資料結構

C演算法

佇列

Queue

佇列 Queue

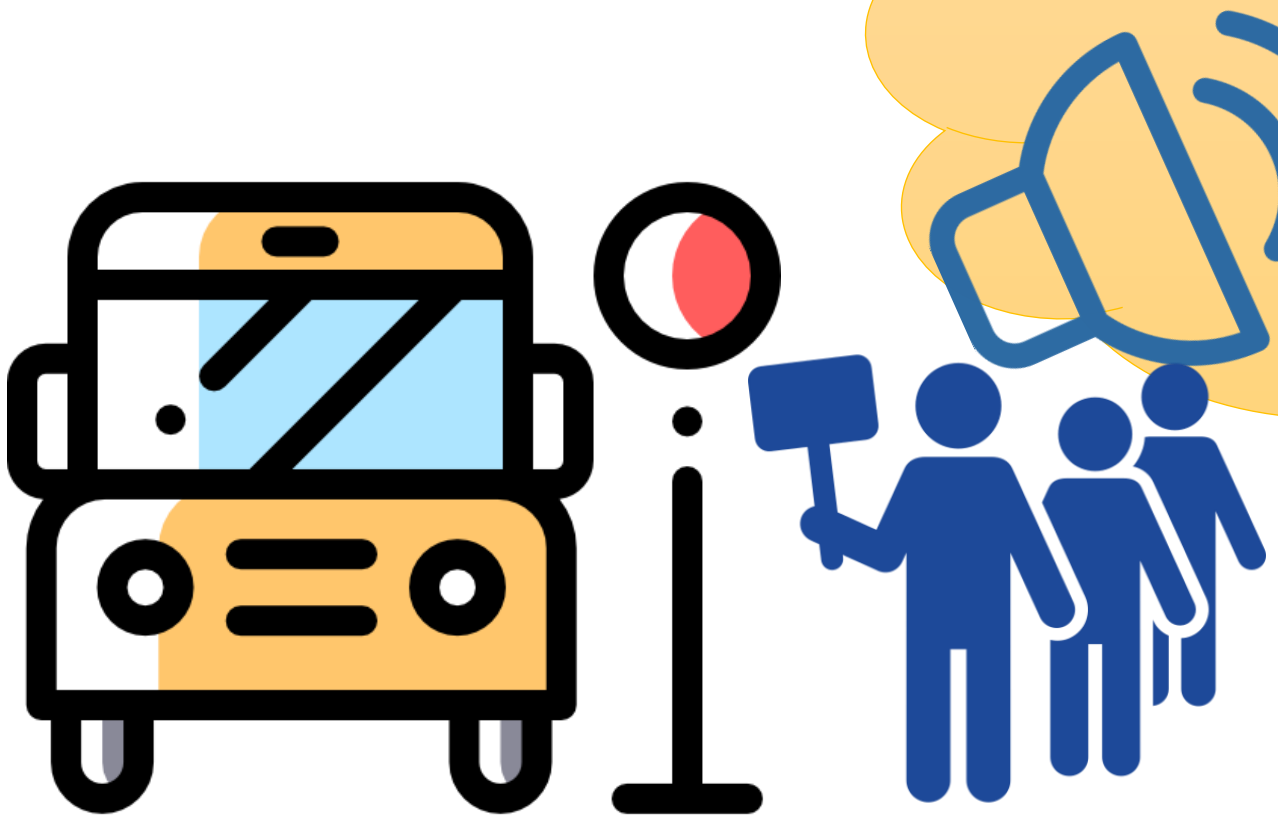
三個等著搭車的乘客，
該如何依序上車呢？


先排隊的
先上車阿！

這就是『佇列 queue』

一邊出口一邊入口的容器
中，先加入的要先取出

就是『佇列』




佇列特性

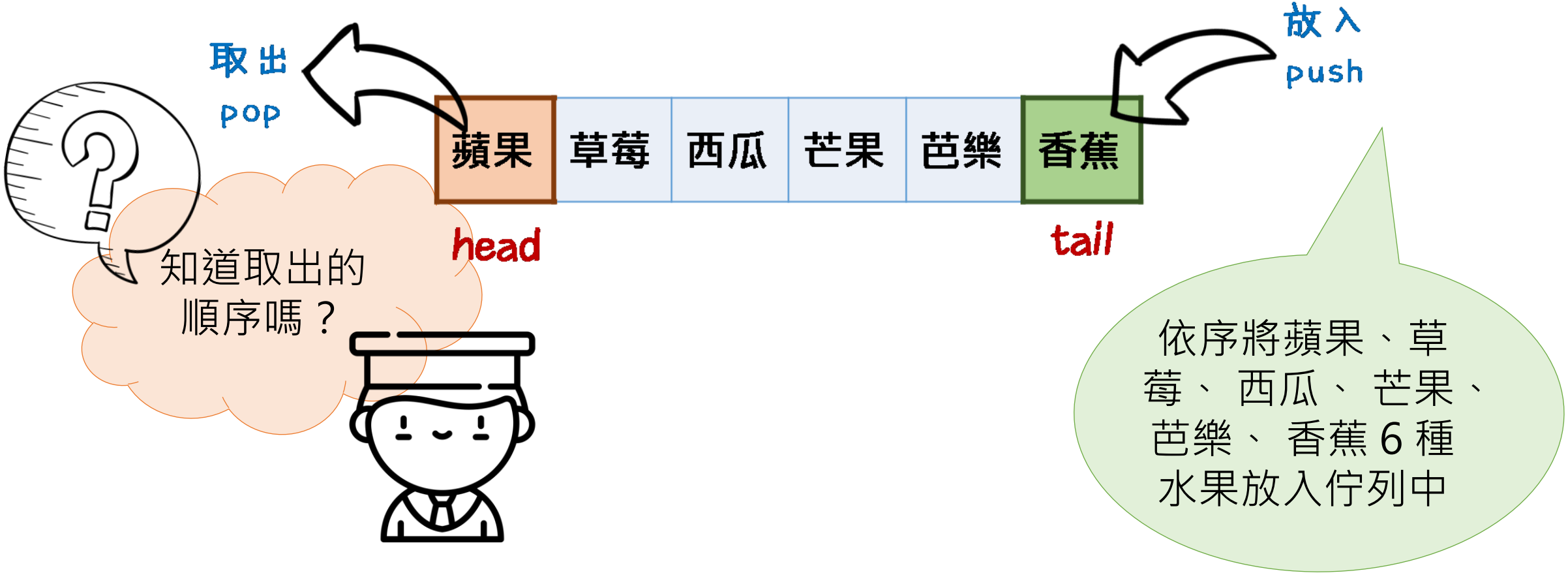
佇列結構

- 是一種串列

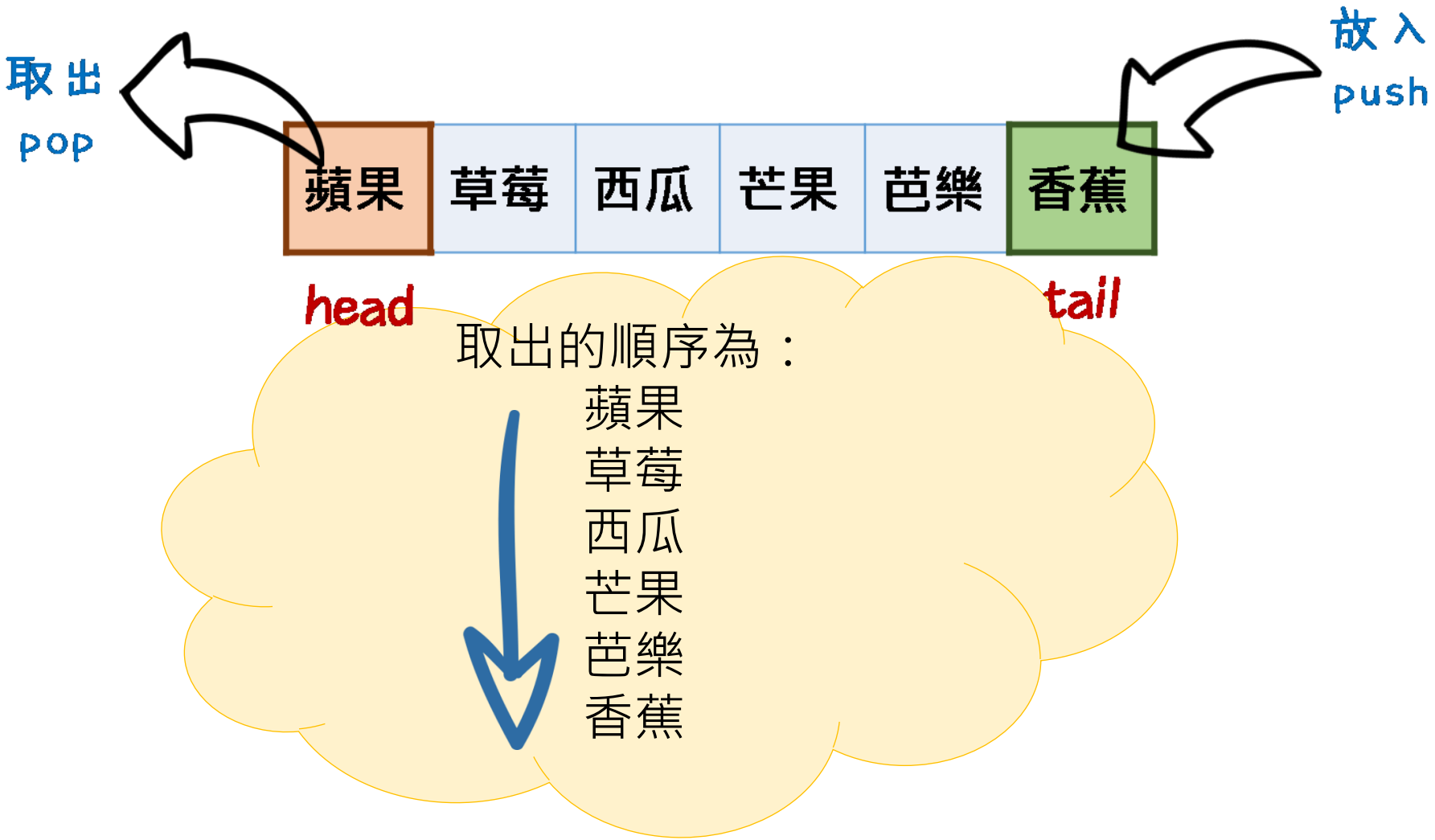
- 分別有**一個出口，一個入口**
  - 入口端稱為 **tail**，用來**新增(放入)**資料
  - 出口端稱為 **head** ，用來**移除(取出)**資料

- 具有 **先進先出(First-In-First-Out, FIFO)** 與 **後進後出(Last-In-Last-Out, LILO)** 的特性

FIFO 與 LILO (1)


取出
pop
放入
push
蘋果
草莓
西瓜
芒果
芭樂
香蕉
head
tail
知道取出的順序嗎？
依序將蘋果、草莓、西瓜、芒果、芭樂、香蕉 6 種水果放入佇列中

FIFO 與 LILO (2)



FIFO 與 LILO (3)

	新增的順序	移除的順序
1	蘋果	蘋果
2	草莓	草莓
3	西瓜	西瓜
4	芒果	芒果
5	芭樂	芭樂
6	香蕉	香蕉

最先放入的會最先移除，
此種特性稱為**先進先出**
(First-In-First-Out, FIFO)

最後放入的會最後移除，
此種特性稱為**後進後出**
(Last-In-Last-Out, LILO)

佇列的實作

- 兩種資料結構可以實作佇列
  - 陣列
    - 優點：資料操作存取快速
    - 缺點：佇列空間是固定的
  - 鏈結串列
    - 優點：佇列內可存放的資料數不受限制
    - 缺點：資料操作存取較慢


用陣列
實作佇列

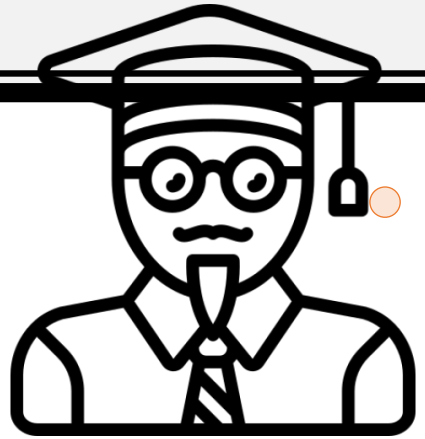
用陣列實作佇列 – 三大元素

- 基本元素
  1. **佇列陣列**
  2. **head 變數**
     - 記錄目前佇列的出口位置
     - 預設初始值為 -1
  3. **tail 變數**
     - 記錄目前佇列的入口位置
     - 預設初始值為 -1

- 輔助元素
  - **陣列大小**
    - 可以直接用變數儲存陣列大小
    - 也可以透過 sizeof 計算陣列大小，所以這個變數可以宣告也可以不宣告。
      - `sizeof(stackList)/sizeof(stackList[0])`

用陣列實作佇列 – 三大元素

```
typedef struct queueStruct {
  int *queueList;
  int head;
  int tail;
  int size;
} queueType;
```



由於佇列有多個基本元素，因此建議使用 **struct (結構)** 儲存佇列資料

用陣列實作佇列 – 五大操作

createQueue	建立佇列
enqueue	新增一個項目
dequeue	移除一個項目
isEmpty	判斷佇列內是否還有資料
isFull	判斷佇列內的資料是否滿了

用陣列實作佇列 – 建立新佇列

```
queueType* createQueue(queueType *queue, int size)
{
  queue = (queueType*)malloc(sizeof(queueType));
  newQueue->queueList = (int *)malloc(sizeof(int) * size);

  newQueue->head = -1;
  newQueue->tail = -1;

  newQueue->size = size;
  return newQueue;
}
```

head 與 tail 分別掌控佇列的出入口位置，一開始需要先將他們初始化為 -1

用陣列實作佇列 – 新增項目

```
int enqueue(queueType *queue, int value)
{
  if (isFull(queue)){
    return -1;
  }
  queue->tail++;
  queue->queueList[queue->tail] = value;
  printf("head:%d, tail: %d, size: %d\n“,queue->head,queue->tail,queue->size);
  return queue->tail;
}
```

1

2

新增項目時需要先檢查
佇列是否已經滿了

tail 是控制入口位置，因此
新增資料的位置由 tail 決定

用陣列實作佇列 – 移除一個項目

```
int dequeue(queueType *queue)
{
  if (isEmpty(queue)){
    return -1;
  }
  queue->head++;
  printf("value: %d, head: %d, tail: %d, size: %d\n", queue->queueList[queue->head], queue->head, queue->tail, queue->size);
  queue->queueList[queue->head] = 0;
  return queue->head;
}
```

1

2

移除項目時需要先檢查佇列內是否有資料

head 是控制出口位置，因此要移除哪個位置的資料是由 head 決定

用陣列實作佇列 – 佇列內是否還有資料

```
int isEmpty(queueType *queue)
{
  return (queue->head == queue->tail) ? 1 : 0;
}
```

當 head 與 tail 是同一個值的時後，就代表佇列已經空了

用陣列實作佇列 – 佇列內是否滿了

```
int isFull(queueType *queue)
{
  return (queue->tail >= queue->size-1) ? 1 : 0;
}
```

若 tail 值大於等於佇列預設空間數-1，就代表佇列已經滿了

用陣列實作佇列

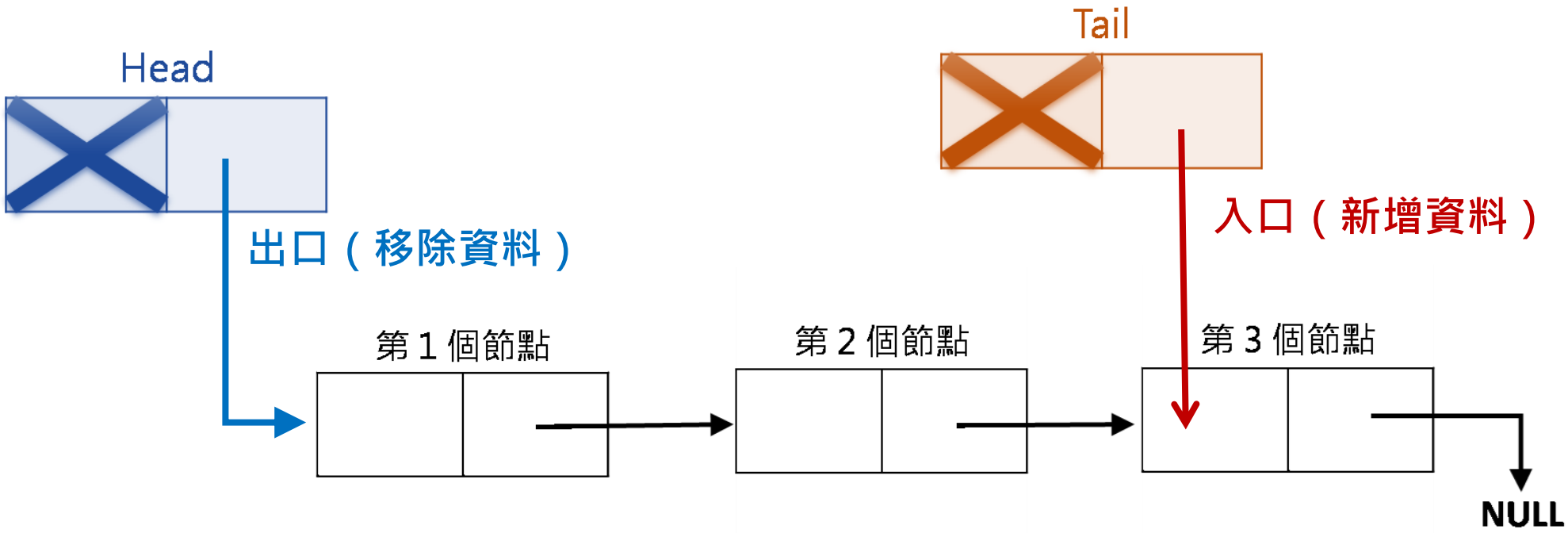
- 三大元素
  - 佇列陣列
  - head 變數
  - tail 變數

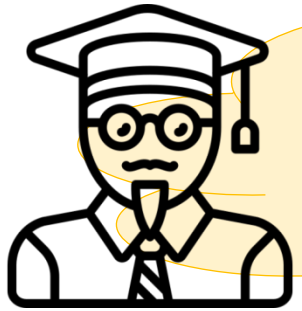
- 五大操作

createQueue	建立佇列	**head = -1, tail = -1**
enqueue	新增一個項目	**tail ++**
dequeue	移除一個項目	**head++**
isEmpty	判斷佇列內是否還有資料	**head == tail ?**
isFull	判斷佇列內的資料是否滿了	**head >= queueSize-1?**


用鏈結串列
實作佇列

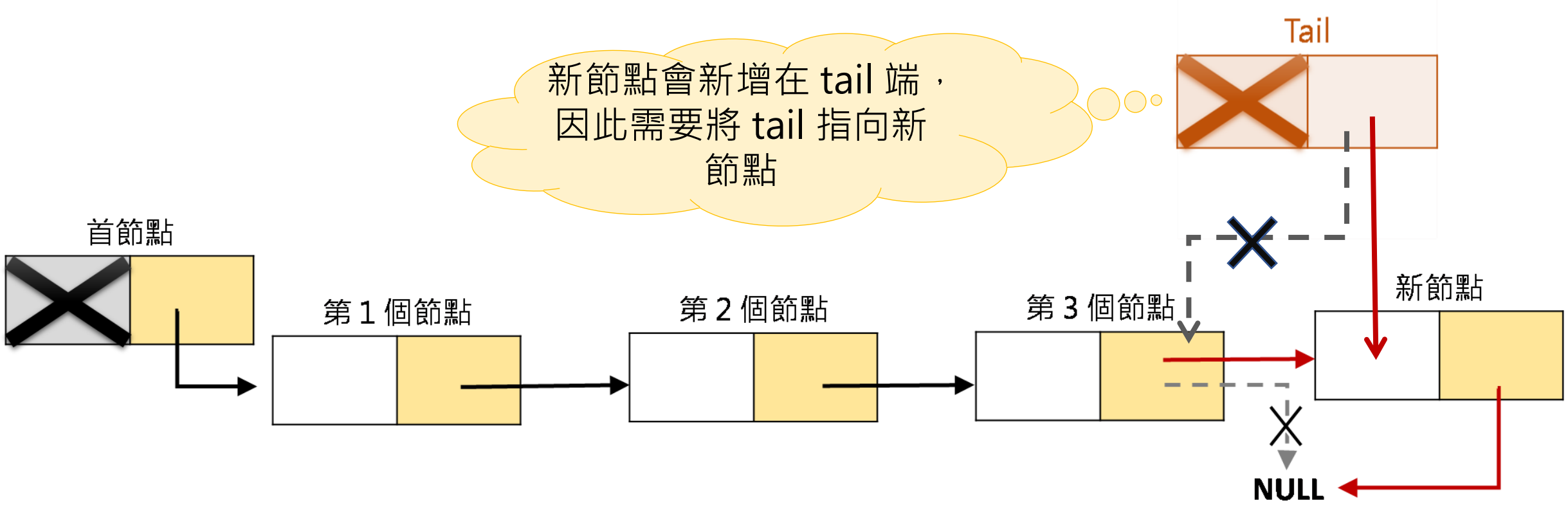
用鏈結串列實作佇列



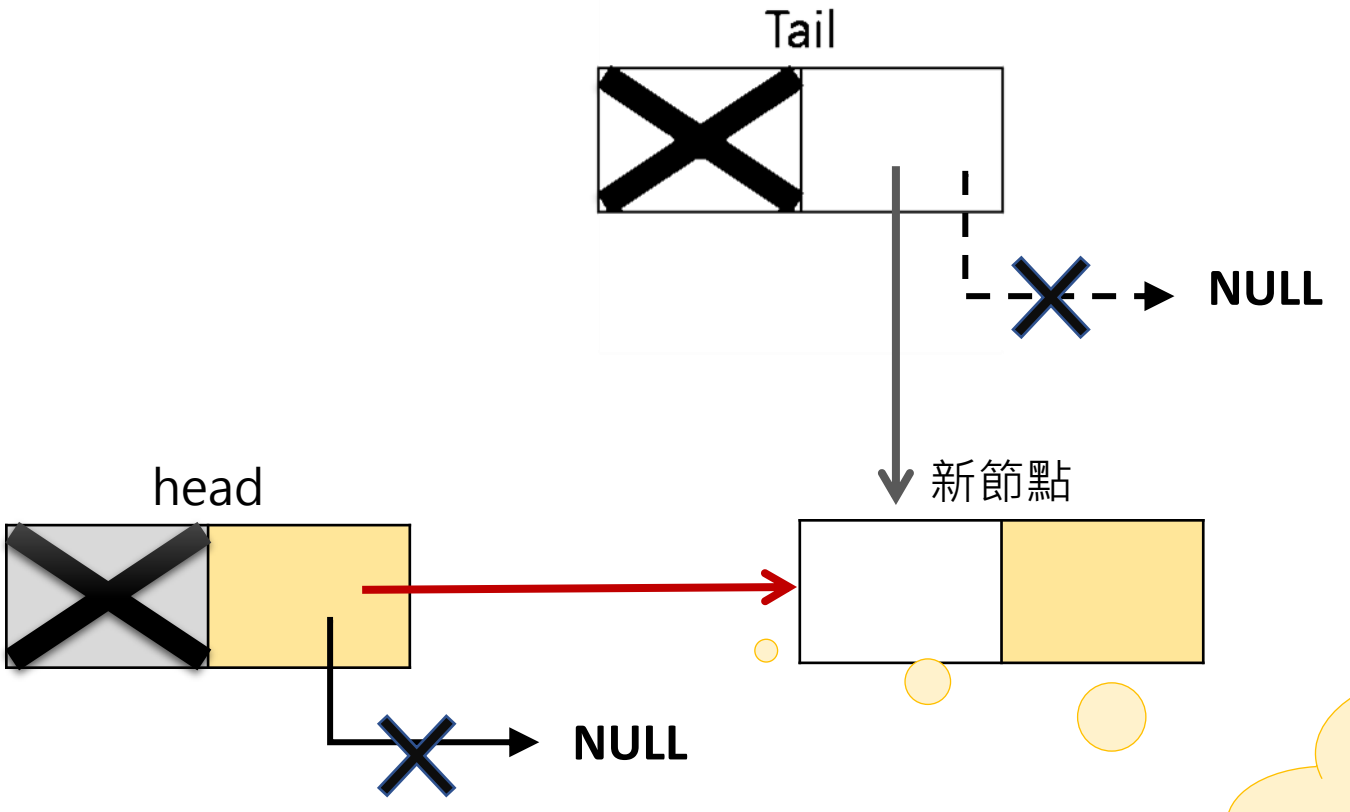


使用鏈結串列實作佇列時，除了使用鏈結串列原有的首結點(head)控制出口位置外，還會加上 tail 結點控制入口位置

用鏈結串列實作佇列 - 新增一個項目(1)



用鏈結串列實作佇列 - 新增一個項目(2)



若是空佇列，除了更動 tail 節點外，也要調整 head 節點

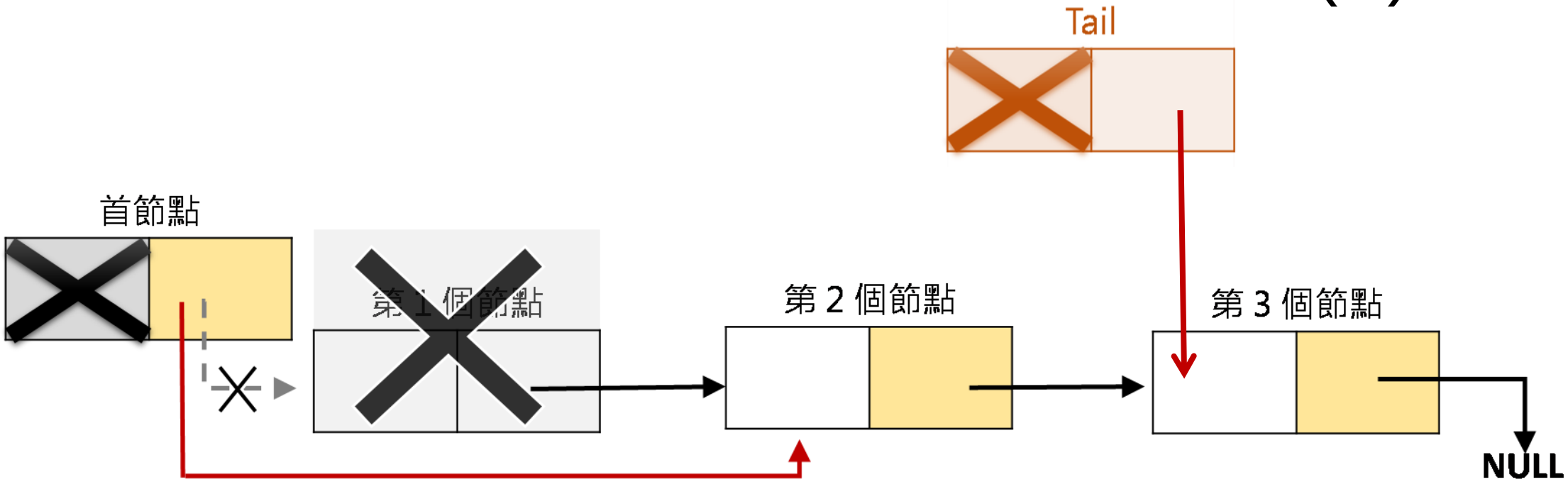
用鏈結串列實作佇列 - 新增一個項目(3)

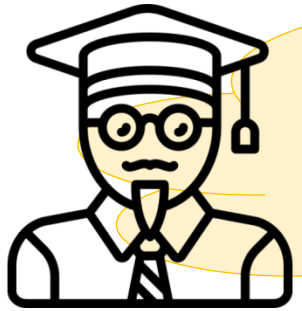
```
void enqueue(queueMaster *queue, int value)
{
  queueNode *node = queue->tail;
  node = (queueNode *) malloc(sizeof(queueNode));
  node->next = NULL;
  node->prev = NULL;
  node->data = value;
  if (queue->head == NULL) {
    queue->head = node;
    queue->tail = node;
  } else {
    queue->tail->next = node;
    node->prev = queue->tail;
    queue->tail = node;
  }
}
```

若是空佇列，head 與 tail 節點都指向新節點

若非空佇列，就調整 tail 節點與前一個節點的 next 值

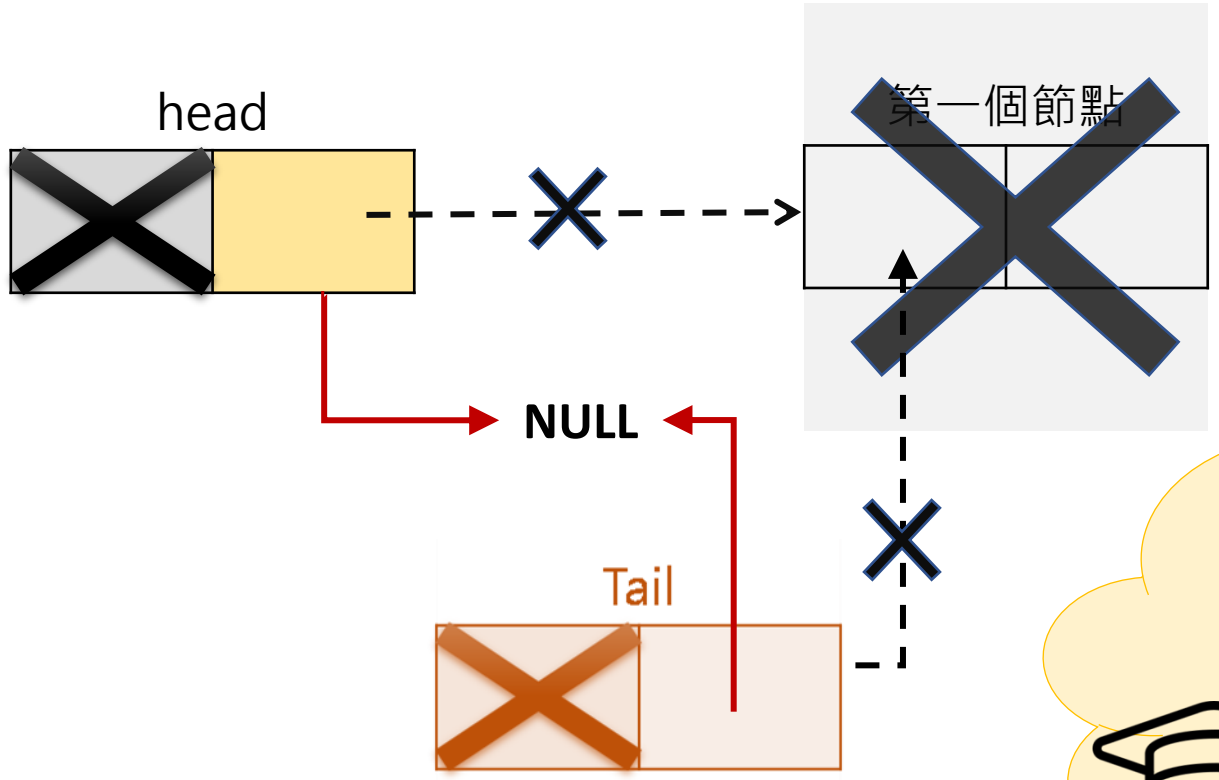
用鏈結串列實作佇列 - 移除一個項目(1)



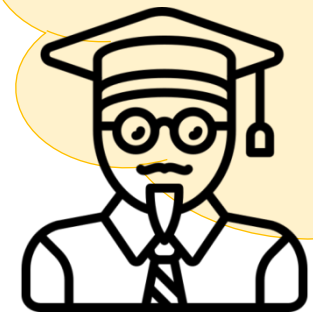


若非空佇列時，移除一個項目就使用鏈結串列**刪除前端節點**的方式移除

用鏈結串列實作佇列 - 移除一個項目(2)



若是佇列內只剩一個節點時，執行移除動作時，tail節點也要更動



用鏈結串列實作佇列 – 移除一個項目(3)

```
int dequeue(queueMaster *queue)
{
  queueNode *node = queue->head;
  if (isEmpty(queue)){
    return -1;
  }
  printf("[dequeue] value: %d\n", node->data);
  if (queue->head == queue->tail) {
    queue->head = NULL;
    queue->tail = NULL;
  } else {
    queue->head = node->next;
    queue->head->prev = NULL;
  }
  free(node);
  return 1;
}
```

若只剩一個節點，
head 與 tail 節點都指
向 NULL

若佇列有多個節點，就
調整 head 內的指標

用鏈結串列實作佇列 - 是否還有資料

```c
int isEmpty(queueMaster *queue)
{
  return (queue->head == NULL) ? 1 : 0;
}
```

當 head 或 tail 指向 NULL時，就代表佇列已經空了。因此也可以使用 tail == NULL 判斷。

用鏈結串列實作佇列 - 移除佇列

```
void freeQueue(queueMaster *queue)
{
  queueNode *currentNode = queue->head;
  queueNode *nextNode = NULL;
  int count = 0;
  while(currentNode) {
    nextNode = currentNode->next;
    free(currentNode);
    count++;
    currentNode = nextNode;
  }
  free(queue);
  printf("free %d nodes\n", count);
}
```

先逐一將使用 malloc() 配置的節點空間透過 free() 釋出記憶體。

最後也 需要將佇列結構變數釋出記憶體空間。

用鏈結串列實作佇列 - 取得項目數量

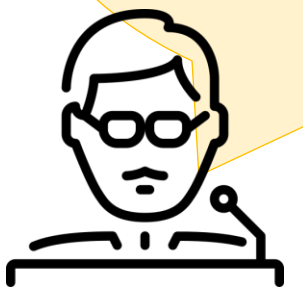
```
int getSize(queueMaster *queue)
{
  queueNode *node = queue->head;
  int count = 0;
  while(node) {
    node = node->next;
    count++;
  }
  return count;
}
```

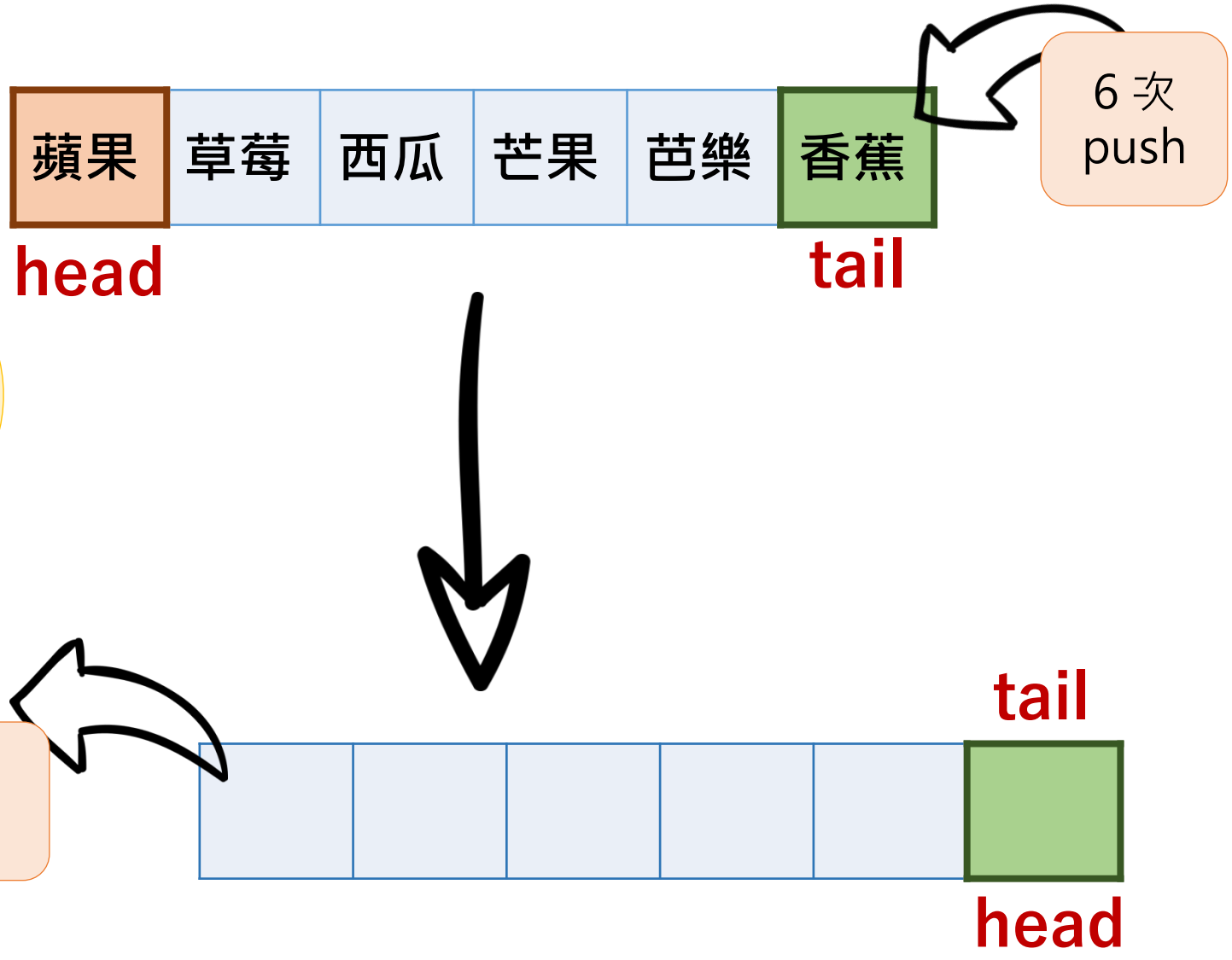
直到 next 指向 NULL 前，逐一檢視所有節點，便可取得佇列內的項目數量

延伸的概念

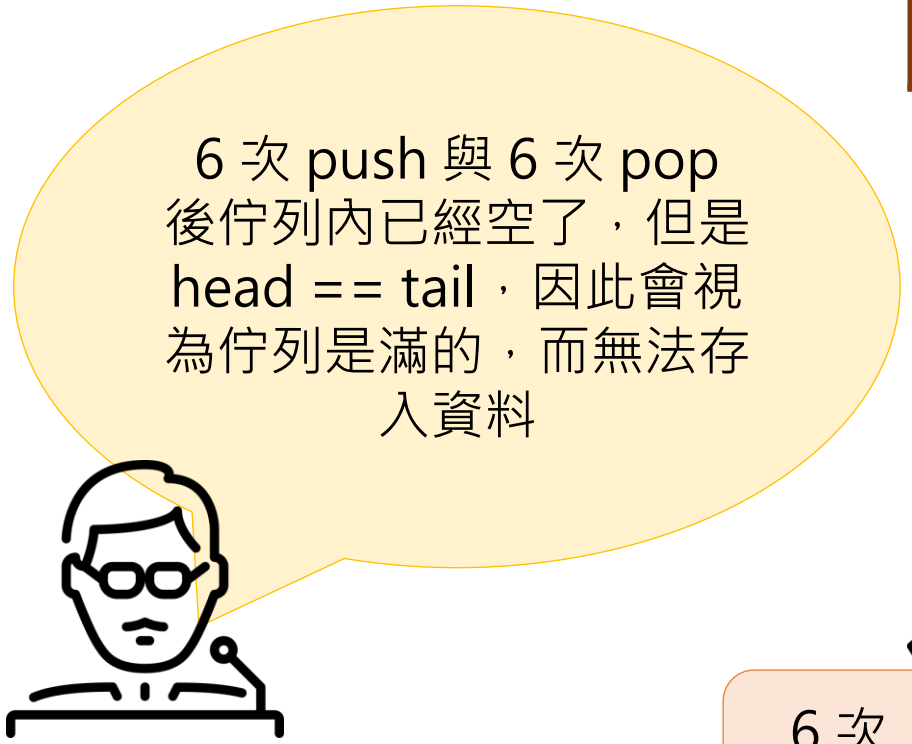
環狀佇列 (1)

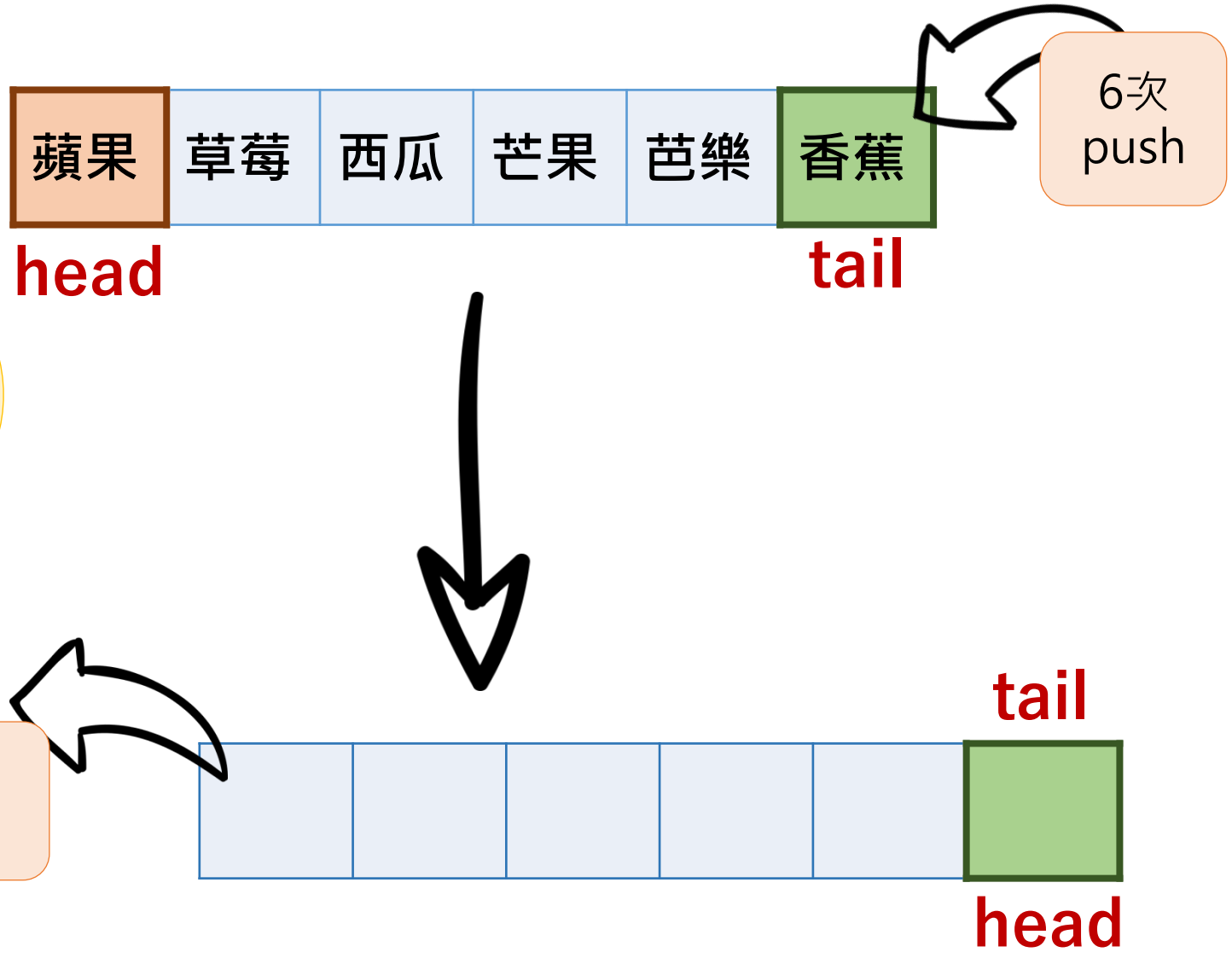
假若使用陣列實作佇列有六個空間的佇列，執行六次的 push，再執行六次的 pop，那這個佇列會如何呢？





環狀佇列 (2)

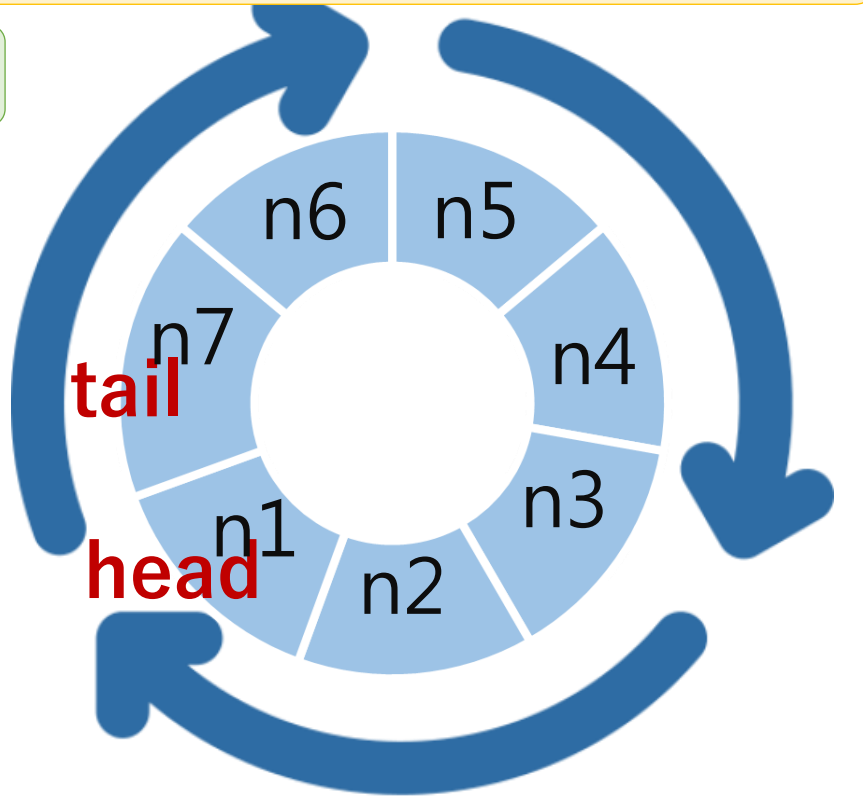

6 次 push 與 6 次 pop 後佇列內已經空了，但是 head == tail，因此會視為佇列是滿的，而無法存入資料


蘋果
草莓
西瓜
芒果
芭樂
香蕉
head
tail
6次 push
6 次 pop
tail
head

環狀佇列 (3)

線性

環狀



tail = (tail + 1) % queueSize

head = (head + 1) % queueSize

從線性的轉成環狀的，只要透過 % 運算子，只要同一時間儲存在佇列中的資料不要超過佇列的空間大小，就可以將空間重複使用