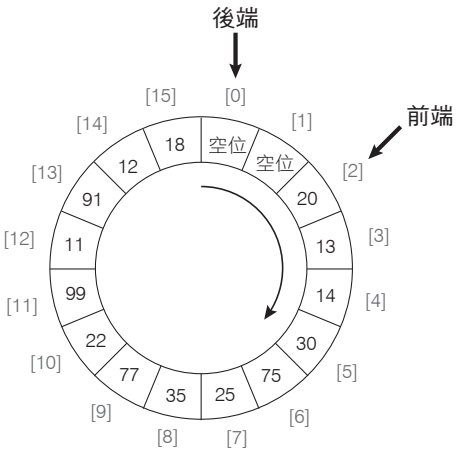
- 「size()」傳回了變數 color_stack 所內含之堆疊裡的資料個數。在此，其傳回了 7，意謂著該堆疊內，目前尚有 7 個資料。

3.2 佇列

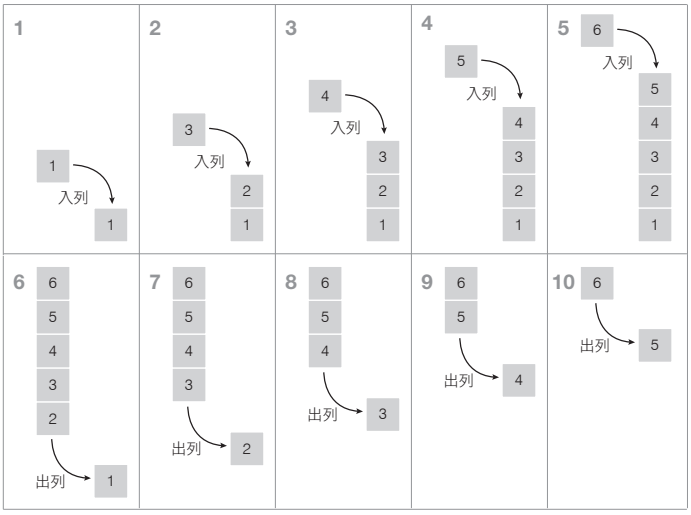
在電腦科學的領域裡，佇列 (queue) 主要分為單端線性佇列 (single-ended linear queue)、雙端線性佇列 (deque, double-ended linear queue) 與環狀佇列 (circular queue) 之連續的資料結構，並且具有以下特性：

- 放入資料的位置，也就是目前之最新一個資料所在的位置，可稱為後端 (back / rear / tail end)。
- 取出資料的位置，也就是目前之最舊一個資料所在的位置，可稱為前端 (front / head end)。
- 先被放入 / 入列 (enqueue) 的資料，也會先被取出 / 出列 (dequeue)。
  - 請特別留意，dequeue (出列) 和 deque (雙端線性佇列) 是看起來相似，但是不同的英文字！
  - 此特性可稱為先進先出 (FIFO: first in, first out)。

▲ 資料被放入單端線性佇列的順序



▲ 內含各個整數資料的環狀佇列



▲ 單端線性佇列的資料之入列與出列的順序

在 Python 與 C++ 程式語言當中，對於「線性」佇列資料結構的支援，早就已經是預設內建的！不過，為了讓兩種程式語言之範例程式碼的實作方式儘量相似，筆者在本節，即將改寫兩種程式語言之相關內建的部分語法，以利讀者們的理解。

從如下範例，可看出在 Python 程式語言中，利用串列資料型態的變數，模擬出佇列資料結構。並且利用自訂函數的定義，以模擬出簡易存取佇列內部資料的動作。

範例：ch03-02-xx-01.py

```
01  orders = ['筆記型電腦', '桌上型電腦', '智慧型手機', '平板電腦', '機械式鍵盤']
02
03  def enqueue(data): orders.append(data)
04  def dequeue(): return orders.pop(0)
05  def front(): return orders[0]
06  def back(): return orders[-1]
07  def size(): return len(orders)
08  def empty(): return len(orders) == 0
09
10  enqueue('無線滑鼠')
11
12  temp = back()
13  print(temp)
14
15  temp = front()
16  print(temp)
17
18  print(orders)
19
20  temp = dequeue()
21  print(temp)
22  print(orders)
23
24  temp = size()
25  print(temp)
26
27  temp = empty()
28  print(temp)
```

輸出結果

```
無線滑鼠
筆記型電腦
['筆記型電腦', '桌上型電腦', '智慧型手機', '平板電腦', '機械式鍵盤', '無線滑鼠']
筆記型電腦
['桌上型電腦', '智慧型手機', '平板電腦', '機械式鍵盤', '無線滑鼠']
5
False
```

說明

- 列 01 的語法，定義了變數 orders，並設定其初始資料為串列常數['筆記型電腦', '桌上型電腦', '智慧型手機', '平板電腦', '機械式鍵盤']。
- 列 03 的語法，定義了新函數 enqueue，並傳入「內含欲被放入單端線性佇列的資料」之參數 data。
  - 因為變數 orders 目前的內含資料，係為一個串列！所以支援「.append()」的函數語法。
  - 「orders.append(data)」可將參數 data 的內含資料，新增 (append) 至變數 orders 所內含之串列的尾端，也就是 '機械式鍵盤' 所在的那一端。本範例在此：
    - 透過串列，來模擬出單端線性佇列的資料結構。
    - 串列的尾端，即用來模擬出單端線性佇列的後端。
- 列 04 的語法，定義了新函數 dequeue。
  - 因為變數 orders 目前的內含資料，係為一個串列！所以亦支援「.pop()」的函數語法。
  - 「orders.pop(0)」可用來取出變數 orders 所內含之串列的第 1 個資料。
  - 「return orders.pop(0)」會將已經被取出的該第 1 個資料，傳回到執行函數 dequeue 的位置，例如：列 20 的位置。
- 列 05 的語法，定義了新函數 front。
  - 因為變數 orders 目前的內含資料，係為一個串列！所以支援「orders[0]」的索引語法。
  - 「orders[0]」並不是取出，而是僅僅傳回變數 orders 所內含之串列的第 1 個資料。換言之，目前在單端線性佇列裡的第 1 個資料，並沒有被出列！
  - 「return orders[0]」會傳回該第 1 個資料，到執行函數 front 的位置，例如：列 15 的位置。

- 列 06 的語法，定義了新函數 back。
  - 因為變數 orders 目前的內含資料，係為一個串列！所以亦支援「orders[-1]」的索引語法。
  - 「orders[-1]」並不是取出，而是僅僅傳回變數 orders 所內含之串列的最後一個資料。換言之，目前在單端線性佇列裡的最後一個資料，並沒有被出列！
  - 「return orders[-1]」會傳回該最後一個資料，到執行函數 back 的位置，例如：列 12 的位置。
- 列 07 的語法，定義了新函數 size。
  - 因為變數 orders 目前的內含資料，係為可迭代的資料結構，例如：串列、值組、集合、字典等等；所以，「len(orders)」會傳回變數 orders 內含之串列裡的資料個數。
  - 「return len(orders)」會傳回該資料個數，到執行函數 size 的位置，例如：列 24 的位置。
- 列 08 的語法，定義了新函數 empty。
  - 「len(orders) == 0」是用來判斷變數 orders 所內含之串列裡的資料個數，是否為 0 ?!
    - 倘若是的話，則傳回 True (代表成立)，意謂著該串列目前是空的 (empty)，不存在任何資料；反之，則傳回 False (代表不成立)。
  - 「return len(orders) == 0」則會將意謂著串列是否為「空的」之 True 或 False，傳回到執行函數 empty 的位置，例如：列 27 的位置。
- 列 10 的語法，執行了函數 enqueue，並在「變數 orders 內含之串列」所模擬出來的單端線性佇列裡，放入資料 '筆記型電腦' 至其後端。
- 列 12 的語法，先是定義了變數 temp，再執行了函數 back，並在「變數 orders 內含之串列」所模擬出來的單端線性佇列裡，先後僅僅傳回其最後一個資料 '機械式鍵盤'，成為了變數 temp 在第 1 個時間點上的內含資料。

- 列 15 的語法，執行了函數 front，並在「變數 orders 內含之串列」所模擬出來的單端線性佇列裡，先後僅僅傳回其第 1 個資料 '筆記型電腦'，成為了變數 temp 在第 2 個時間點上的內含資料。
- 列 18 的語法，可顯示出變數 orders 目前內含之單端線性佇列的資料結構 ['筆記型電腦', '桌上型電腦', '智慧型手機', '平板電腦', '機械式鍵盤', '無線滑鼠']。
- 列 20 的語法，執行了函數 dequeue，並在「變數 orders 內含之串列」所模擬出來的單端線性佇列裡，取出並傳回其第 1 個資料 '筆記型電腦'，成為了變數 temp 在第 3 個時間點上的內含資料。
- 列 22 的語法，可二度顯示出變數 orders 目前內含之單端線性佇列的資料結構 ['桌上型電腦', '智慧型手機', '平板電腦', '機械式鍵盤', '無線滑鼠']。
  - 此時可發現，該資料結構當中，原本位於前端的資料 '筆記型電腦'，已經消失了！
- 列 24 的語法，執行了函數 size，並在「變數 orders 內含之串列」所模擬出來的單端線性佇列裡，計算並傳回其資料個數 5，成為了變數 temp 在第 4 個時間點上之內含資料。
- 列 27 的語法，執行了函數 empty，並在「變數 orders 內含之串列」所模擬出來的單端線性佇列裡，判斷是否已經不存在任何資料了？！最後傳回 True (代表成立) 或 False (代表不成立)，成為了變數 temp 在最後一個時間點上的內含資料。

從如下範例，可看出在 C++ 程式語言中，利用其支援的佇列資料型態，直接模擬出佇列資料結構。並且利用自訂函數的定義，以模擬出簡易存取佇列內部資料的動作。

範例：ch03-02-xx-02.cpp

```
01  #include <iostream>
02  #include <queue>
03
```

```
04  using namespace std;
05
06  queue<string> orders;
07
08  void initialize()
09  {
10    for (string data: {"筆記型電腦", "桌上型電腦", "智慧型手機", "平板電腦",
       "機械式鍵盤"})
11    {
12      orders.push(data);
13    }
14  }
15
16  void enqueue(string data){ orders.push(data); }
17
18  string dequeue()
19  {
20    string data = orders.front();
21
22    orders.pop();
23
24    return data;
25  }
26
27  string front() { return orders.front(); }
28
29  string back() { return orders.back(); }
30
31  int size() { return orders.size(); }
32
33  bool empty() { return orders.empty(); }
34
35  int main(void)
36  {
37    initialize();
38
39    enqueue("無線滑鼠");
```

```
40
41      cout << back() << "\n" << front() << "\n";
42
43      cout << dequeue() << "\n" << size() << "\n" << empty() << endl;
44
45      return 0;
46  }
```

輸出結果

```
無線滑鼠
筆記型電腦
筆記型電腦
5
0
```

說明

- 列 02 的語法，載入了和類別 queue 之相關的資源庫，以便如下的語法可以被利用：
  - 列 06 之代表類別 queue 的關鍵字 queue。
  - 列 12 與列 16 之用來推入資料至單端線性佇列裡的函數「.push」。
  - 列 20 與列 27 之用來僅僅「傳回」單端線性佇列之第 1 個資料的函數「.front」。
  - 列 22 之用來「彈出」單端線性佇列之第 1 個資料的函數「.pop」。
  - 列 29 之用來僅僅「傳回」單端線性佇列之最後一個資料的函數「.back」。
  - 列 31 之用來傳回單端線性佇列的資料個數之函數「.size」。
  - 列 33 之用來判斷「在單端線性佇列裡，是否已經不存在任何資料」的函數「.empty」，並傳回 1 (代表成立) 或 0 (代表不成立)。
- 列 06 的語法，定義了單端線性佇列 (queue) 資料型態的變數 orders，並且只能內含字串 (string) 資料。

- 列 08 至列 14 的語法，定義了函數 initialize，以便一開始在變數 orders 所內含之空的單端線性佇列裡，逐一推入數個字串資料。
  - 其中，列 10 的語法，係為相當特別的迴圈敘述！請留意其語法的細節。
- 列 16 的語法，定義了新函數 enqueue，並傳入「內含欲被推入單端線性佇列的資料」之參數 data。
  - 因為 orders 被定義成為單端線性佇列資料型態的變數！所以支援「.push()」的函數語法。
  - 「orders.push(data)」可將參數 data 的內含資料，推入至變數 orders 所內含之單端線性佇列的後端，也就是 "機械式鍵盤" 所在的那一端。
  - 本範例在此，改寫了 C++ 核心系統原本所支援之單端線性佇列的資料結構和函數，進而成為了執行較為簡便的新函數。
- 列 18 至列 25 的語法，定義了新函數 dequeue。
  - 因為 orders 被定義成為單端線性佇列資料型態的變數！所以亦支援「.front()」與「.pop()」的函數語法。
  - 在列 20 的語法裡，先是定義了區域變數 data；然後，「orders.front()」係為用來僅僅傳回變數 orders 所內含之單端線性佇列的第 1 個資料，並且成為了區域變數 data 的初始資料。
  - 列 22 的語法，「orders.pop()」，可用來彈出並刪除變數 orders 所內含之單端線性佇列的第 1 個資料。
  - 列 24 的語法「return data」則會將區域變數 data 所內含的資料，傳回到執行函數 dequeue 的位置，例如：列 43 的位置。
- 列 27 的語法，定義了新函數 front。
  - 因為 orders 被定義成為單端線性佇列資料型態的變數！所以支援「.front()」的函數語法。

- 「orders.front()」並不是彈出，而是僅僅傳回變數 orders 所內含之單端線性佇列的第 1 個資料。換言之，目前在單端線性佇列裡的第 1 個資料，並沒有被彈出！
- 「return orders.front()」會將單端線性佇列的第 1 個資料，傳回到執行函數 front 的位置，例如：列 41 的位置。

◆ 列 31 的語法，定義了新函數 size。

- 因為 orders 被定義成為單端線性佇列資料型態的變數！所以支援「orders.size()」的函數語法。
- 「orders.size()」會傳回變數 orders 內含之單端線性佇列裡的資料個數。
- 「return orders.size()」會將單端線性佇列的資料個數，傳回到執行函數 size 的位置，例如：列 43 的位置。

◆ 列 33 的語法，定義了新函數 empty。

- 因為 orders 被定義成為單端線性佇列資料型態的變數！所以支援「.empty()」的函數語法。
- 「orders.empty()」是用來判斷變數 orders 所內含之單端線性佇列裡，是否已經不存在任何資料了？！並傳回 1 (代表成立) 或 0 (代表不成立)。
  - 倘若是的話，則傳回 1 (代表成立)，意謂著該串列目前是空的 (empty)，不存在任何資料；反之，則傳回 0 (代表不成立)。
- 「return orders.empty()」會將意謂著串列是否為「空的」之 1 或 0，傳回到執行函數 empty 的位置，例如：列 43 的位置。

◆ 列 37 的語法，執行了函數 initialize，使得變數 orders 的內含資料，成為了 {"筆記型電腦", "桌上型電腦", "智慧型手機", "平板電腦", "機械式鍵盤"}。

◆ 列 39 的語法，執行了函數 enqueue，使得字串資料 "無線滑鼠"，被入列變數 orders 所內含之單端線性佇列的後端，進而成為了最後一個資料。

- 在列 41 的語法裡：
  - 先是執行了函數 back，並在變數 orders 所內含的單端線性佇列裡，僅僅傳回其最後一個資料 "無線滑鼠"，並顯示在畫面上。
  - 接著執行了函數 front，並在變數 orders 所內含的單端線性佇列裡，僅僅傳回其第 1 個資料 "筆記型電腦"。
- 在列 43 的語法裡：
  - 執行了函數 dequeue，並在變數 orders 所內含的單端線性佇列裡，出列並傳回其第 1 個資料 "筆記型電腦"。
  - 接著執行了函數 size，並在變數 orders 所內含的單端線性佇列裡，計算並傳回其資料個數 5。
  - 再接著執行了函數 empty，並在變數 orders 所內含的單端線性佇列裡，判斷是否已經不存在任何資料了？！最後傳回 1 (成立) 或 0 (不成立)。

06

CHAPTER

APCS 相關考題的實作與解析

本章學習重點

- 6.1 實作題的綜合演練 part 1
- 6.2 實作題的綜合演練 part 2
- 6.3 實作題的綜合演練 part 3
- 6.4 實作題的綜合演練 part 4

6.1 實作題的綜合演練 part 1

以下是「編碼」相關題目。

任何文字與數字在電腦中儲存時都是使用二元編碼，而所謂二元編碼也就是一段由 0 與 1 構成的序列。在本題中，A ~ F 這六個字元由一種特殊方式來編碼，在這種編碼方式中，這六個字元的編碼都是一個長度為 4 的二元序列，對照表如下：

字元	A	B	C	D	E	F
編號	0101	0111	0010	1101	1000	1100

請你寫一個程式從編碼辨識這六個字元。

輸入格式

第一行是一個正整數 N，$1 \le N \le 4$，以下有 N 行，每行有 4 個 0 或 1 的數字，**數字間彼此以空白隔開**，每一行必定是上述六個字元其中之一的編碼。

輸出格式

輸出編碼所代表的 N 個字元，字元之間不需要空白或換行間格。

範例一：輸入 1 0 1 0 1	範例二：輸入 1 0 0 1 0
範例一：正確輸出 A	範例二：正確輸出 C
範例三：輸入 2 1 0 0 0 1 1 0 0	範例四：輸入 4 1 1 0 1 1 0 0 0 0 1 1 1 1 1 0 1
範例三：正確輸出 EF	範例四：正確輸出 DEBD

評分說明

輸入包含若干筆測試資料，每一筆測試資料的執行時間限制均為 1 秒，依正確通過測資筆數給分。其中：

第 1 子題組 50 分：$N = 1$。

第 2 子題組 50 分：$N \leq 4$。

範例：ch06-01-xx-01.py

```
01  code_to_alphabet = {'0101': 'A', '0111': 'B', '0010': 'C', '1101': 'D',
     '1000': 'E', '1100': 'F'}
02
03  amount = eval(input(''))
04
05  current_code, result = '', ''
06
07  for i in range(amount):
08    current_code = input('').replace(' ', '')
09
10    result += code_to_alphabet[current_code]
11
12  print(result)
```

輸入資料	輸入資料	輸入資料	輸入資料
1 0 1 0 1	2 1 0 0 0 1 1 0 0	1 0 0 1 0	4 1 1 0 1 1 0 0 0 0 1 1 1 1 1 0 1
輸出結果	輸出結果	輸出結果	輸出結果
A	EF	C	DEBD

說明

- 列 01 的語法，善用了 Python 程式語言所支援的字典資料型態，定義了變數 code_to_alphabet，其初始資料為帶有編碼對應到大寫英文字母的字典常數。

- 列 03 的語法，將文字的「讀取」、「轉換成為數值」、「存放至變數 amount」的 3 個動作，濃縮成為單一列的程式碼。其中，變數 amount 中的數值，即是代表輸入之編碼到底有幾組的數量。
- 列 05 的語法，同時定義了變數 current_code 與 result，其初始資料皆為空字串。
- 列 07 的迴圈 for 語法，使得編碼的數量，成為了列 08 與列 10 的程式碼，會被執行次數的次數。
- 列 08 的語法，將後續被輸入而代表特定編碼的文字，移除掉其所有夾雜的空格 (space) 字元，最後存放至變數 current_code 裡。
- 列 10 的語法，將多組被輸入的編碼，轉換成為對應的大寫英文字母，並漸次銜接並存放至變數 result 裡。

範例：ch06-01-xx-02.cpp

```
01  #include <iostream>
02  #include <map>
03
04  using namespace std;
05
06  map<string, string> code_to_alphabet
07  {
08    {"0101", "A"}, {"0111", "B"}, {"0010", "C"},
09    {"1101", "D"}, {"1000", "E"}, {"1100", "F"}
10  };
11
12  map<string, string>::iterator it;
13
14  int amount, i, j;
15  string data, current_code, result;
16
17  int main(void)
18  {
19    cin >> amount;
```

```
20    cin.ignore();
21
22    for (i = 0; i < amount; i++)
23    {
24      getline(cin, data);
25
26      current_code = "";
27
28      for (j = 0; j < data.size() + 1; j++)
29        if (data[j] != ' ' && data[j] != '\0')
30          current_code += data[j];
31
32      it = code_to_alphabet.find(current_code);
33      result += it->second;
34    }
35
36    cout << result << endl;
37
38    return 0;
39  }
```

輸入資料	輸入資料	輸入資料	輸入資料
1 0 1 0 1	2 1 0 0 0 1 1 0 0	1 0 0 1 0	4 1 1 0 1 1 0 0 0 0 1 1 1 1 1 0 1
輸出結果	輸出結果	輸出結果	輸出結果
A	EF	C	DEBD

說明

- 列 02 的語法，載入了資源庫 map，以便支援運用資料型態 map 和副屬的資料型態 iterator，以及相關的函數。
- 列 06 至列 10 的語法，善用了 C++ 程式語言所支援的 map 資料型態，定義了變數 code_to_alphabet，其初始資料為帶有編碼與大寫英文字母之對應關係的 map 常數。

- 列 12 的語法，定義了可用來指向資料型態 map 之變數內部各個資料的迭代器變數 it。
- 列 14 的語法，定義了代表編碼的個數之變數 amount，以及迴圈之迭代用途的變數 i 與 j。
- 列 15 的語法，分別定義了變數 data、current_code 與 result。其中：
  - 變數 data 是用來存放，在迴圈各個迭代中被讀取進來，並且夾雜空格字元與編碼的文字。
  - 變數 current_code 是用來存放，在迴圈各個迭代中，被去除空格 (space) 字元之後的特定編碼字串。
  - 變數 result 是用來存放，在迴圈各個迭代中，漸次被銜接起來的大寫英文字母。
- 列 19 的語法，將代表編碼之個數的整數值，讀取進來並存放至變數 amount 裡。
- 列 20 的語法，可避免畫面上額外顯示出多餘的換列。
- 列 22 的迴圈 for 語法，使得編碼的數量，成為了列 24 至列 33 的語法，會被執行的次數。
- 列 24 的語法，使得夾雜空格 (space) 字元的編碼文字，被存放至變數 data 裡。
- 列 26 的語法，將空字串，指定成為變數 current_code 的內含資料。
- 列 28 的迴圈 for 語法，使得編碼文字中的字元個數，成為了列 29 的語法，會被執行的次數。
- 列 29 的語法，是用來判斷，倘若目前被處理的字元，並不是空格字元或字串「結尾」字元的話，則向下執行列 30 的語法。
- 列 30 的語法，將所有並非空格字元和字串「結尾」字元的其他字元，漸次銜接並存放至變數 current_code 裡。

- 列 32 的語法，是用來將迭代器變數 it，指向到變數 current_code 所代表之特定編碼所在的那組資料。
- 列 33 的語法，是用來將特定編碼所對應的大寫英文字母，漸次銜接並存放至變數 result 裡。