程式設計
Ch16. Stack and Queue

Chuan-Chi Lai 賴傳淇

Department of Communications Engineering
National Chung Cheng University

Spring Semester, 2024

Outline

回顧：抽象資料型別
Abstract Data Type

回顧：抽象資料型別

- 當我們在設計演算法的時候，通常都會先想好整個演算法的架構才會開始寫，免得造成在撰寫的過程中一直在新增或刪除函式，而我們在構思這些函式的時候所運用到的就是抽象資料型別 (Abstract Data Type, ADT)。
- ADT 是一種理論上的概念，由資料 (Data) 及操作 (Operation) 組成，著重於資料的運算，並不考慮實作時的細節或資料本身的性質、可藉由不同的 Data Structure 來實作。

回顧：抽象資料型別 (Abstract Data Type)

範例

- 假設今天我們要做的是學生名冊的建立，我們的資料就有學生的個資，而這些資料的特性及操作如下：

Data	Operation
1. 學生的座號是有順序的	CreateClass: 建立一個新班級
2. 每個學生的資料型態是相同的或同性質的	DeleteClass: 刪除一個班級
	IsEmpty: 判斷一個班級人數是否為空
	Number: 查詢班級人數
	Add: 加一位學生至班級
	Delete: 刪除一位學生資料
	GetInfo: 查詢學生資料

回顧：抽象資料型別 (Abstract Data Type)

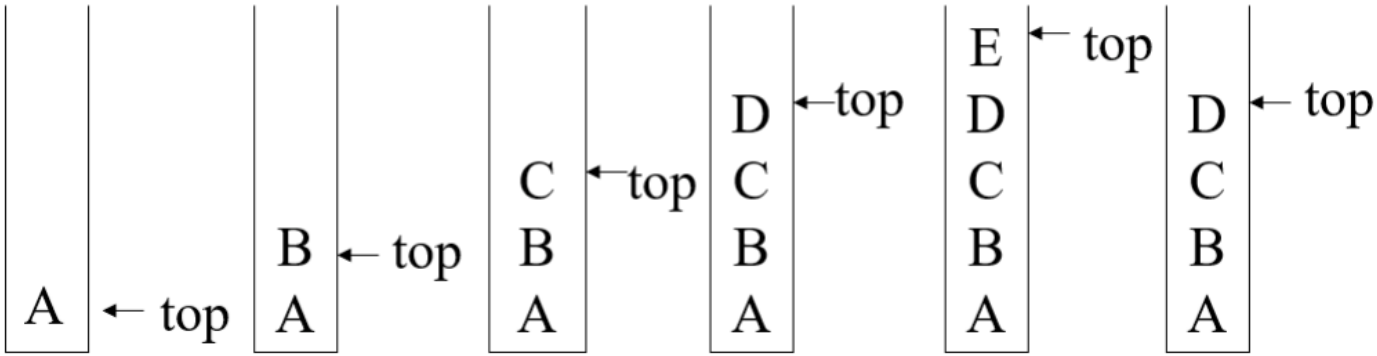
- 8 種常見的抽象資料型別：

No.	ADT	重點
1	Stack	Last In, First Out (LIFO)
2	Queue	First In, First Out (FIFO)
3	Dictionary/Map	key-value pairs 的集合物件，key 在集合物件中不會重複
4	Set	每個物件只會出現一次，無順序性
5	Sequences/List	每個物件可出現多次，且有順序性
6	Priority Queue	有優先等級的 Queue，Dequeue 時依照優先等級取值
7	Graph	- 由 Vertices 和 Edges 組成 - 可分為有向圖或無向圖 - 可用 Adjacency Matrix 和可用 Adjacency List 表示
8	Tree	- 為不含環狀/迴路的圖 - Edges 的數量為 Vertices 的數量減 1

堆疊
Stack

堆疊 (Stack)

- 我們在先前介紹遞迴時，提到了 Stack LIFO(Last In and First Out) 的特性，而這種特性常被運用在遞迴、回溯法及深度優先搜尋法上。



堆疊的抽象資料型態

- 堆疊是一次只能從最上面增加或移除東西的抽象資料型態：

堆疊的抽象資料型態

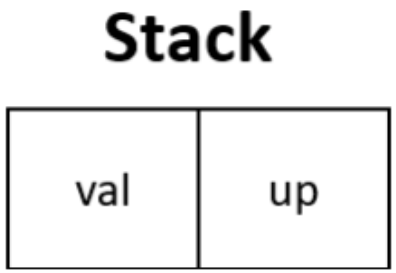
- Data：一個線性串列，所有的運算處理皆由頂端元素開始
- Operation：分為會改變堆疊狀態的 push 和 pop，以及觀察堆疊狀態的 top、is_empty、is_full、print 和 size。
  - push：把元素放進去堆疊。
  - pop：把堆疊最頂端的元素拿出來。
  - top：顯示堆疊頂端的元素。
  - is_empty：判斷堆疊是否為空。
  - is_full：判斷堆疊是否為滿。
  - print：把堆疊中的元素從全部印出來。
  - size：堆疊高度。

以鏈結串列實作堆疊
Implement Stack with Linked Lists

建立堆疊

- 堆疊是一種 ADT，因此不限定實作所使用的資料結構。
- 在這裡我們使用鏈結串列來實作堆疊，val 為每一個堆疊的值，而 up 則指向上面的堆疊。

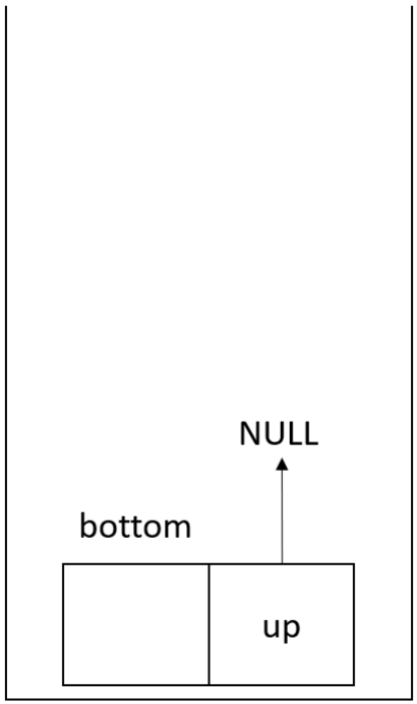
```
1 typedef struct Stack {
2     int val;
3     struct Stack *up;
4 } Stack;
```



以鏈結串列實作堆疊 (Implement Stack with Linked Lists)

- 宣告指標 bottom 指向 Stack 底端以便後續利用，Stack 底端，只負責指出最底端的位置，因此不存放任何數值。

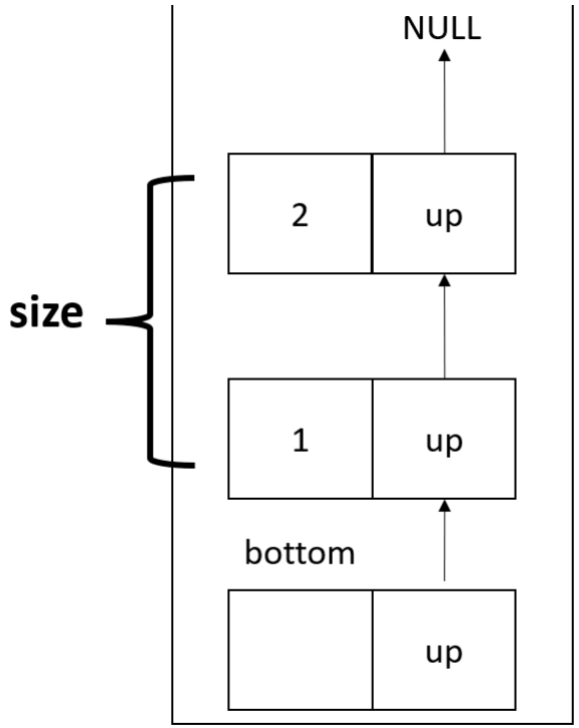
```
1 int main() {
2     Stack *bottom = (Stack*)malloc(sizeof(Stack));
3     bottom->up = NULL;
4 }
```



計算長度

- 知道堆疊高度的方法就是從最下面的堆疊往上到頂端元素並計數。

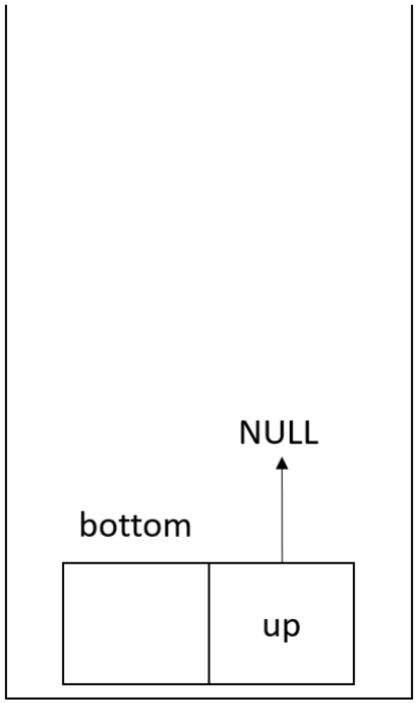
```
1 void size(Stack *bottom) {
2     int count = 0;
3     Stack *search = bottom->up;
4     while(search != NULL) {
5         count++;
6         search = search->up;
7     }
8     printf("%d\n", count);
9 }
```



判斷堆疊為空

- 因為 bottom 只負責指出最底端的位置，所以只要指向的位置為空，這個堆疊就是空的。

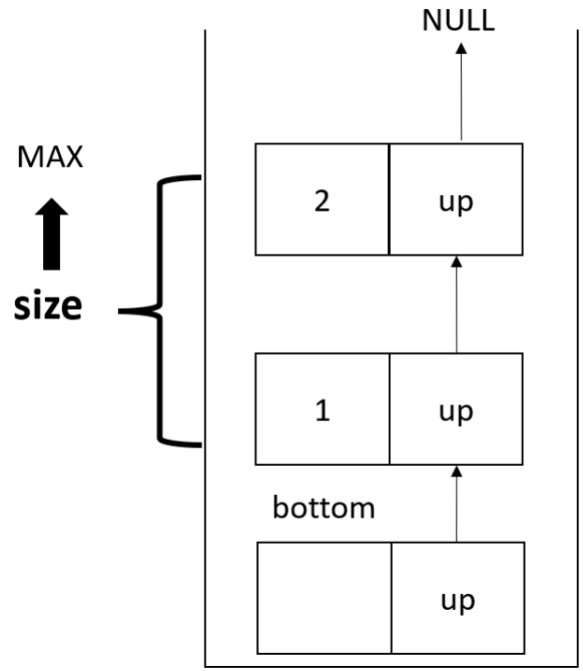
```
1 bool is_empty(Stack *bottom) {
2     if(bottom->up == NULL) {
3         return true;
4     }
5     return false;
6 }
```



判斷堆疊為滿

- 而相對的，只要堆疊高度到了最高點的話，這個堆疊就是滿的。

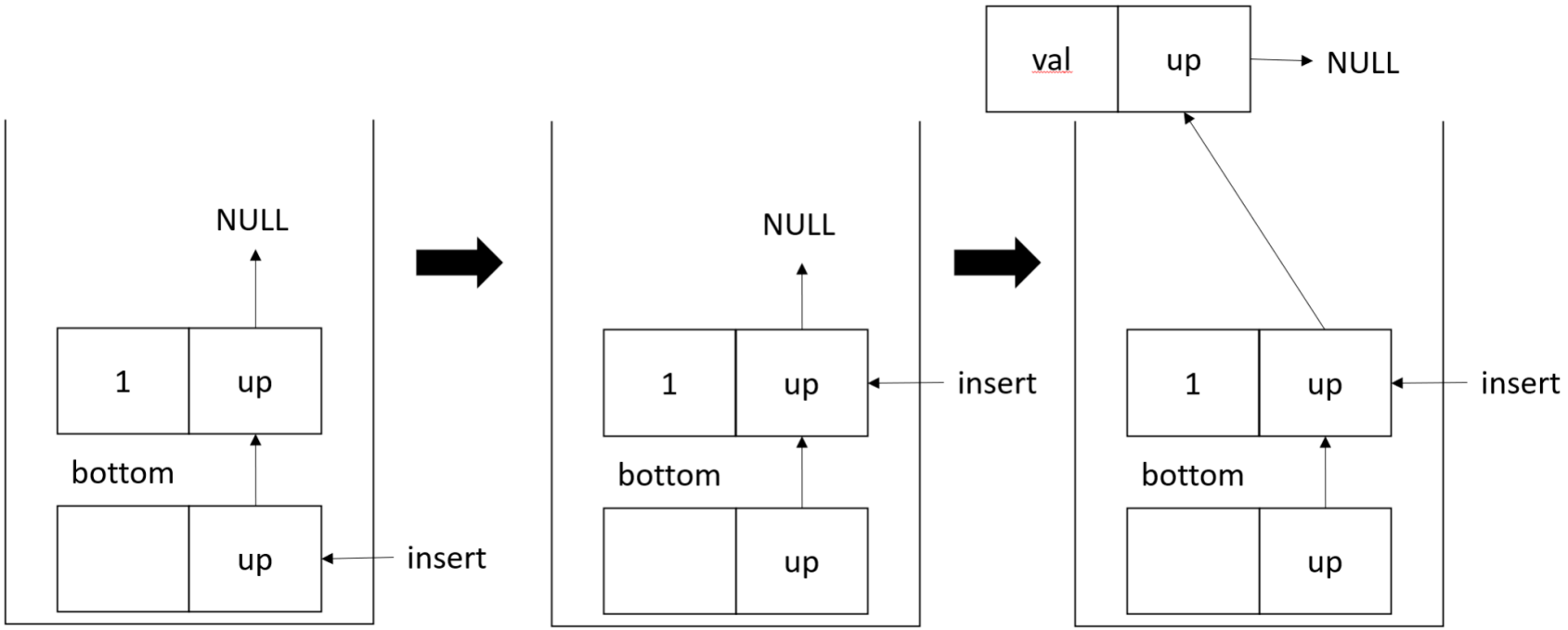
```
1 bool is_full(Stack *bottom) {
2     if(size(bottom) == MAX) {
3         return true;
4     }
5     return false;
6 }
```



- MAX 通常為事先訂好的常數。

實作 push

- 確認完堆疊沒滿後，找出此堆疊的最高點，在這個堆疊上再新增元素，進行串接。

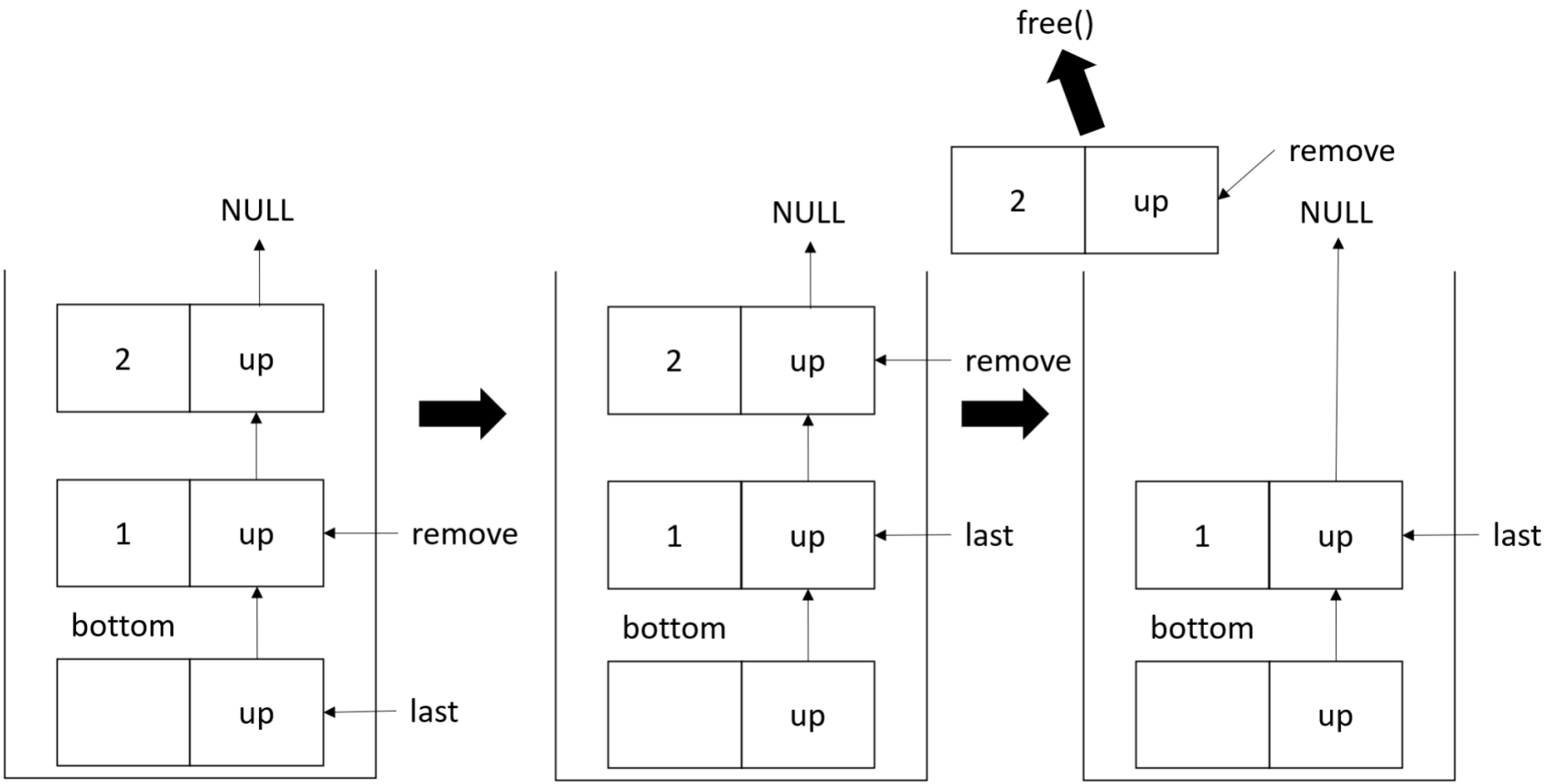


實作 push 程式碼

```
1  void push(Stack *bottom, int val) {
2      if(is_full(bottom)) {
3          printf("Stack is full.\n");
4          return;
5      }
6      Stack *insert = bottom;
7      while(insert->up != NULL){
8          insert = insert->up;
9      }
10     Stack *new = (Stack*)malloc(sizeof(Stack));
11     new->up = NULL;
12     new->val = val;
13     insert->up = new;
14     printf("insert %d.\n", val);
15 }
```

實作 pop

- 確認完堆疊非空後，往上找最高點，記錄次高點後，把最高點移除，次高點變成最高點。



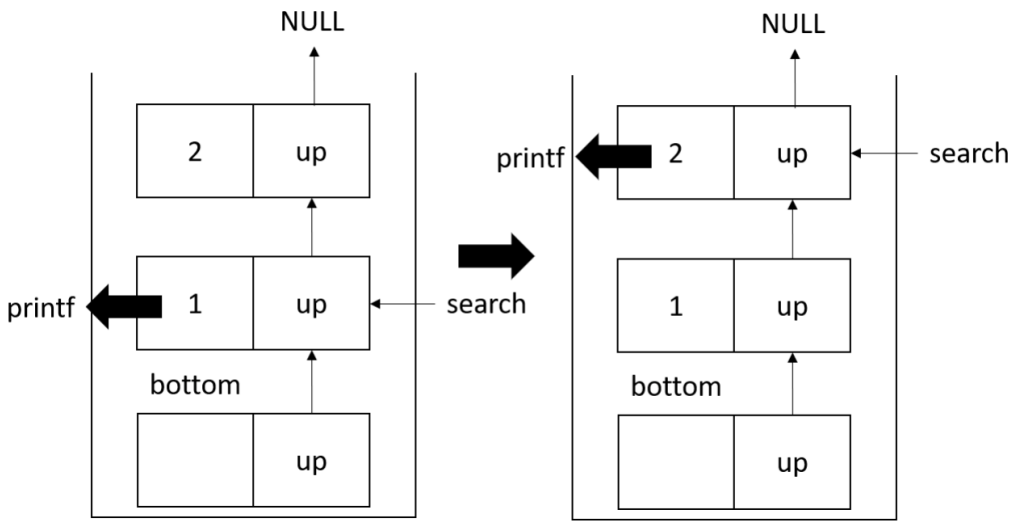
實作 pop 程式碼

```
1  int pop(Stack *bottom) {
2      if(is_empty(bottom)) {
3          printf("Stack is empty.\n");
4          return -1;
5      }
6      Stack *last = bottom;
7      Stack *remove = bottom->up;
8      while(remove->up != NULL){
9          last = remove;
10         remove = remove->up;
11     }
12     int remove_val = remove->val;
13     free(remove);
14     remove = NULL;
15     last->up = NULL;
16     return remove_val;
17 }
```

印出堆疊所有元素

- 由下往上找堆疊並一一印出。

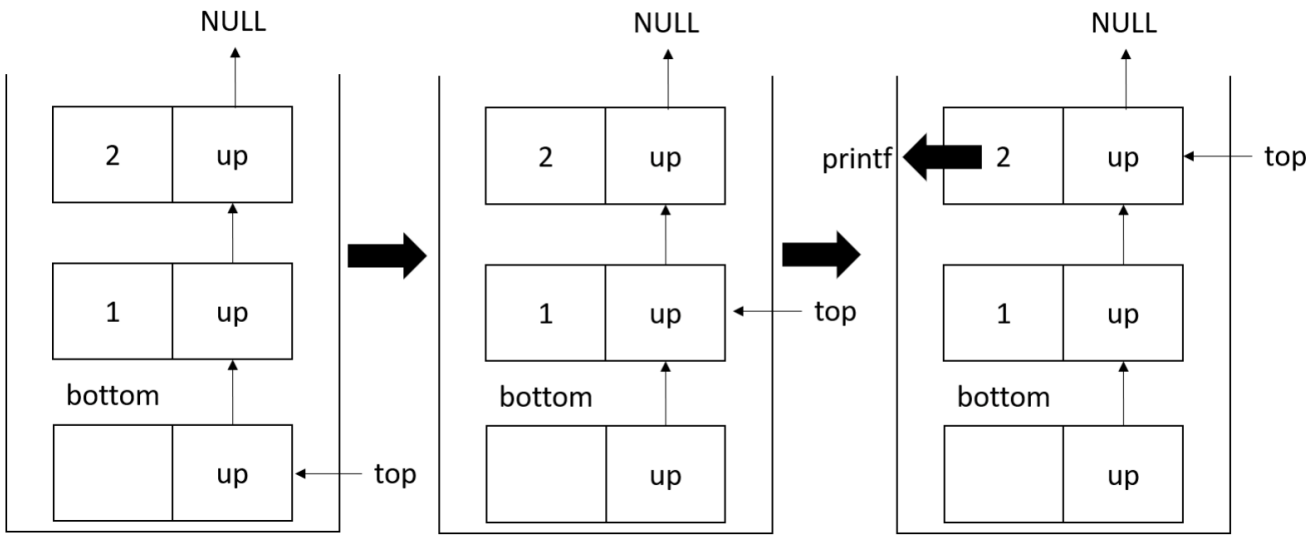
```
1 void print(Stack *bottom) {
2     Stack *search = bottom->up;
3     while(search != NULL) {
4         printf("%d ", search->val);
5         search = search->up;
6     }
7     printf("\n");
8 }
```



查看頂端的元素

- 由下往上找堆疊並印出最高點的資料 (但不從堆疊中取出)。

```
1 int now_top(Stack *bottom) {
2     Stack *top = bottom;
3     while(top->up != NULL) {
4         top = top->up;
5     }
6     return top->val;
7 }
```



表達式
Expression

表達式 (Expression)

- 在進入表達式前，我們先來認識一下甚麼叫前序 (Prefix Notation)、中序 (Infix Notation) 和後序 (Postfix Notation) 的算式，它們差別是在運算子 (operator，如 +、-、*、/) 和運算元 (operand，如 12、-3) 的放置順序，順序如下：

表達式順序	
前序	運算子 運算元 運算元 (+ab)
中序	運算元 運算子 運算元 (a+b)
後序	運算元 運算元 運算子 (ab+)

- 一般書寫常用的表達式為中序式，而編譯器常用的表達式為後序式。

運算子優先順序

- 若使用中序式的表達式，在運算時會需要考慮到運算子優先順序的問題。以下列舉 C 語言之運算子及其優先權。

表達式 - 運算子優先順序 (1)

符號	操作	優先權(越大越優先)[1]	關聯性
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

表達式 - 運算子優先順序 (2)

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right

表達式 - 運算子優先順序 (3)

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

註1. The precedence column is taken from Harbison and Steele.
註2. Postfix form
註3. Prefix form

表達式 - 中序式轉後序式

表達式 - 中序式轉後序式

- 假設我們今天有一個中序式子，我們想要把這個式子轉成後序式，我們就能運用 stack 的方法來實際做做看:

$$a + b * c \rightarrow a\ b\ c * +$$

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

表達式 - 後序式轉中序式

表達式 - 後序式轉中序式

- 而後序式轉中序式同樣也可以用 stack 來達成：

$$a\ b\ c\ *\ +\ ->\ a\ +\ b\ *\ c$$

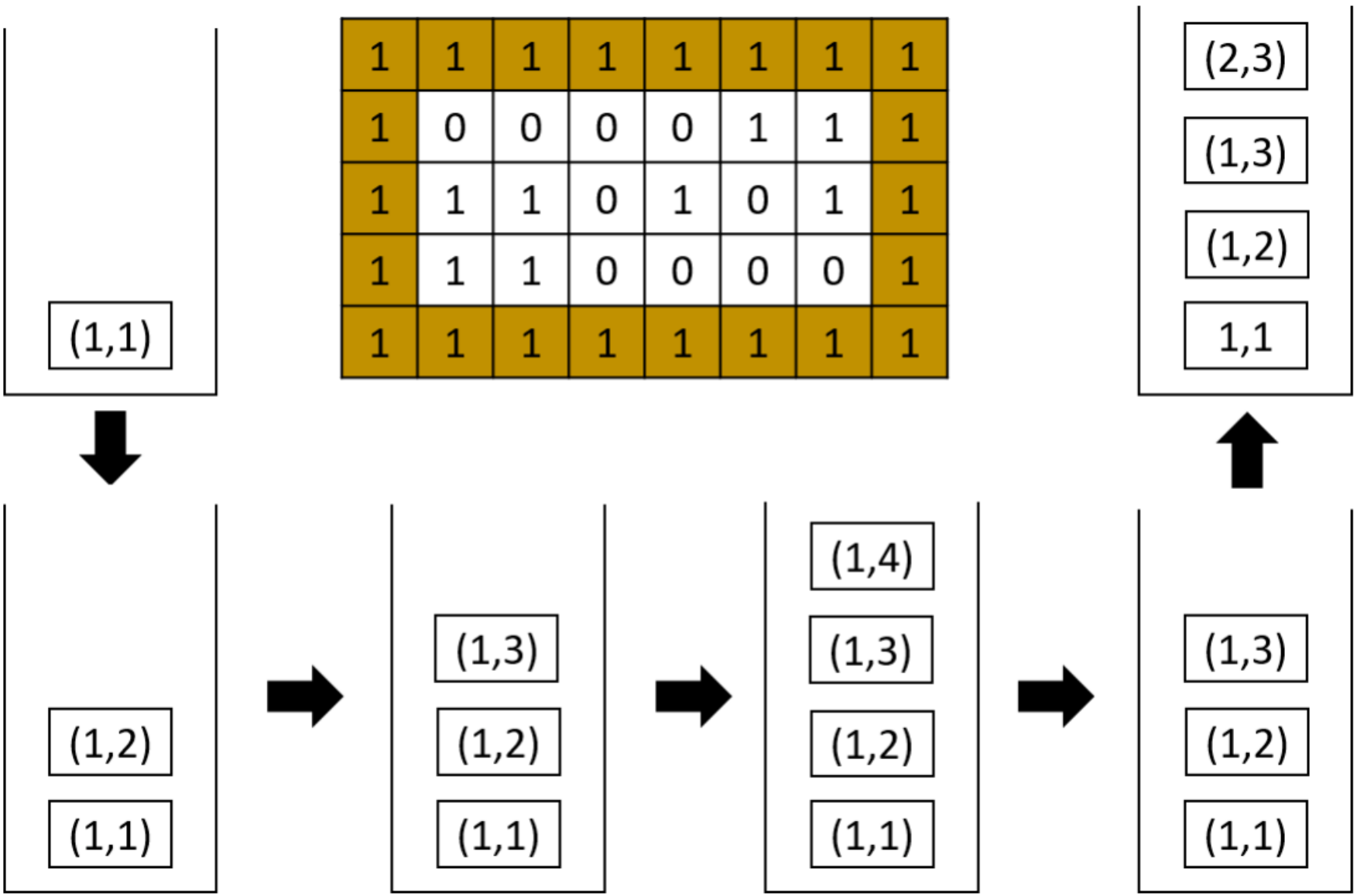
Token	Stack [0]	[1]	[2]	Top
a	a			0
b	a	b		1
c	a	b	c	2
*	a	(b*c)		1
+	a+(b*c)			0

Stack 例題

老鼠走迷宮 Maze

- 老鼠走迷宮是一個經典的 Stack 應用，我們可以從起點開始，把一個點丟入堆疊後，搜尋頂端堆疊附近的點可不可以通過，如果可以，則把附近的點再丟入堆疊上，重複此過程。
- 而要實作迷宮，為了防止找點時會超出邊界，我們通常都會在迷宮的外面再圍一座牆 (棕色)，如下頁圖所示。
- 請試著實作程式，輸入迷宮地圖以及起終點的座標，輸出起終點是否相通。

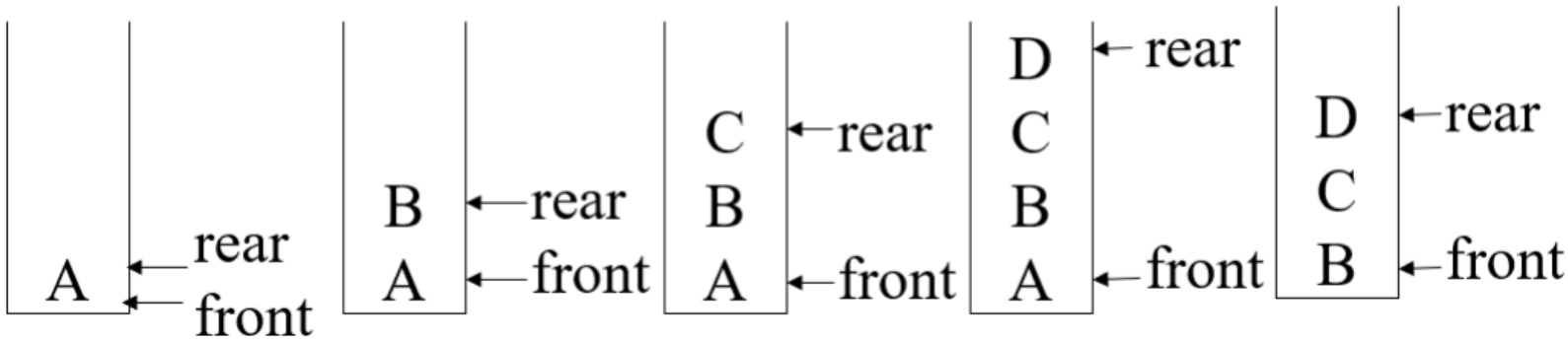
例題 - 老鼠走迷宮 Maze



佇列
Queue

佇列 (Queue)

- 而 Queue 的特性為 FIFO(First In and First Out)，就像排隊，有分排頭跟排尾，過程中會一直有人來排隊，也會有人滿足需求而離開隊伍。



佇列的抽象資料型態

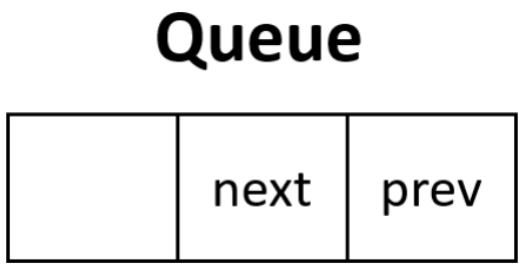
- Data：一個線性串列，資料的進入點為尾端，移出點為前端。
- Operation：分為會改變佇列狀態的 enqueue 和 dequeue，以及觀察佇列狀態的 is_empty、print 和 size。
  - enqueue：把元素插入至佇列尾端。
  - dequeue：把佇列最前端的元素拿出來。
  - is_empty：判斷佇列是否為空。
  - print：把佇列中的元素從全部印出來。
  - size：佇列長度。

以鏈結串列實作佇列
Implement Queue with Linked Lists

建立佇列

- 佇列是一種 ADT，因此不限定實作所使用的資料結構。
- 在這裡我們選用雙向鏈結串列來實作佇列，val 為每一個元素的值，而 next 則指向後面的元素，prev 則是指向前面的元素。

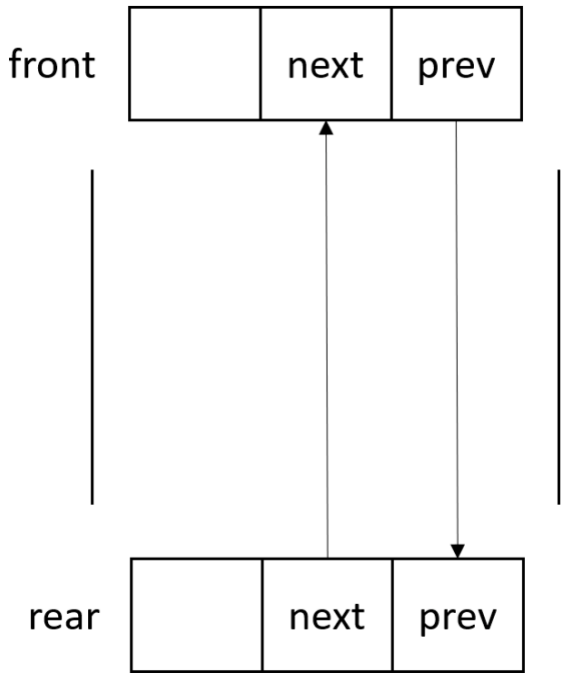
```
1 typedef struct Queue {
2     int val;
3     struct Queue *next;
4     struct Queue *prev;
5 } Queue;
```



建立佇列

- 宣告指標 rear 及 front 並將其指向最尾端與最前端，並將其串接。此二鏈結串列元素不儲存數值。

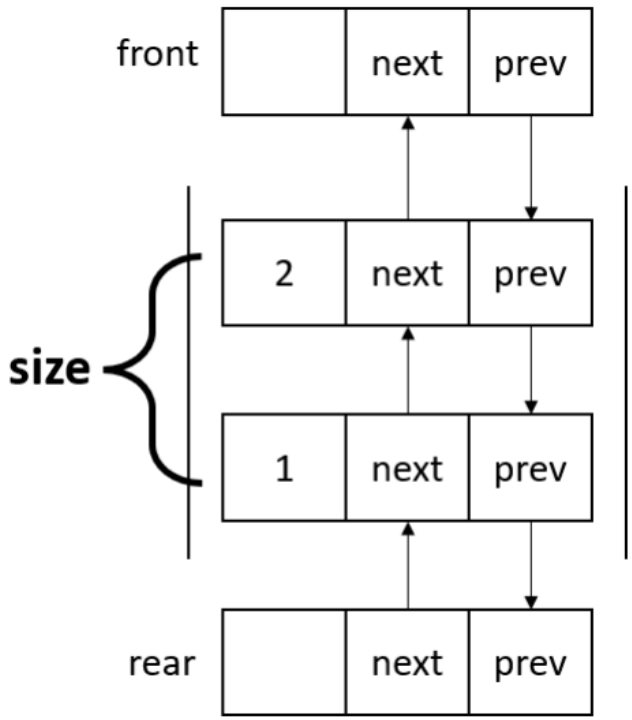
```
1 int main() {
2     Queue *rear = (Queue*)malloc(sizeof(Queue));
3     Queue *front = (Queue*)malloc(sizeof(Queue));
4     rear->next = front;
5     front->prev = rear;
6 }
```



計算長度

- 知道佇列長度的方法就是從最尾端的元素往前數到前端並計數。

```
1  int size(Queue *rear, Queue *front)
       {
2      int count = 0;
3      Queue *search = rear->next;
4      while(search != front) {
5          count++;
6          search = search->next;
7      }
8      return count;
9  }
```

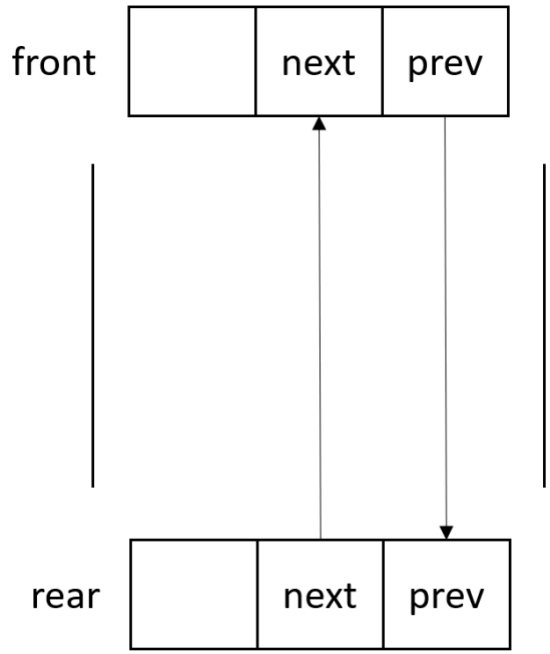


以鏈結串列實作佇列

判斷佇列為空

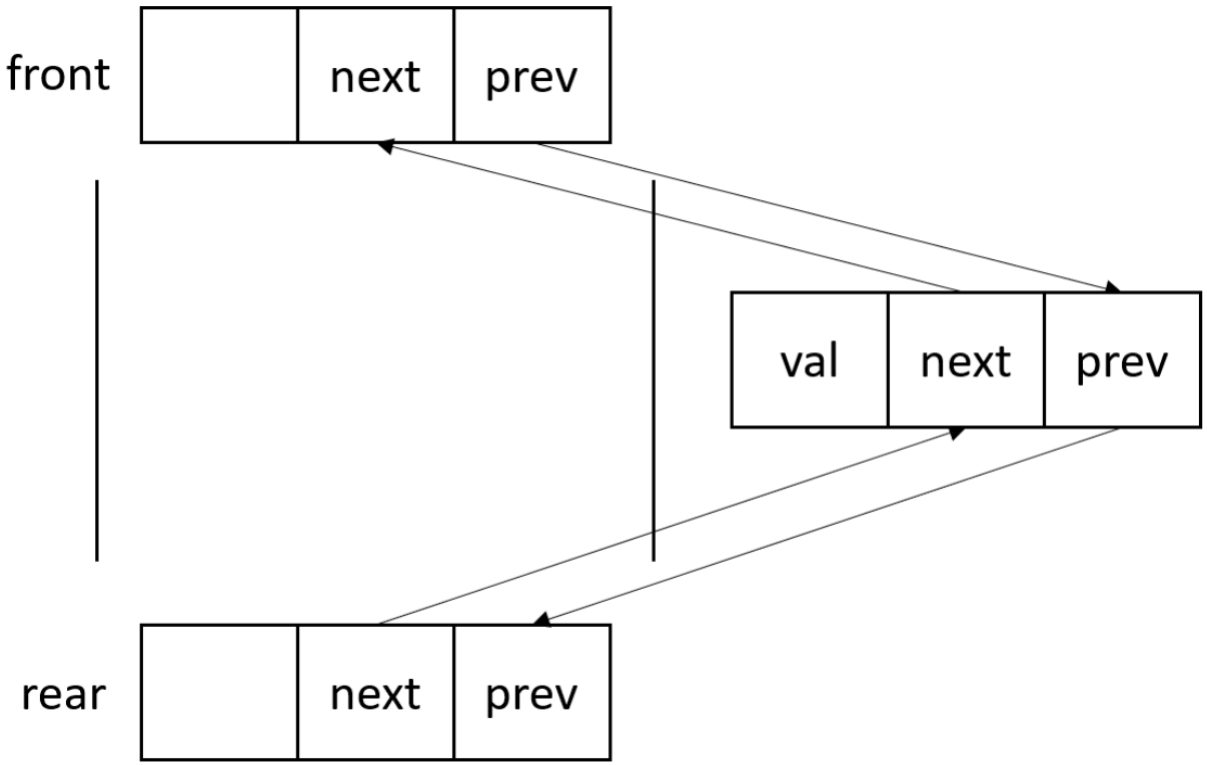
- 當 rear 的前面指向的是 front 就代表序列裡面沒有任何值，因此為空。

```
1 bool is_empty(Queue *rear, Queue *front) {
2     if(rear->next == front) {
3         return true;
4     }
5     return false;
6 }
```



實作 enqueue

- 新增時從最尾端進入，並把前後兩端彼此串聯。



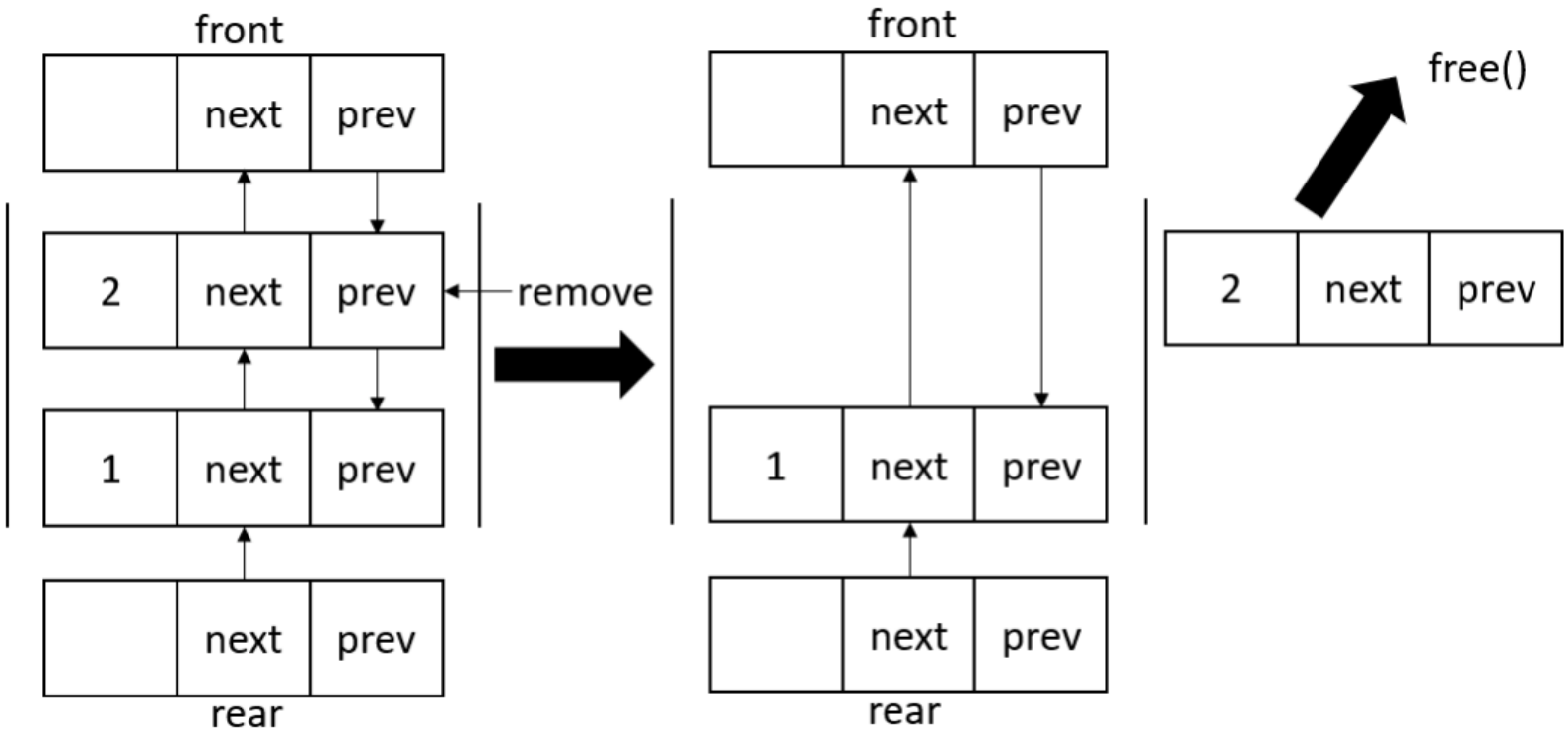
以鏈結串列實作佇列

實作 enqueue 程式碼

```
1 void enqueue(Queue *rear, Queue *front, int val) {
2     Queue *new = (Queue*)malloc(sizeof(Queue));
3     new->val = val;
4     new->next = rear->next;
5     rear->next = new;
6     new->prev = rear;
7     new->next->prev = new;
8     printf("Enqueued %d.\n", val);
9 }
```

實作 dequeue

- 確認佇列不為空後，把前端指向的空間移除並指向第二順位的元素。

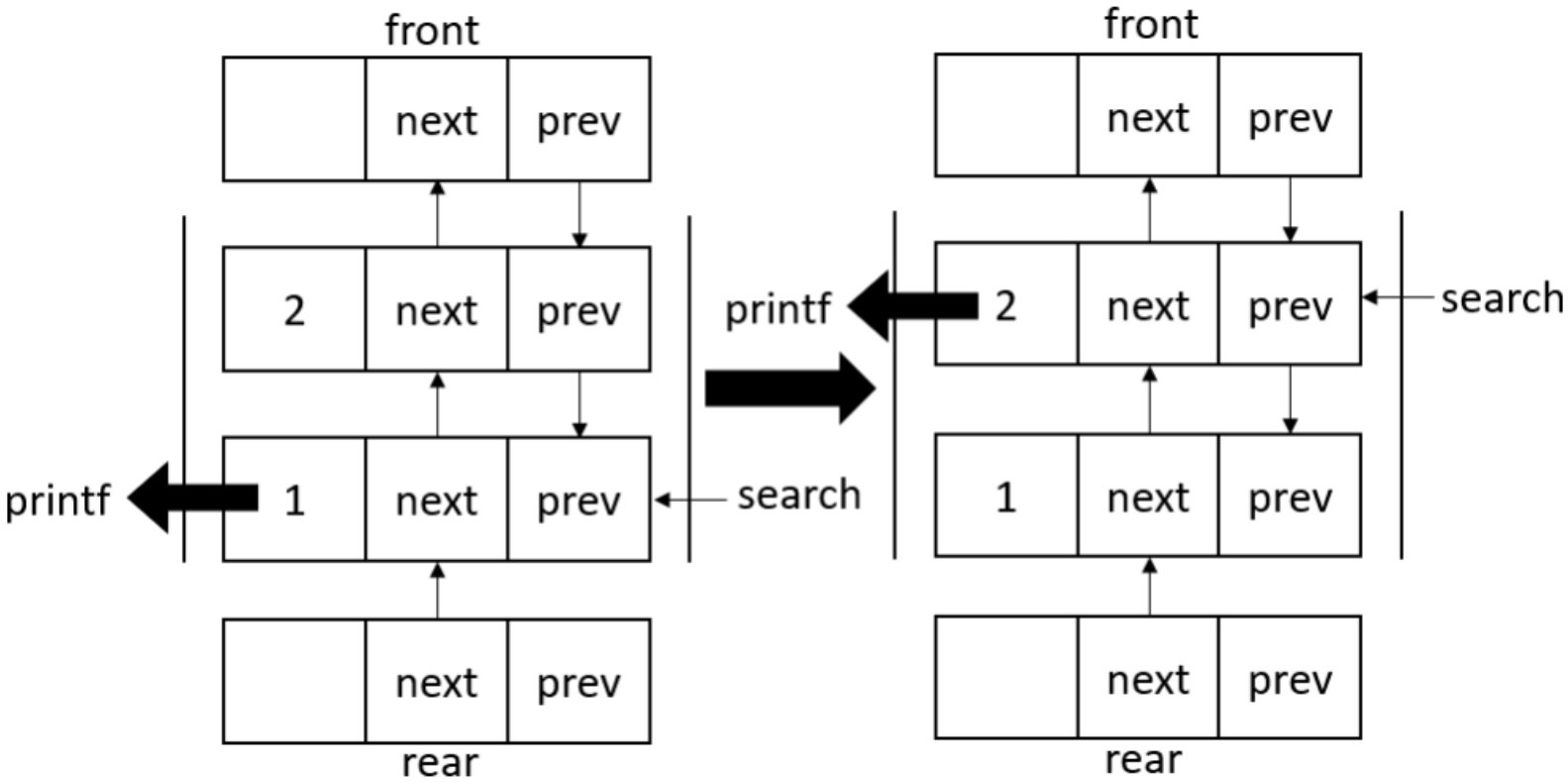


實作 dequeue 程式碼

```
1  int enqueue(Queue *rear, Queue *front) {
2      if (is_empty(rear,front)) {
3          printf("Queue is empty.");
4          return -1;
5      }
6      Queue *remove = front->prev;
7      int val = remove->val;
8      front->prev = remove->prev;
9      front->prev->next = front;
10     free(remove);
11     remove = NULL;
12     return val;
13 }
```

印出佇列所有元素

- 從佇列最尾端偵測到最前端，並把每一個元素印出。



以鏈結串列實作佇列

實作佇列 print 程式碼

```
1 void print(Queue *rear, Queue *front) {
2     Queue *search = rear->next;
3     while (search != front) {
4         printf("%d ", search->val);
5         search = search->next;
6     }
7     printf("\n");
8 }
```

補充

- 以陣列實作堆疊 (Implement Stack with Arrays)
- 以陣列實作循序佇列 (Implement sequential Queue with Arrays)
- 以陣列實作循環佇列 (Implement Circular Queue with Arrays)

以陣列實作堆疊
Implement Stack with Arrays

以陣列實作堆疊 (Implement Stack with Arrays)

- 前面我們用了鏈結串列來實作堆疊，但是在找頂層元素時卻會需要把所有鏈結串列都走過一次，因此接下來我們會用陣列實作堆疊。

以陣列實作堆疊 (Implement Stack with Arrays)

建立堆疊

- 宣告 top 和 capacity 為全域變數，top 為堆疊的頂端，capacity 為此堆疊最大的高度。
- 先初始化建立一個資料型別為整數且長度為 1 的陣列空間。

```
1 int top = 0;
2 int capacity = 1;
3 int main() {
4     int *stack = (int*)malloc(capacity * sizeof(int));
5 }
```

以陣列實作堆疊 (Implement Stack with Arrays)

size

- 堆疊的高度為頂端所在的位置。

```
1 bool size(){
2     return top;
3 }
```

以陣列實作堆疊 (Implement Stack with Arrays)

is_full/is_empty

- 用 top 可以偵測堆疊的空及滿。

```
1 bool is_full(){
2     return top >= capacity;
3 }
4 bool is_empty(){
5     return top == 0;
6 }
```

以陣列實作堆疊 (Implement Stack with Arrays)

stack_full

- 如果數量會超過 capacity 的話就用 realloc 來調整堆疊的大小，因為有可能會有分配失敗的問題，所以不能直接對 stack 做 realloc。

```
1 void stack_full(int *stack){
2     int *new_stack = (int*)realloc(stack,capacity*2*sizeof(int));
3     if (!new_stack) {   //分配失敗
4         perror("overflow");
5     }
6     stack = new_stack;
7 }
```

以陣列實作堆疊 (Implement Stack with Arrays)

push

- 如果堆疊中的數量大於 capacity 的話就用 stack_full 來增加 capacity 的大小，並把 value 塞進堆疊裡。

```
1 void push(int *stack, int value){
2     if (is_full(stack)) {
3         stack_full(stack);
4     }
5     stack[++top] = value;
6 }
```

以陣列實作堆疊 (Implement Stack with Arrays)

pop

- 如果堆疊為空的話就回傳負數，若非則回傳頂端元素並把頂端位置-1。

```
1 int pop(int *stack){
2     if (is_empty(stack)) {
3         printf("The stack is empty.\n");
4         return -999;
5     }
6     return stack[top--];
7 }
```

以陣列實作堆疊 (Implement Stack with Arrays)

now_top

- 回傳 stack 當前頂端的元素。

```
1 int now_top(int *stack){
2     return stack[top];
3 }
```

以陣列實作佇列
Implement Queue with Arrays

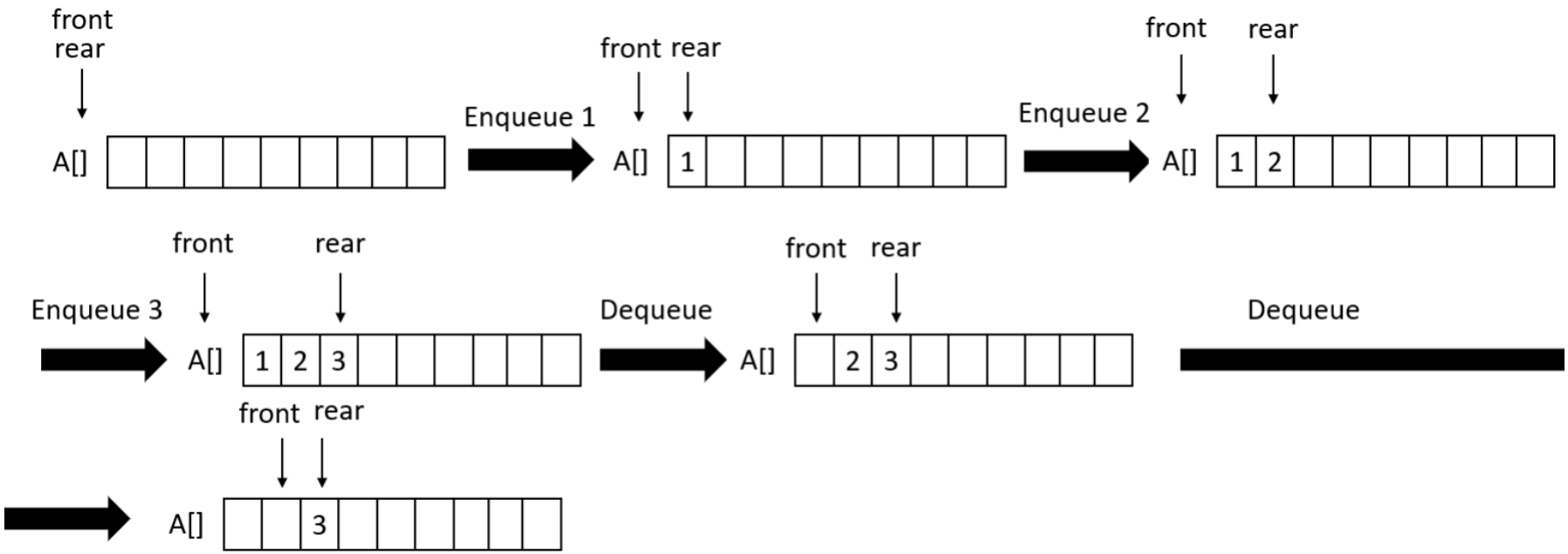
以陣列實作佇列 (Implement Queue with Arrays)

- 使用陣列，我們可以用以下兩種方式來實做佇列：
  - 循序佇列 (Sequential Queue)：
    - 簡單易實作
  - 循環佇列 (Circular Queue)：
    - 節省記憶體

以陣列實作循序佇列
Implement Sequential Queue with Arrays

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

- 使用順序陣列時要先宣告一個陣列放置元素，並宣告兩個變數來指出 front 及 rear 的位置，rear 一定大於 front。



以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

建立佇列

- 初始化並宣告三個全域變數，rear 為佇列的尾端，front 為佇列的前端，capacity 為此佇列最大的長度。

```
1 int rear = -1;
2 int front = -1;
3 int capacity = 1;
4 int main(){
5     int *queue = (int*)malloc(capacity*sizeof(int));
6 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

size

- 佇列的長度為尾端所在的位置減去前端所在的位置。

```
1 int size(){
2     int rear - front;
3 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

is_full/is_empty

- 利用 rear 及 front 可以偵測佇列的空及滿。

```
1 bool is_full(){
2     return rear >= capacity;
3 }
4 bool is_empty(){
5     return rear == front;
6 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

queue_full

- 如果數量會超過 capacity 的話就用 realloc 來調整堆疊的大小，因為有可能會有分配失敗的問題，所以不能直接對 queue 做 realloc。

```
1 void queue_full(int *queue){
2     int *new_queue = (int*)realloc(queue,capacity*2*sizeof(int));
3     if (!new_queue) {   //分配失敗
4         perror("overflow");
5     }
6     queue = new_queue;
7 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

enqueue

- 如果佇列中的數量大於 capacity 的話就用 queue_full 來增加 capacity 的大小，並把 value 插進佇列裡。

```
1 void enqueue(int *queue, int value){
2     if (is_full()) {
3         queue_full(queue);
4     }
5     queue[++rear] = value;
6 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

dequeue

- 如果佇列為空的話就回傳負數，若非則回傳前端元素並把前端位置+1。

```
1 int dequeue(int *queue){
2     if (is_empty()) {
3         printf("The queue is empty.\n");
4         return -999;
5     }
6     return queue[++front];
7 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

now_front

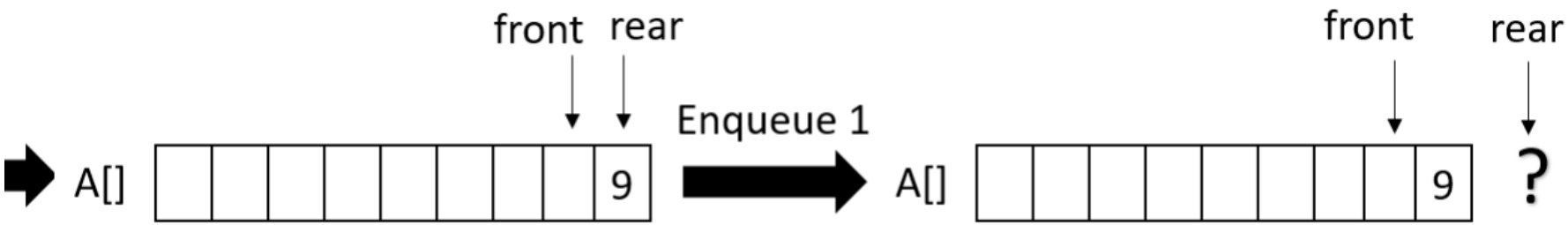
- 回傳 queue 前端的元素。

```
1 int now_front(int *queue){
2     return queue[front+1];
3 }
```

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

產生的問題

- 雖然實作上非常容易，但也非常容易會遇到以下狀況：



- 從上面的圖片我們可以看到：
  - front 前面的空間全部被浪費掉了。
  - rear 一指出陣列外面就會出現假溢出問題 (False Overflow)。

以陣列實作循序佇列 (Implement Sequential Queue with Arrays)

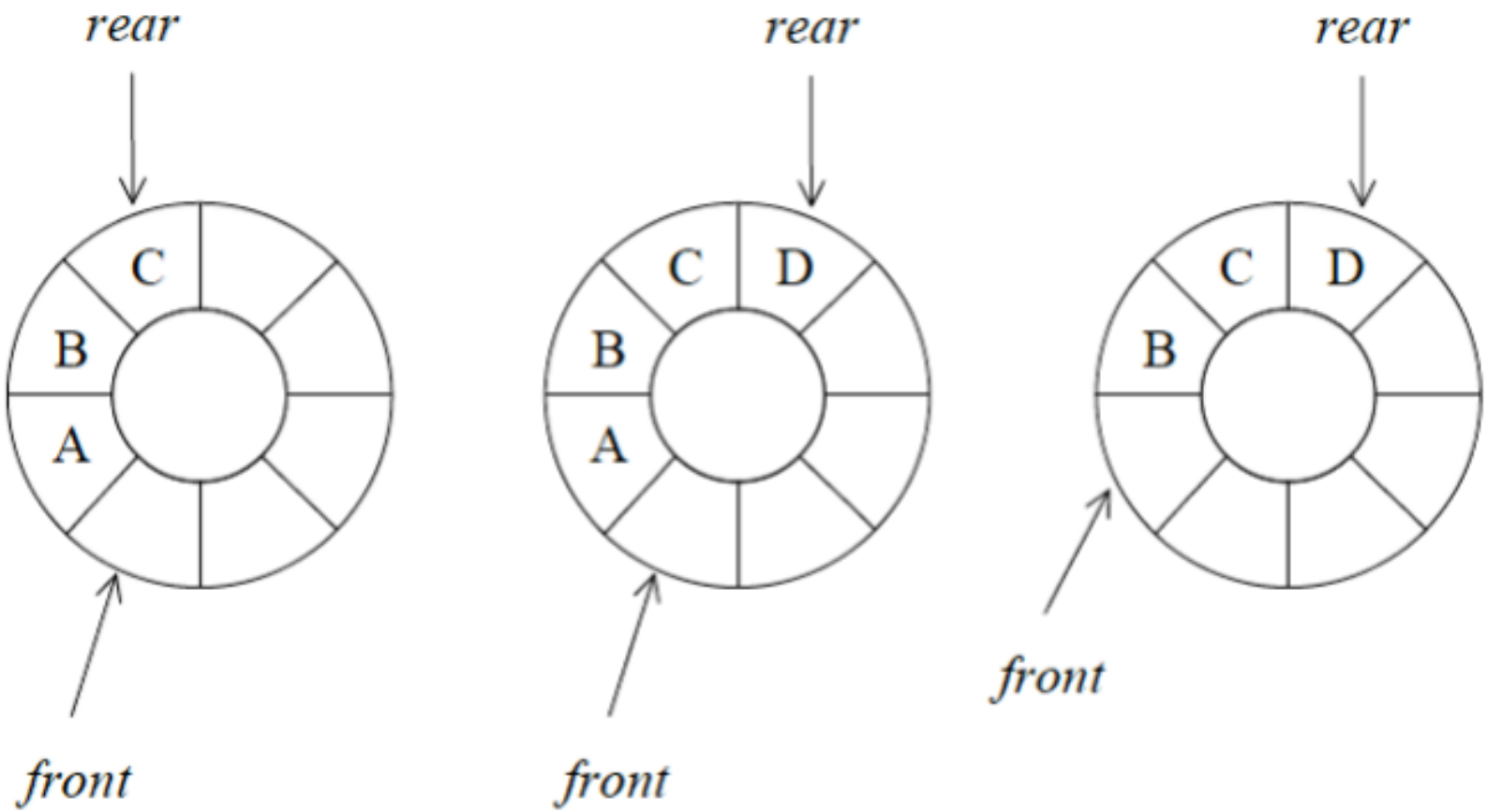
問題解決辦法

- 關於浪費空間及假溢出的問題我們可以透過 realloc() 這個函式來重新分配記憶體空間，新增更大的空間來繼續裝元素。
- 不過重新分配記憶體會造成所需要的運算成本及時間的浪費，所以在佇列陣列的實作上我們會比較常使用循環佇列。

以陣列實作循環佇列
Implement Circular Queue with Arrays

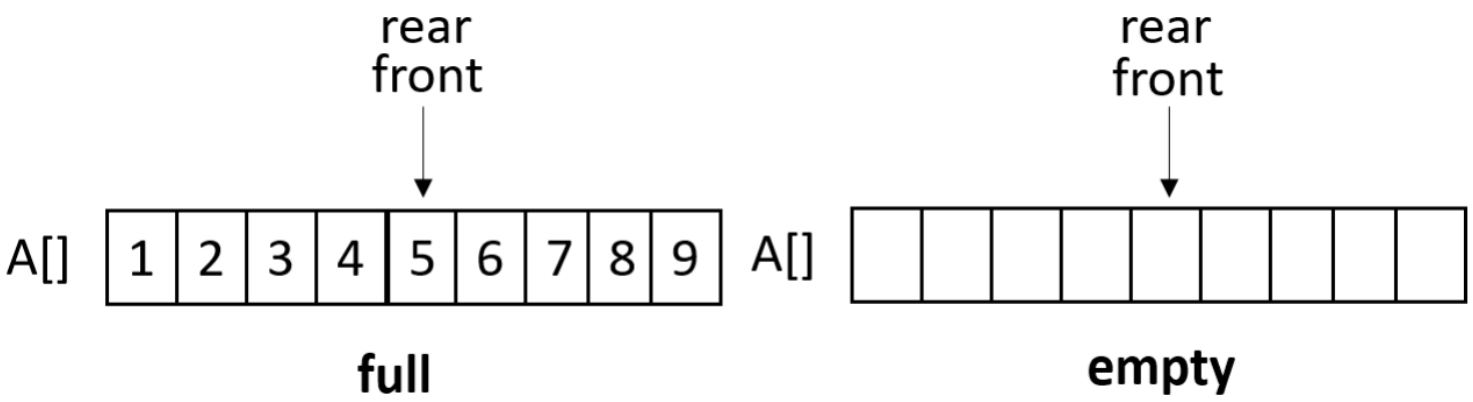
以陣列實作循環佇列 (Implement Circular Queue with Arrays)

- 使用順序陣列時要先宣告一個陣列放置元素，並宣告兩個變數來指出 front 及 rear 的位置，rear 一定大於 front。



以陣列實作循環佇列 (Implement Circular Queue with Arrays)

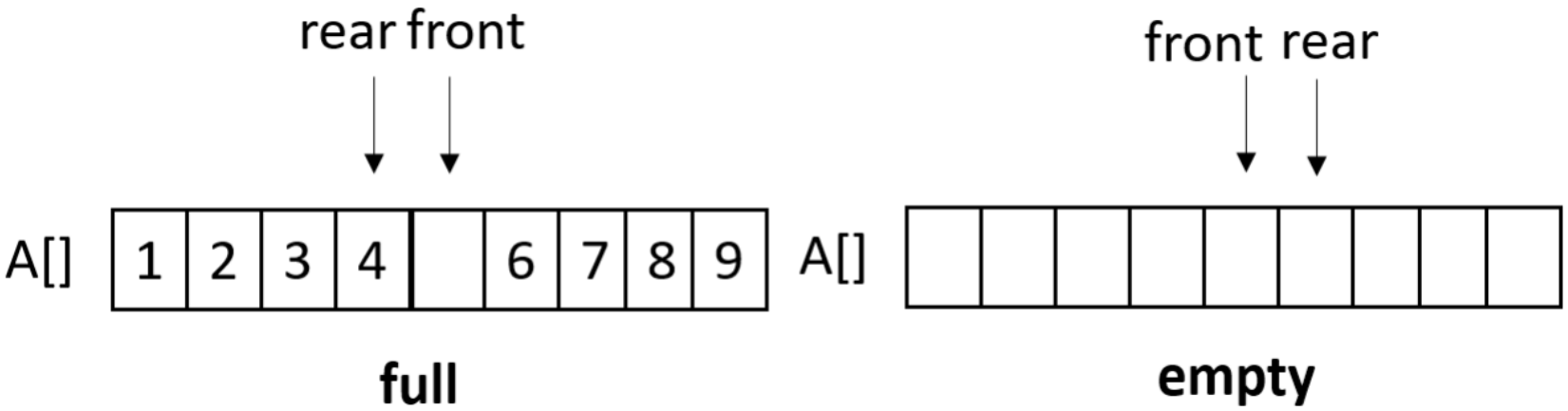
- 而循環序列也式會遇到一些問題，例如：
  1. 當 rear 和 front 重合時無法判斷柱列是滿的還是空的。



  2. 因為 front 跟 rear 會從尾巴跑到前面所以不知道誰大誰小，有沒有通用的公式可以算長度和序列是否已滿呢?

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

- 首先，為了確保不會發生“當 rear 和 front 重合時無法判斷柱列是滿的還是空的”，我們把填滿序列的條件設定成陣列長度-1，這樣在 rear==front 的時候即可確定這個序列是空的。



- 而要確定序列填滿於否的公式為 $(\text{rear} + 1) \% \text{capacity} == \text{front}$

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

- 而計算序列長度的公式可以用兩個方向討論並整合起來。
  - 當 rear $>$ front 時，長度為 rear $-$ front
  - 當 front $>$ rear 時，長度為 rear $+$ capacity $-$ front

- 而我們綜合了以上兩種狀況後就可以得出一個通用公式：

$$\text{length} = (\text{rear} - \text{front} + \text{capacity}) \ \% \ \text{capacity}$$

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

建立循環佇列

- 初始化並宣告三個全域變數，rear 為佇列的尾端，front 為佇列的前端，capacity 為此佇列最大的長度。

```
1 #define MAX_LENGTH = 5
2 int rear = -1;
3 int front = -1;
4 int capacity = MAX_LENGTH;
5 int main(){
6     int *queue = (int*)malloc(capacity*sizeof(int));
7 }
```

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

size

- 佇列的長度為 length = (rear − front + capacity) % capacity。

```
1 int size(){
2     return (rear - front + capacity) % capacity;
3 }
```

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

is_full/is_empty

- 佇列滿的條件設成總長度-1 可以避免跟佇列空的條件重疊。

```
1 bool is_full(){
2     return size() >= capacity - 1;
3 }
4 bool is_empty(){
5     return rear == front;
6 }
```

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

enqueue

- 如果佇列中的數量大於 capacity 的話就回傳滿的訊息，若尾端超出空間的話就回到起點，並把 value 插進佇列裡。

```
1 void enqueue(int *queue, int value){
2     if (is_full()) {
3         printf("The queue is full!\n");
4         return;
5     }
6     queue[(rear+1)>=capacity?0:++rear] = value;
7 }
```

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

dequeue

- 如果佇列為空的話就回傳負數，若非則回傳前端元素並把前端位置+1，而前端超出空間的話就回到起點。

```
1 int dequeue(int *queue){
2     if (is_empty()) {
3         printf("The queue is empty.\n");
4         return -999;
5     }
6     return queue[(front+1)>=capacity?0:++front];
7 }
```

以陣列實作循環佇列 (Implement Circular Queue with Arrays)

now_front

- 回傳 queue 前端的元素。

```
1 int now_front(int *queue){
2     return queue[(front+1)>=capacity?0:front+1];
3 }
```

Q & A