Ch10. 佇列 Queue

在介紹完堆疊之後，就可以來看下一個資料結構，佇列 Queue。

課程大綱

A. 佇列 Queue 簡介
B. 佇列 Queue 實作
C. Queue @ C++ STL
D. Deque
E. Priority Queue

首先，一樣會簡介佇列的工作原理，以及在什麼情況下會使用佇列。

再來，會示範如何實作出一個佇列，接著，直接使用 C++ STL 裡面的 Queue 來解決一些 LeetCode 裡面的題目。

最後，會介紹兩種特化的 Queue：Deque 和 Priority Queue。

第一節：佇列 Queue 簡介

1. 佇列 Queue

(1) 佇列的特性與常用操作

跟堆疊類似，佇列有固定的新增和刪除方向，但是佇列的插入和刪除方向在「異側」，這又叫做 first-in-first-out（FIFO），最先進入結構的資料會最先出來。

這樣的運作模式就像在電影院排隊入場時，新來的人要從後面（隊伍尾端）開始排，而入場是從前面（隊伍的開頭）一個個入場，假如新增資料在右邊，刪除資料則從左邊，兩種操作在「異側」進行的，就叫做佇列。

First in（最先進去的）就會 first out（最先出來）。

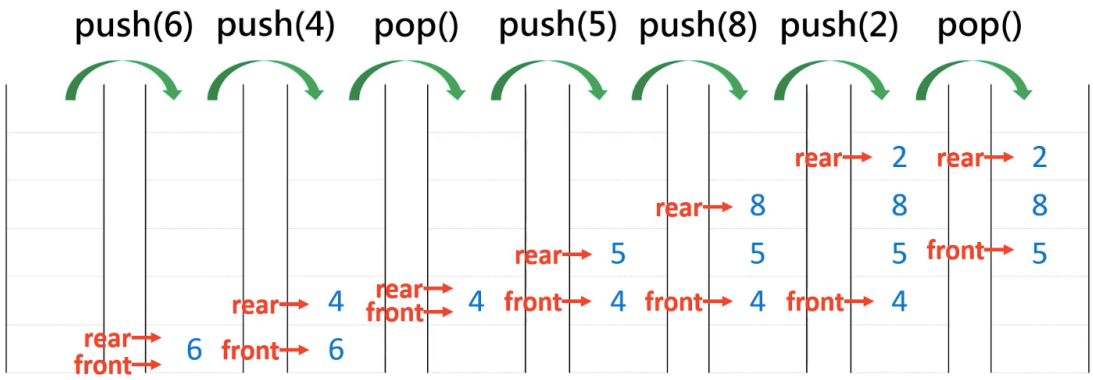
佇列常見的操作與堆疊非常相似，唯一的不同是因為資料有「兩端」，所以可以回傳開頭的資料，也可以回傳結尾的資料：front 代表前端 / 刪除端的資料，rear 則代表的是末端 / 新增端的資料。

佇列 Queue 常用操作（比堆疊多一種操作，因為兩端資料都可回傳）	
push	新增一筆資料
pop	刪除一筆資料
front	回傳前端的資料
rear	回傳末端的資料
empty	確認 queue 中是否有資料
size	回傳 queue 中的資料個數

(2) 在佇列中新增與刪除資料

下面的例子中，push 是「新增資料」，pop 是「刪除資料」。因為新增和刪除在異側，有兩個指標 front 和 rear 分別指向兩端的資料。

另外，因為從「上方」新增，從「下方」刪除，所以用圖形描述 Queue 時，兩端都是開口（不像 stack 只有上方是開口）。



操作	front	rear	說明
push(6)	6	6	front 和 rear 都指向唯一的一筆資料 6
push(4)	6	4	rear 指向新加入的資料 4 front 指向接下來進行刪除會刪掉的資料 6
pop()	4	4	從下方把 6 刪掉，rear 和 front 都指向 4
push(5)	4	5	從上方新增 5
push(8)	4	8	從上方新增 8
push(2)	4	2	從上方新增 2
pop()	5	2	把最下方的 4 刪掉

(3) 練習判斷佇列中的新增與刪除

給定 queue = {1,2,3}，方向為右進左出，經過以下操作後，佇列的內容為何？

A. push(4)

B. pop()

C. push(5)

D. push(6)

E. push(7)

F. pop()

G. pop()

解答

A. push(4)：{1,2,3,4}

B. pop()：{2,3,4}

C. push(5)：{2,3,4,5}

D. push(6)：{2,3,4,5,6}

E. push(7)：{2,3,4,5,6,7}

F. pop()：{3,4,5,6,7}

G. pop()：{4,5,6,7}

2. 佇列 Queue 的用途

A. 依序處理先前的資訊
   a. 常用來做資料的緩衝區
   b. 記憶體（標準輸出、檔案寫入）
   c. 印表機輸出
   d. CPU 的工作排程
B. 迷宮探索、搜尋：Breadth-First Search（BFS）
C. 無法得知 queue 裡有哪些資料：與堆疊相同，只能以 pop 依序拿出資料

通常佇列 Queue 會被用來依序處理資訊，比如資料的緩衝：進行 cout 和 cin 的時候，電腦並不會馬上進行輸出，而是先存放在緩衝區 buffer 裡面，等到緩衝區滿的時候，再依序輸出，因為每次輸出都很花費時間。

舉個生活上的類比，當媽媽要求你去買蘋果，你可能不會馬上出門，而是先將這個任務寫在清單上，隨後媽媽再要你去買可樂，你可能也不會馬上出門，繼續記錄在清單上。當媽媽再叫你去買醬油時，因為清單寫滿了，你才決定一次去把清單上的東西買齊。

這時「3」就是你的緩衝區大小，累積到需要買「3」樣東西的時候，才會一次去買回來。電腦的緩衝區概念與此相似，其結構基本上就是佇列 queue，因為先記入緩衝區的任務應先被處理，與佇列的資料處理次序相符。

印表機的輸出和 CPU 的工作排程也是：假設今天教室中的二十台電腦共用一台印表機，如果這些電腦在很短的時間內都要求列印，印表機應該如何決定該先印誰的文件呢？比較公平的做法應該是哪台電腦的文件先傳送到印表機，就先把這台電腦到文件印出，所以印表機當中就有類似佇列的資料結構，它每接收到一個要列印的文件，就放到佇列中，可以不停接收，而真正進行列印時，

則從另一端取出一個一個文件依序印出，先到者就先印出來。

CPU 的「工作排程」也是將每個工作一一放進佇列，之後從佇列的另一端拿出來依序處理。

跟 Stack 一樣，我們無法跳過前面的資料得知 Queue 中間的某筆資料內容為何，所以要知道中間的資料內容，一定要依序使用 pop，將在前面的資料全部取出後才能得知。

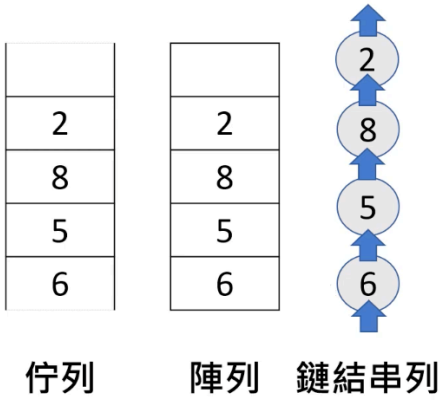
3. 佇列的架構

佇列的架構主要有兩種，第一種是比較好理解的線性佇列：

(1) 線性佇列

a. 可以用陣列或鏈結串列來實作

b. 佇列在陣列中容易遇到容量問題



使用陣列實作線性佇列的時，會遇到容量的問題：陣列的長度如果是 N，則最多只能新增 N 次資料。

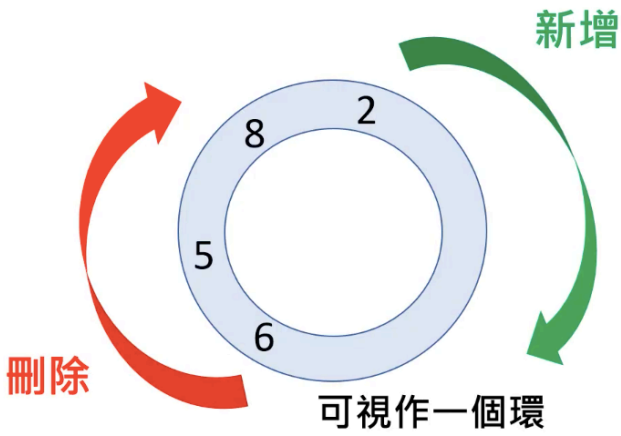
陣列實作的 Queue，從右邊新增資料，並從左側刪除（刪除後不會移動剩下的資料）。因為右邊的空間一直被填充，左邊空出的空間卻不能再被使用，所以很快整個陣列就會被塞滿。

(2) 環狀佇列 Circular Queue

環狀佇列可以解決上述的容量問題，但在「擴充空間」上則比較麻煩。

環狀佇列的新增端（假設是右邊）滿了之後，要再新增資料，會回頭從最左邊開始塞，這很像把線性佇列的頭跟尾連在一起變成環狀，雖然新增和刪除仍然是在異側，但是兩端連在一起，如同一個環。

雖然這樣可以一定程度解決容量問題，但是空間擴充相當比較麻煩，因此在實作佇列時，會選擇使用沒有容量問題的鏈結串列來進行（不過相對的，佔用記憶體空間就較大）。



第二節：佇列的實作

接下來就要開始實作佇列，本書使用鏈結串列來示範。

1. 佇列的節點

首先，宣告一個包含資料與指標 Next 的節點 Node

佇列的節點 Node	
1	template<typename T>
2	struct Node{
3	T Data;
4	Node* Next;
5	};

2. 宣告佇列類別

在佇列的類別中宣告 First 與 Rear 指標（其實叫做 Front 更貼切，但是這個關鍵字已經使用在函式的名稱上了），First 和 Rear 都要初始化為空指標。

分別完成下列函式

A. Front：回傳刪除端的資料

B. Back：回傳最後新增的資料

C. Empty：判斷是否為空

D. Size：回傳裡面有幾筆資料

E. Push：從 Rear 端新增一筆資料

F. Pop：從 Front 端刪除一筆資料

G. Print_Queue：印出佇列中的資料（方便測試，STL 裡面沒有）

佇列的宣告：Class Queue	
1	template<typename T>
2	class Queue{
3	private:
4	Node<T>* First;
5	Node<T>* Rear;
6	public:
7	Queue();
8	T Front();
9	T Back();
10	bool Empty();
11	int Size();
12	void Push(T);
13	void Pop();
14	void Print_Queue();
15	}

3. 方法的實作邏輯

(1) Front：回傳刪除端（前端，排隊時進場的方向）的資料

A. 確認 First 不為空指標

B. 回傳 First 指到的資料

(2) Rear：回傳新增端（後端，排隊時新來的人排的方向）的資料

A. 確認 Rear 不為空指標

B. 回傳 Rear 指到的資料

(3) Empty：確認 Queue 裡是否有資料

A. 確認 First 跟 Rear 是否皆為空指標

(4) Size：查詢 Queue 的長度

A. 確認 First 跟 Rear 皆不為空指標（否則回傳 0）

B. 計算從 First 走到 Rear 需要經過幾個 node

(5) Push：新增一筆資料

A. 宣告一個指向空指標的新 Node

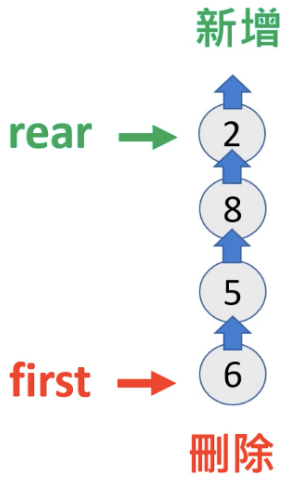
B. 讓 Rear 指到的 Node 指向該 Node

C. 讓 Rear 也指向該 Node

(6) Pop：刪除一筆資料

A. 讓 First 指向下個 Node

B. 若下個 Node 為空指標，則令 Rear 為空指標
（因為原本只有一筆資料，刪除後變為空佇列）



每個節點的指向都是「從 first 指向 rear 方向」，這樣實作起來會比由 rear 指向 first 容易。

4. 佇列的建構式與 Print_Queue

初始化一個佇列，並完成其中的：

A. 建構式

B. Print_Queue 函式

建構式 Queue	
1	template <typename T>
2	Queue<T>::Queue(){
3	// First 與 Rear 指向空指標
4	First = Rear = nullptr;
5	}

Print_Queue：印出佇列中資料	
1	template <typename T>
2	void Queue<T>::Print_Queue(){
3	
4	// 例外處理：空 Queue
5	if (Empty())
6	cout << "This queue is empty!" << endl;
7	
8	// 一般情形
9	else {
10	// 節點指標 current 從 First 開始往後移動
11	Node<T>* current = First;
12	
13	cout << "Queue: ";
14	
15	// current 移動到 Rear 之前繼續處理
16	while(current != Rear){
17	cout << current->Data << " ";
18	current = current->Next;
19	}
20	

```
21              // 輸出 Rear 的資料
22              cout << current->Data << endl;
23          }
24
25  }
```

5. 實作 Queue 的其他方法

Front：回傳刪除端的資料

```
1   template <typename T>
2   T Queue<T>::Front(){
3       // 例外處理：空 queue
4       if(Empty()){
5           cout < "Error: This queue is empty!" << endl;
6           return 0;
7       }
8       // 一般情況
9       return First->Data;
10  }
```

Rear：回傳新增端的資料

```
1   template <typename T>
2   T Queue<T>::Rear(){
3       // 例外處理：空 queue
4       if(Empty()){
5           cout < "Error: This queue is empty!" << endl;
6           return 0;
7       }
8       // 一般情況
9       return Rear->Data;
10  }
```

Empty：確認 Queue 是否為空

```
1  template <typename T>
2  bool Queue<T>::Empty(){
3      // 當 First 和 Rear 都是空指標時，回傳 True，否則回傳 False
4      return (First && Rear)== nullptr;
5  }
```

Size：查詢 Queue 的長度

```
1  template <typename T>
2  int Queue<T>::Size(){
3
4      // 例外處理：空 queue
5      if(Empty())
6          return 0;
7
8      // 一般情形
9      int len = 1
10
11     // 計算 current 從 First 到 Rear 經過幾個節點
12     Node<T>* current = First;
13     while(current != Rear){
14         current = current->Next;
15         len++;
16     }
17     return len;
18 }
```

Push：新增一筆資料

```
1  template<typename T>
2  void Queue<T>::Push(T value){
3
4      // 例外情況:空 queue
5      if(Empty()){
6          // 新增一個節點（資料是 value，next 是空指標）
```

```
7               First = Rear = new Node(value, nullptr);
8               return ;
9       }
10
11      // 一般情形
12      // 新增一個節點，從 Rear 端加入
13      Node<T>* new_node = new Node<T>{value, nullptr};
14      Rear->Next = new_node;
15      Rear = new_node;
16
17  }
```

Pop：刪除一筆資料

```
1   template<typename T>
2   void Queue<T>::Pop(){
3
4       // 例外處理：空 queue
5       if(Empty())
6               return ;
7
8       // 一般情形
9       // 用 tmp 記錄 First 的位置以便後續釋放
10      Node<T>* tmp = First;
11      // First 往後移一筆資料，相當於把第一筆資料移到鏈結串列之外
12      First = First->Next;
13      delete tmp;
14
15      // 例外處理：如果刪除完是空 queue，要調整 Rear
16      if(First == nullptr)
17              Rear = nullptr;
18  }
```

測試 Queue [main.cpp]

```
1   int main(){
2
3       // 宣告並初始化佇列 data
4       Queue<int> data;
5       for(int i=1;i<=10;i++)
6           data.Push(i);
7
8       // 印出 data 中的所有資料
9       data.Print_Queue();
10      // 印出刪除端資料
11      cout << "Front: " << data.Front() << endl;
12      // 印出新增端資料
13      cout << "Back: " << data.Back() << endl;
14
15      // 刪除一筆資料
16      data.Pop();
17      data.Print_Queue();
18
19      // 新增一筆資料 11
20      data.Push(11);
21      data.Print_Queue();
22
23      return 0;
24  }
```

執行結果

```
Queue: 1 2 3 4 5 6 7 8 9 10
Front: 1
Back: 10
Queue: 2 3 4 5 6 7 8 9 10
Queue: 2 3 4 5 6 7 8 9 10 11
```

5. 用佇列模擬洗牌

A. 題目

給定一疊卡牌，輸入 n 時代表牌有 n 張，編號分別從 1 到 n，每次操作會依照下面順序：

a. 將目前最上面的卡牌丟掉
b. 再把一張卡牌從最上面放到最下面

重複以上操作，每一輪會減少一張牌，直到剩下最後一張牌時，請輸出該張牌的編號。

B. 出處

https://onlinejudge.org/index.php?option=onlinjudge&Itemid=8&page=show_problem&problem=1876

C. 輸入與目標輸出

```
Please enter N:
>> 5
Queue: 1 2 3 4 5
Now, discard card #1
Now, put card #2 to the bottom.
Queue: 3 4 5 2
Now, discard card #3
Now, put card #4 to the bottom.
Queue: 5 2 4
Now, discard card #5
Now, put card #2 to the bottom.
Queue: 4 2
Now, discard card #4
Now, put card #2 to the bottom.
Queue: 2
The last remaining card is: #2
```

D. 解題邏輯

這疊卡牌可以看成一個 Queue，每次丟掉一張牌時從最上面刪除。

第一步是從上方（刪除端）拿出一張牌丟掉，第二步是再從上方（刪除端）拿出一張牌放到下面（新增端），不斷重複，直到 Queue 的長度 Size() 為 1。

洗牌 Shuffle

```
1  int main(){
2
3      int N;
4      cout << "Please enter N:";
5      cin >> N;
6
7      // 用佇列 data 來代表卡堆
8      Queue<int> data;
9
10     // 把 1 到 10 依序放到 data 中
11     for(int i=1;i<=N;i++)
12         data.Push(i);
13
14     // 在卡堆中剩下一張牌前繼續進行
15     while(data.Size()>1){
16
17         // 刪除掉卡堆最上面的那張牌
18         int delete_number = data.Front();
19         cout << "Now, discard card #" << delete_number << endl;
20         data.Pop();
21
22         // 再把最上面的一張牌放到卡堆下方（新增進 Queue）
23         int move_number = data.Front();
24         cout << "Now, put the card #" << move_number << " to the bottom."
25         << endl;
26         data.Pop();
```

```
27              data.Push(move_number);
28
29          }
30
31          // 印出剩下的一張牌
32          // data.Front() 和 data.Back() 此時指到同一張牌
33          cout << "The last remaining card is: # " << data.Front();
34
35          return 0;
36      }
```

第三節：STL 中的 queue

接著來看 C++ STL 裡面的佇列 queue 應該如何使用。

1. queue 的使用

A. 引用函式庫

```
#include <queue>
```

B. 宣告：<> 中代表 queue 要存放什麼型態的資料

```
queue<datatype> queue_name;
```

注意：queue 與 stack 一樣沒有迭代器！

2. 佇列 queue 的操作

沒有在特定位置新增、在特定位置刪除的函式：

STL 中 queue 的操作	
queue.push(value);	新增一筆資料
queue.pop();	刪除一筆資料
queue.front();	回傳刪除端資料
queue.back();	回傳新增端資料
queue.empty();	判斷 queue 是否為空
queue.size();	回傳 stack 長度

3. 使用 STL 中的 queue

測試 STL 中的 Queue

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4
5  int main()
6  {
7          queue<int> data;
8          for(int i=0;i<10;i++)
9                  data.push(i);
10
11         // queue: 0 1 2 3 4 5 6 7 8 9
12
13         cout << data.front() << endl;        // 0
14         cout << data.back() << endl;         // 9
15
16         data.pop();
17         data.pop();
18
19         // queue: 2 3 4 5 6 7 8 9
20
21         cout << data.front() << endl;          // 2
22         cout << data.back() << endl;           // 9
```

```
23
24          return 0;
25
26  }
```

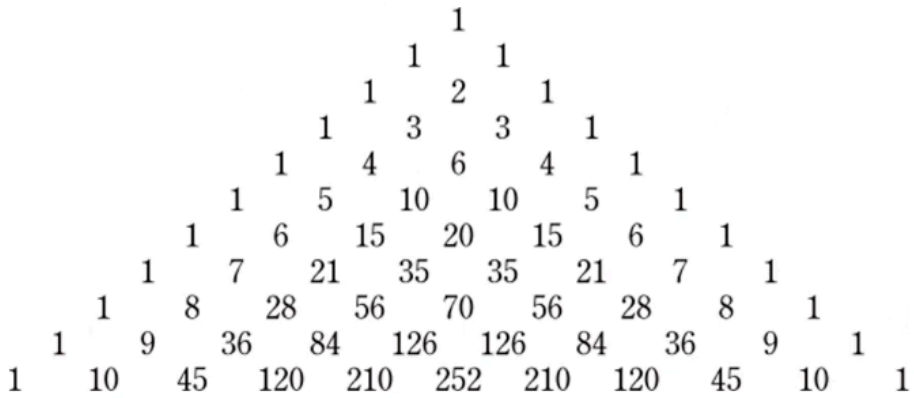
4. 巴斯卡三角形

A. 題目

巴斯卡三角是二項式定理中的圖形，以下圖為例，每列數列中「第一個」與「最後一個」數字皆為一，其餘則都是上方數列中前後兩個數字的和。

B. 目標輸出

讓使用者輸入一整數 N，輸出 N 層的巴斯卡三角形。



C. 解題邏輯

把每一列都看成是一個 queue，每次從 queue 中取出兩筆資料，只要一直從目前的 queue 中拿出兩個數字相加，就會產生下一列的值，比如 1 4 6 4 1 兩兩拿出來相加會變成

(1+4) (4+6) (6+4) (4+1)

這也就是下一列除了頭尾的 1 以外的「5 10 10 5」，前後各加上 1，即可產生下一列的「1 5 10 10 5 1」。

巴斯卡三角形

```
1   #include <iostream>
2   #include <queue>
3   using namespace std;
4
5   int main()
6   {
7       // 取得使用者輸入的整數 N
8       int N;
9       cout << "Please enter N:";
10      cin >> N;
11
12      queue<int> data;
13      int temp1, temp2;
14
15      cout << "Pascal Triangle:" << endl;
16
17      // 第一列中的數字只有 1
18      data.push(1);
19      // 第一列比較特別，直接輸出
20      cout << 1 << endl;
21
22      // 第二列到第 N 列
23      // 迴圈的 i 代表第幾列
24      for (int i=2;i<=N;i++){
25
26          // 要把 temp1 設為 0，
27          // 之後才能跟這行開頭的 1 相加產生下一行開頭的 1
28          temp1 = 0;
29
30          // 正要產生第 i 列時，從第 i-1 列拿出資料
31          // 總共有 i-1 筆，因此 j 從 1 到 i-1 共執行 i-1 次
32          for (int j=1;j<=i-1;j++){
33
```

```
34                  // 把目前要處理的上一列資料拿出來
35                  // 比如 1 4 6 4 1 中的 6
36                  temp2 = data.front();
37                  data.pop();
38
39                  // 此時的 temp1 會是 6 的前一個數字 4
40                  // 因此相加產生下一行 1 5 10 10 5 1 的第一個 10
41                  cout << temp1 + temp2 << " ";
42                  data.push(temp1 + temp2);
43                  // 把 temp1 往後移動到 temp2，也就是從 4 移動到 6
44                  temp1 = temp2;
45
46                  // 每一列尾端的 1 不會自動產生
47                  // 不像開頭的 1 可由 temp1 和上一行開頭的 1 產生
48                  // 所以要手動多加一個 1
49                  data.push(1);
50                  cout << 1 << endl;
51              }
52          }
53
54          return 0;
55  }
```

執行結果

```
Please enter N:
>> 5
Pascal Triangle:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

5. 舞會配對

A. 題目

舉辦一個舞會，舞會中的男生、女生人數不相同，為了公平起見，人數多的性別必須輪流與異性跳舞，用大寫 A、B、C、... 表示男生，小寫 a、b、c、... 表示女生，輸出 N 輪跳舞的配對情形。

輸入：跳舞輪數、男生數目、女生數目
輸出：每輪的舞伴配對情形

B. 說明

因為一個人與異性跳完舞後，就要排到同性隊伍的最後面，直到所有同性都跳過舞了才會再次輪到，與佇列特性相符，因此本題適合使用佇列 queue。

C. 目標輸出

以四男三女，共跳六輪為例

男：ABCD
女：abc

Please enter rounds or party, number of boys and girls:
>> 6 4 3
There are 6 rounds, 4 boys, and 3 girls.
Round #1: A<-->a
Round #2: B<-->b
Round #3: C<-->c
Round #4: D<-->a
Round #5: A<-->b
Round #6: B<-->c

D. 解題邏輯

編號 A 的男生與女生跳完後，要等到編號 BCD 的男生都分別跟女生跳完後才會再輪到。

可以把男生和女生各放入一個 queue 當中（類似排隊），當排到隊伍最前面，跳完舞後，就回到隊伍最後面重新等待，也就是一筆資料從佇列中 pop 掉後，就馬上重新 push 到佇列中等待下輪處理。

舞會配對

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4
5  int main()
6  {
7      // 宣告兩個 queue 來存放代表男生和女生的字元
8      queue<char> boys, girls;
9      int rounds, number_boy, number_girl;
10
11     // 取得使用者輸入的參數
12     cout << "Please enter rounds, number of boys and girls:" << endl;
13     cin >> rounds >> number_boy >> number_girl;
14
15     // 依序把大寫 A B C 放入男生的陣列
16     for (int i=0;i<number_boy;i++){
17         // 'A'+0 = 'A', 'A'+1 = 'B', ...
18         boys.push('A'+i);
19     }
20
21     // 依序把小寫 a b c 放入女生的陣列
22     for (int i=0;i<number_girl;i++){
23         // 'a'+0 = 'a', 'a'+1 = 'b', ...
24         girls.push('a'+i);
```

```
25          }
26
27          // 共進行 rounds 輪
28          for (int i=0;i<rounds;i++){
29              // 從兩個佇列中各取出隊伍最前端的男生和女生
30              char boy = boys.front();
31              boys.pop();
32              char girl = girls.front();
33              girls.pop();
34
35              // 輸出這輪跳舞的兩個人代號
36              cout << "Round #" <<    i+1 << ":" << boy << "<-->" << girl << endl;
37
38              // 把這輪跳舞的男生和女生分別重新排到隊伍的最後面去
39              boys.push(boy);
40              girls.push(girl);
41          }
42
43          return 0;
44  }
```

6. LeetCode#232 用堆疊實作佇列 Implement Queue using Stacks

A. 題目

使用堆疊時做一個「先進先出（FIFO）」的佇列。

該佇列需要提供以下功能

A. void push(int x)：新增元素 x 到佇列後方
B. int pop()：將佇列最前方的元素回傳並從佇列中移除
C. int peek()：回傳佇列最前方的元素
D. boolean empty()：佇列是空的話，回傳 true，否則回傳 false

B. 出處：https://leetcode.com/problems/implement-queue-using-stacks/

C. 解題邏輯

本題要試著用 stack 去實作 queue，注意 stack 與 queue 的差別是 stack 的新增和刪除在同側，queue 的新增和刪除在異側，這代表如果要從這個 queue 刪除一筆資料，（因為是用 stack 實作的）就同等於刪掉 stack 的最下面一筆資料，並維持其他筆資料的順序。

為了要取用到這個最下面一筆資料，要把整個 stack 倒過來才能完成。

使用兩個 stack 可以達成類似的效果，當把裝著原本資料的 stack 裡的資料逐一拿出來放到另一個 stack 裡面，資料的順序就會全部倒過來。比如原本是從底部到上方依序為 [1,2,3]，移到另一個 stack 後就變為 [3,2,1]，此時把最上面的資料去除，最後把剩下的資料再移回原本的 stack 即可。

就像這樣，stack 很常用來把資料倒置，只要把一個 stack 的資料移到另一個 stack，資料的順序會正好倒過來。

用堆疊實作佇列 MyQueue

```
1  class MyQueue{
2  public:
3      // 宣告兩個堆疊 data 和 tmp
4      stack<int> data,tmp;
5
6      // 建構式，不需實作
7      MyQueue(){};
8
9      // 新增資料
10     void push(int x){
11         data.push(x);
12     }
13
14     // 刪除資料
15     int pop(){
16
17         // 把 data 裡的資料完全放到 tmp 之中，同時 data 清空
18         while(!data.empty()){
19             tmp.push(data.top());
20             data.pop();
21         }
22
23         // 刪掉原本 data 最下面的資料（tmp 最上面的資料）
24         int result = tmp.pop();
25
26         // 把資料從 tmp 放回 data
27         while(!tmp.empty()){
28             data.push(tmp.top());
29             tmp.pop();
30         }
31
32         return result;
33     }
```

```
34
35      // 回傳 queue 的前面一筆資料（data 堆疊的最下面一筆資料）
36      // 基本上與 pop 相同，只是不需要用刪除該筆資料
37      int peek(){
38
39          // data 裡的資料完全放到 tmp 之中，同時 data 清空
40          while(!data.empty()){
41              tmp.push(data.top());
42              data.pop();
43          }
44
45          // result 取得原本 data 最下面的資料（tmp 最上面的資料）
46          // 注意不需刪除，因此是用 top 而非 pop
47          int result = tmp.top();
48
49          // 把資料從 tmp 放回 data
50          while(!tmp.empty()){
51              data.push(tmp.top());
52              tmp.pop();
53          }
54
55          return result;
56      }
57
58      // 檢查 queue 是否為空
59      bool empty(){
60          return data.empty();
61      }
62
63  }
```

7. LeetCode#1700. 分配三明治 Number of Students Unable to Eat Lunch

A. 題目

學校的食堂供應「圓形」和「正方形」兩種三明治，分別以 0 和 1 的數字表示。每個學生都要排在隊伍（以佇列表示）中取用，且各自喜歡圓形或正方形其中一種三明治。

食堂中的三明治數量與學生數量相同，所有三明治的資料被放在一個堆疊當中。

給定兩個整數陣列

sandwiches[i] 是第 i 個三明治的形狀（i=0 是堆疊頂端）

students[j] 是第 j 個學生的喜好（j=0 是佇列前端）

若學生遇到不喜歡的三明治絕對不取用，回傳無法吃到三明治的學生數量。

B. 出處：https://leetcode.com/problems/number-of-students-unable-to-eat-lunch/

C. 說明

三明治類型用 0 或 1 表示，分別代表圓形和方形的三明治，而每個學生有自己愛吃的類型，一定要遇到愛吃的，學生才會拿走一個三明治，假設：

students = [1,1,0,0]
sandwiches = [0,1,0,1]

第一個學生想吃 1，第二個學生也想吃 1，第三和第四個學生則想吃 0。一開始第一個學生沒有遇到他想吃的 1 類型三明治，所以他不會拿走這個三明治，而是主動排到隊伍的最後面去，學生的佇列變成 [1,0,0,1]（原本的第一個 1 移到最後方）。

第二個學生一樣沒有遇到想吃的三明治，三明治佇列維持 [0,1,0,1]，所以他也排到最後面去，學生佇列變成 [0,0,1,1]。

第三個學生想吃 0 類型的三明治，而三明治佇列的開頭正好是 0 類型的三明治 [0,1,0,1]，所以第三個學生拿走了開頭的三明治並不繼續排隊，學生的陣列變成 [0,1,1]，三明治陣列變成 [1,0,1]，後續依此類推。

如果最後兩個佇列都清空，就代表所有學生都拿到了三明治，如果有三明治剩下，輸出沒有吃到三明治的學生有幾名。

Sandwiches

```
1  class Solution{
2  public:
3      int countStudents(vector<int>& students, vector<int>& sandwiches){
4
5          // 用佇列儲存學生的喜好資訊
6          // 因為學生是「排隊」取用三明治
7          queue<int> S;
8
9          for(int i=0;i<sandwiches.size();i++){
10             // 取出當前可取用的第一個三明治的形狀
11             int shape = sandwiches[i];
12             // len 是目前隊伍中剩下的學生人數
13             int len = S.size();
14             // counts 記錄這個三明治「不符合」多少個學生的喜好
15             int counts = 0;
16
17             while(true){
18
19                 // 隊伍前端學生的喜好剛好等於可取用三明治時
20                 // 把最前端的學生從佇列中去除
21                 if (shape == S.front()){
22                     S.pop();
23                     break;
24                 }
25                 // 可取用三明治與隊伍最前端學生的喜好不同
26                 else{
```

```
27                              // 隊伍最前面的學生回到最後面排隊
28                              Q.push(Q.front());
29                              Q.pop();
30
31                              // 「喜好不符合」的學生數量 +1
32                              counts++;
33
34                              // 當目前「喜好不符合」的學生數量
35                              // 與隊伍中學生總數相等，代表已經輪完一輪
36                              // 隊伍中的學生都不喜歡最前端的三明治
37                              // 根據題目要求，回傳目前隊伍裡的學生人數
38                              if (counts == len){
39                                      return counts;
40                              }
41
42                      } // end of if
43                  } // end of while
44              } // end of for
45
46              // 如果全部的三明治都分發完畢，回傳 0
47              return 0;
48
49      } // end of countStudents
50  }; // end of Solutions
```

第四節：Deque 和 Priority Queue

接下來，來看兩個 C++ STL 裡面特化的 Queue：Deque 跟 Priority Queue。

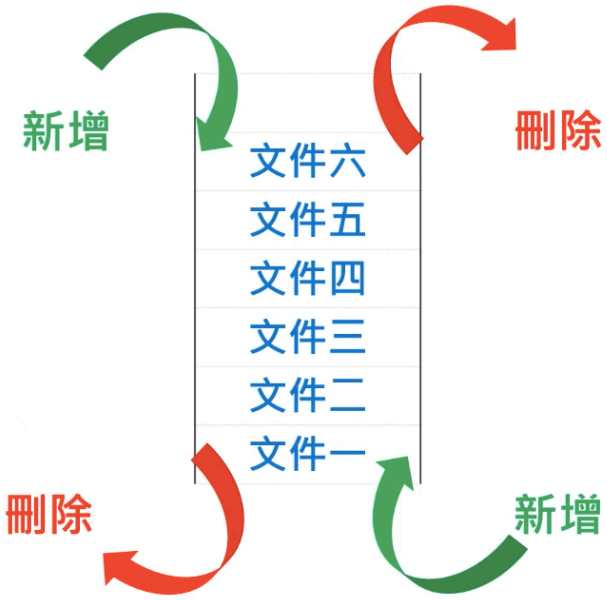
1. Deque

(1) Deque 的特性

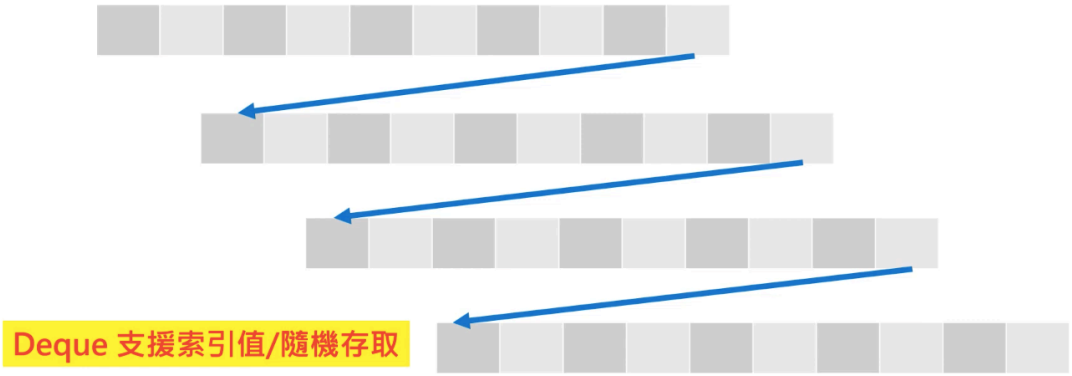
A. double-ends queue
B. 兩端都允許新增或刪除
C. 支援索引值存取與迭代器
D. 引入方式：#include <deque>
E. 利用間接索引完成
   a. map 到許多記憶體空間
   b. 零散的連續記憶體

Stack 只能從同一個方向新增或刪除，Queue 只能從不同方向新增或刪除，在下面的圖中，假設是 Stack，只能從上面新增、上面刪除，假設是 Queue，則只能從上面新增、下面刪除。

如果我們會同時用到兩端的新增和刪除，那麼 Stack 和 Queue 的功能都不適合，因此又發展出了一個資料結構 Deque，兩邊都可進行新增和刪除。



另外，Deque 的特色是支援索引值存取和迭代器，也就是在取用資料的時候，沒有一定要從兩端開始的限制，而可以直接從中間把資料取出來。一般來說，可以用索引值存取的前提是記憶體位置連續，那為什麼 Deque 可以實現索引值存取呢？這是利用間接索引完成的，它用了很多不連續的記憶體空間來拼成一個完整的「塊」。



上圖有四個連續的記憶體區塊，這四個區塊會組成一個連續的 Deque，雖然整個 Deque 是不連續的，但是每個區塊中的記憶體是連續的，這讓它可以支援索引值存取。

要取用第五筆資料，就從第一個區塊開始數 1、2、3、4、5 就可以了，如果要取用第 12 筆資料（假設每個區塊可以各存放 10 筆資料），一樣從第一個區塊開始，走到底後，會指出下一個區塊在哪裡，直接到下個區塊取出即可。

Deque 的原理很像把鏈結串列裡的每個 Node 放大成一整個區塊來儲存資料。

(2) Deque 的常用語法

Deque 的常用語法	
push_back	尾端新增
push_front	頭端新增
pop_back	刪除尾端元素
pop_front	刪除頭端元素
insert	插入特定元素於特定位置
erase	移除某筆資料
clear	清空整個 deque

因為 Deque 的兩端都可以新增和刪除，所以它的常用語法中，新增和刪除也分成兩邊，也由於沒有只能從兩端讀取的限制，所以可以直接使用 insert 或 erase 在特定的位置插入或移除資料。

(3) Deque 和向量 Vector 的差異

Deque 和 Vector 的差異		
	Deque	Vector
頭端新增 / 刪除	$O(1)$	$O(n)$
尾端新增 / 刪除	$O(1)$	$O(1)$
中段新增 / 刪除	$O(n)$	$O(n)$
索引值 / 迭代器存取	支援	支援
記憶體	連續的記憶體組成不連續的塊	連續
大小	彈性	固定
適用情形	兩端新增 / 刪除 不需取得中段的資料	隨機 / 索引值存取

比較 Deque 和 Vector，Deque 在頭端新增只需要 $O(1)$，Vector 則需要 $O(n)$，因為新增後要把後面的資料全部平移；尾端新增則兩者都是 $O(1)$，至於中段的新增和刪除，兩者都是 $O(n)$。

Deque 的大小是彈性的，如果要臨時增加，就多加上一塊連續記憶體存放，Vector 則是固定的。

在適用方面，當大多數的新增和刪除發生在兩端，不需要取得中段的資料時，可以優先考慮 deque，vector 則支援「最快的索引值存取」。

(4) 用 deque 完成先前的練習

A. 卡牌分配：之前用 queue

B. 括號配對：之前用 stack

deque 版卡牌分配

```
1   #include <iostream>
2   #include <deque>
3   using namespace std;
4
5   int main()
6   {
7       // 取得使用者輸入的 N
8       int N;
9       cout << "Please enter N:";
10      cin >> N;
11
12      // 宣告 deque
13      deque<int> data;
14
15      // 把牌放入 deque
16      for (int i=1;i<=N;i++)
17          data.push_back(i);
18
19      // 剩下一張牌之前繼續進行
20      while (data.size()>1){
21          // 去除第一張牌
22          data.pop_front();
23          // 把第二張牌放到卡堆最後面
24          int tmp = data.front();
25          data.pop_front();
26          data.push_back(tmp);
27      }
28      cout << "Last: " << data.front() << endl;
29
```

```
30      return 0;
31  }
```

Deque 版括號配對

```
1   #include <iostream>
2   #include <deque>
3   using namespace std;
4
5   bool isValid(string str){
6
7       deque<char> data;
8
9       // 遇到左括號：放到 deque 中
10      for (char c:str){
11          if (c == '(' || c == '[' || c=='{'){
12              data.push_back(c);
13              continue;
14          }
15      }
16
17      // 遇到右括號但 deque 中沒有對應的左括號：輸出 error
18      if ((c==')' || c==']' || c=='}') && data.empty()){
19          return false;
20      }
21
22      // 遇到右小括號，而 deque 最上面是左小括號，把左小括號去掉
23      // 注意這裡 push 是從後面，pop 也是從後面
24      // 所以是把 deque 當作 stack 來用
25      if (c==')'){
26          if (data.back()=='('){
27              data.pop_back();
28          }
29          else{
30              return false;
```

```
31              }
32          }
33
34          // 遇到右中括號
35          else if (c==']'){
36              if (data.back()=='['){
37                  data.pop_back();
38              }
39              else{
40                  return false;
41              }
42          }
43
44          // 遇到右大括號
45          else if (c=='}'){
46              if(data.back()=='{'){
47                  data.pop_back();
48              }
49              else{
50                  return false;
51              }
52          }
53
54          // 如果遇到的左括號全部被用掉，回傳 true
55          // 如果有左括號沒有被對應到，則回傳 false
56          if (data.empty())
57              return true;
58          else
59              return false;
60
61  } // end of isValid
62
63  int main()
64  {
```

```
65      // 宣告要檢查字串和 deque
66      string str = "{[a+b+(s+f)-d]+2}";
67
68      // 利用 isValid 函式檢查括號對應
69      if(isValid(str))
70          cout << "Valid!" << endl;
71      else
72          cout << "Not valid!" << endl;
73
74      return 0;
75  }
```

執行結果

Valid!

如果把任一個左括號或右括號拿掉，讓括號不對應，則執行結果為 "Not valid!"。

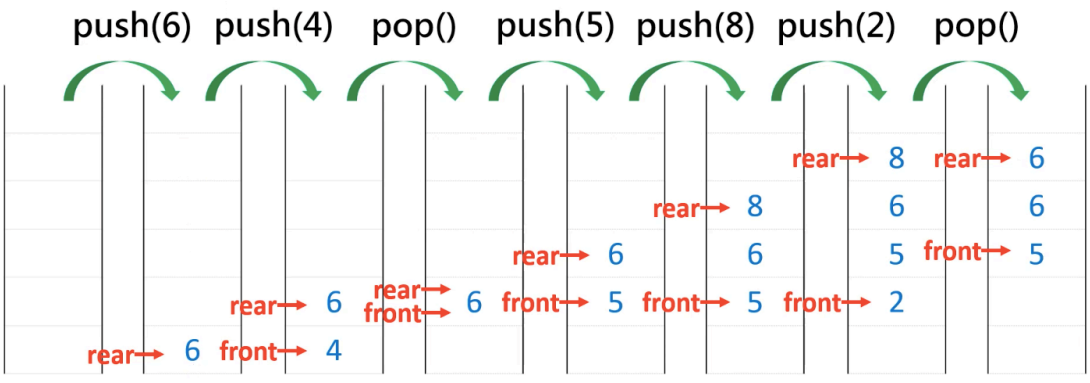
2. Priority Queue 優先權佇列

A. Priority 優先度：額外賦予資料權重

B. Queue：依照優先權依序排列後再依序輸出

(1) Priority Queue 的特性

Priority Queue 的輸出順序不是資料的排列順序，而是根據「權重大小」，權重大的先輸出，權重小的後輸出，權重的決定方式則可以由使用者自訂。



上面的例子裡，假設「數字小的先輸出」。先放入 6 之後再放入 4，因為 4 比 6 小，所以 4 更靠近輸出端，pop 時把 4 輸出。

再輸入 5，因為 5 也比 6 小，因此會放在 6 下面，而輸入 8 時，因為 8 比 6 大，所以會比 6 離輸出端更遠，被加在 6 上面的位置。就像這樣，每次新增與刪除資料不再是根據新增的順序，而是根據「權重大小」，刪除資料時，也是根據優先權的大小加以取出。

Priority Queue 因為每次插入資料就加以排序，所以可以依照權重大小順序吐出資料。

A. Max-Priority Queue：讀取資料時拿到「權重最大」的資料

B. Min-Priority Queue：讀取資料時拿到「權重最小」的資料

(2) Priority Queue 的基本操作

Priority Queue 的基本操作	
insert	將資料插入 queue
increase key	改變某資料的權重
extract max	取得權重最大的資料，並自 queue 中刪除

(3) Priority Queue 的使用

A. 引入函式庫

```
#include<queue>
```

B. 宣告類別實體

```
priority_queue<datatype,container,compared_method>
```

a. datatype：要比較的資料型態

b. container：組成 queue 的容器（vector 或 deque），預設是 vector

c. compared_method：比較方式

d. 最簡單的宣告方式是只傳入資料型態 priority_queue<datatype>

- 預設權重越大越接近 top

C. 控制優先權順序

a. compared_method

- greater<datatype>：由小到大
- lesser<datatype>：由大到小
- 預設由大到小（lesser）輸出、即「小於運算子 <」

b. 自定義函式（細節是物件導向的內容）

- 重載運算子 <
- 寫一個結構或類別，內含「小括號運算子（）」重載

(4) 測試 priority queue

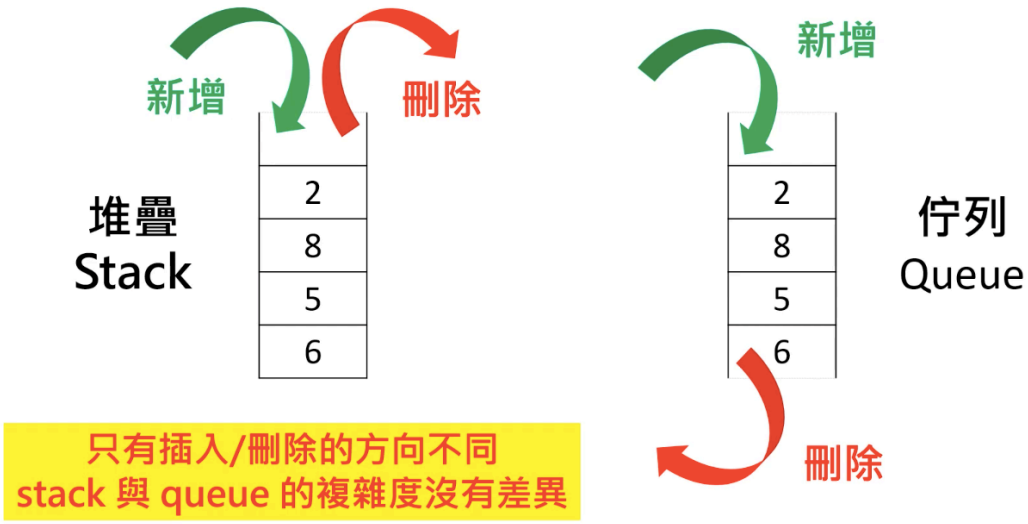
測試 priority queue	
1	priority_queue<int> p_queue;
2	p_queue.push(8);
3	p_queue.push(3);
4	p_queue.push(5);
5	p_queue.push(7);
6	p_queue.push(2);

一般的 queue 因為先進先出，資料順序是 8 3 5 7 2，但根據 priority queue 的預設順序，則會是 8 7 5 3 2。

A. p_queue：8 7 5 3 2

B. p_queue.top()：8

C. p_queue.pop()：return 8，p_queue = 7 5 3 2



總結而言，佇列 queue 和上一章中介紹的堆疊 stack 只有「插入 / 刪除資料的方向」限制不同，操作的複雜度上則沒有差異。