

週三作業討論



HW

雙向(環狀)鏈結串列

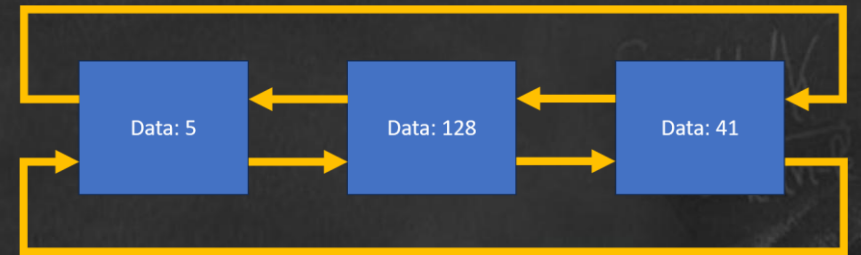
Q:完成以下程式，使輸出結果如右



```
struct DLLNode {  
    int data;  
    DLLNode* prev;  
    DLLNode* next;  
};  
  
struct DLL {  
    DLLNode* head;  
}
```

```
Int main(){  
    //宣告3個DLL  
    //其中2個分別用pushback和pushfront  
    放入11、22、33、44、55  
    //印出3個DLL  
}
```

```
11 22 33 44 55  
55 44 33 22 11  
empty
```



殘酷對決：Array vs. Linked List

沒有完美的資料結構，只有最適合的情境。

比較項目	Array (陣列)	Linked List (鏈結串列)
記憶體配置	連續空間，大小固定	分散存放，動態增長
讀取 (找第 N 個)	O(1) 極快 (用算的)	O(N) 慢 (必須從頭找起)
插入 / 刪除	O(N) 慢 (牽一髮動全身)	O(1) 極快 (只改指標)
適用情境	頻繁查詢 (如：資料庫索引)	頻繁新增修改 (如：播放清單)

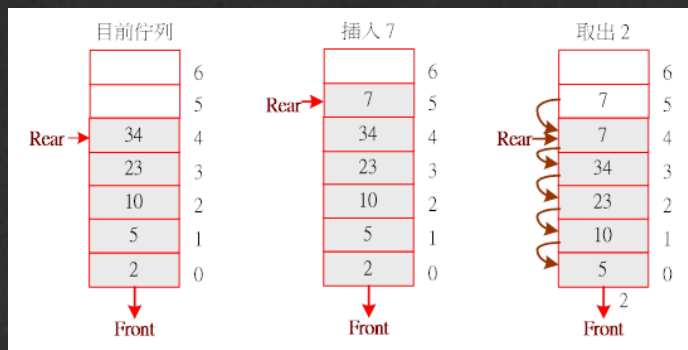
Stacks, Queues & Hash Tables



Queues and Stacks(佇列與堆疊)



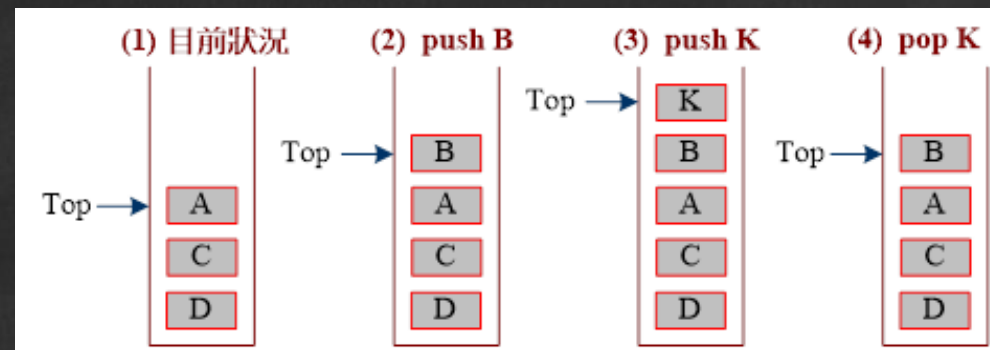
Queues(EX:排隊)



先進先出



Stacks(EX:堆書)



後進先出，只有頂端需要變數來做紀錄

皆只有兩種操作方式：push(放入)&pop(取出)

Queue (佇列)：講求公平的排隊

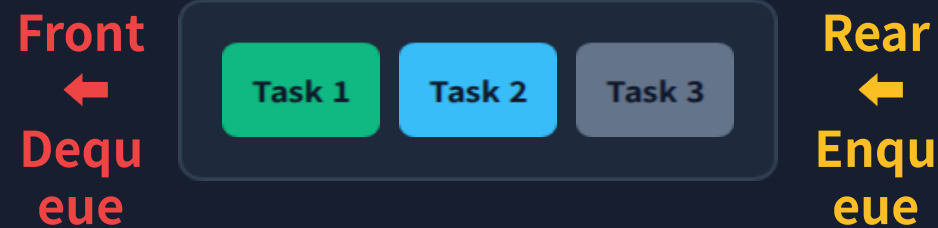
Queue 就像買手搖飲時的「**排隊人潮**」。

- **核心精神 FIFO**：First-In, First-Out (先進先出)。
先來排隊的人優先處理。

- **兩大專屬動作**：

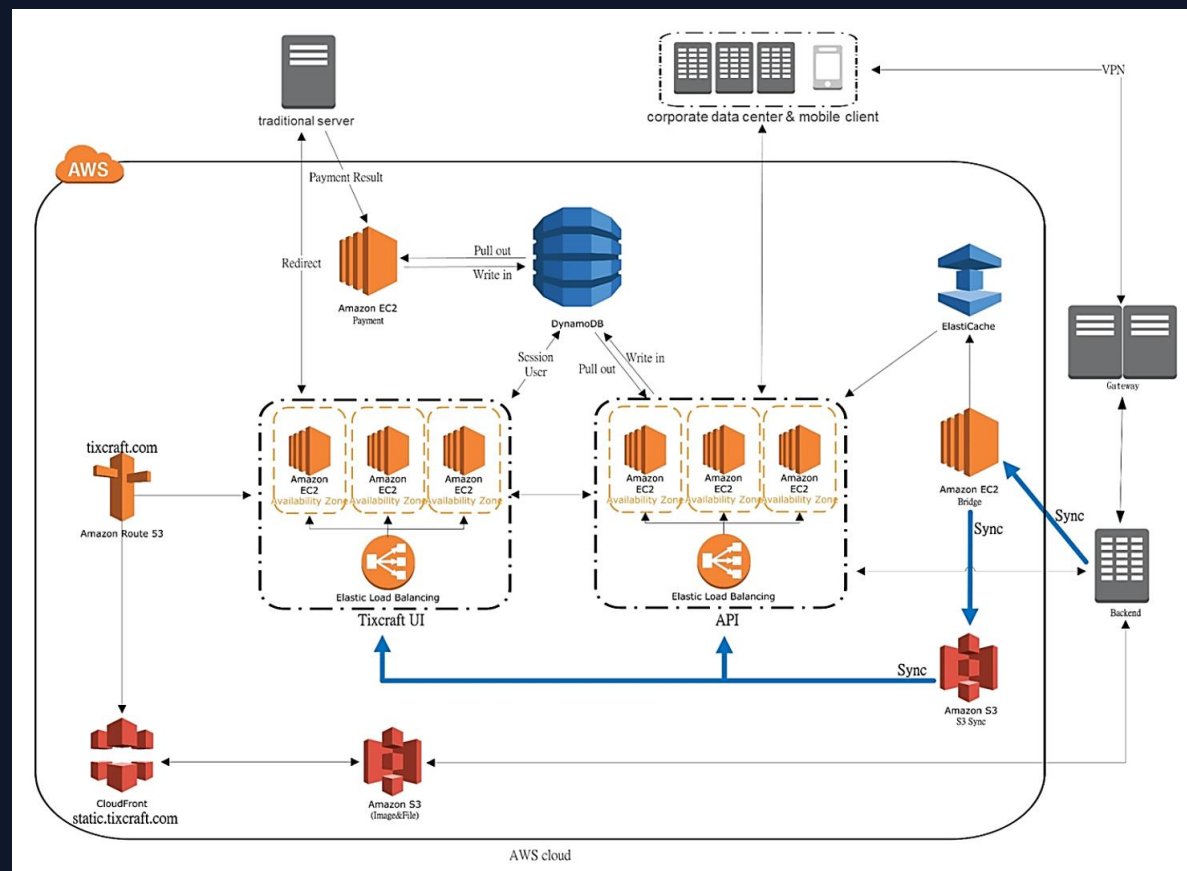
- **Enqueue (入隊)**：新資料走到隊伍的「尾巴 (Rear)」排隊。

- **Dequeue (出隊)**：最前面的資料從「頭部 (Front)」離開。



Queue 的真實應用

生活中的印表機、或是買演唱會門票防塞車的系統，背後全都是 Queue。



環狀佇列 (Circular Queue) 解決空間浪費

如果用 Array 實作 Queue 會遇到致命問題：

- Front 一直前進離開，Rear 一直往後加。
- Array 前面雖然空出來了，但 Rear 已經頂到最後面，造成**假性溢位 (False Overflow)**！
- 解法：環狀佇列。讓 Rear 走到陣列盡頭時，利用**取餘數 (%)** 技巧，繞回陣列最前面的空位。

Index 更新的數學公式

:

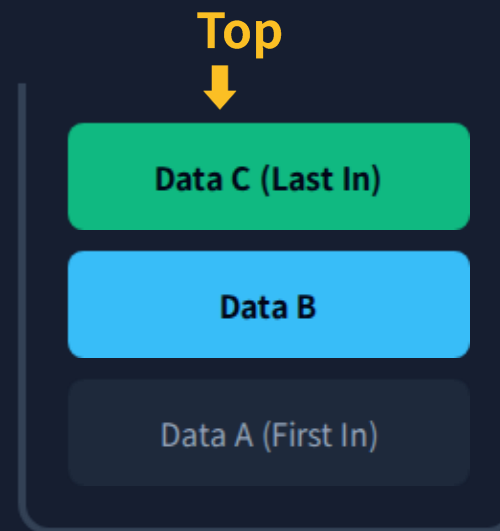
```
Rear = (Rear + 1) % MAX_SIZE;
```

```
Front = (Front + 1) % MAX_SIZE;
```

Stack (堆疊)：後進先出的藝術

Stack 就像去吃壽司時「**疊盤子**」，或是一個狹窄的桶子。

- **核心精神 LIFO**：Last-In, First-Out (後進先出)。最後放進去的，一定是第一個被拿出來。
- **兩大專屬動作**：
 - **Push (推入)**：把資料壓入堆疊的最頂端。
 - **Pop (彈出)**：把堆疊最頂端的資料拿出來。
- **封閉性**：嚴格禁止從中間抽出資料。



Stack 的真實應用

為什麼 Word 編輯器可以按 Ctrl+Z (復原)？為什麼程式碼可以知道下一步去哪？

應用 1：Undo 機制 (復原)

每一次打字或刪除，軟體就把動作 **Push** 進 Stack。
當你按復原時，就 **Pop** 出最新的一個動作並反向執行。

應用 2：Call Stack (呼叫堆疊)

當 A 函數呼叫 B 函數，系統會把 A 的狀態暫存並 Push 起來。等 B 執行完 Pop 掉後，系統就知道要回到 A 繼續執行。(無窮遞迴就會引發 Stack Overflow !)



Queues and Stacks (佇列與堆疊)

Q:完成以下程式，使輸出結果如右

```
struct StackArray {
```

```
StackArray(int cap) {  
    capacity = cap;  
    top = -1;  
    stack = new int[capacity];  
}  
~StackArray() {  
    delete[] stack;  
}
```

```
    int top;           //最上層的元素  
    int capacity;      //最大容量  
    int* stack;  
    void DoubleCapacity()  
    bool IsFull()  
    bool IsEmpty()  
    void Push(int value)  
    void Pop()  
    int GetSize()  
    void Print()  
}
```

```
int main(){
```

```
    //宣告容量為3的StackArray  
    //用函式檢查是否為空  
    //若結果為空則用迴圈放入11、22、33、44、  
    55、66  
    //當容量全滿時，先印出後再將容量變為2倍  
    //繼續放入剩下的元素，再印出  
    //將55和66取出(後進先出)，然後用函式求得陣列  
    中的元素數量  
}
```

```
11 22 33  
11 22 33 44 55 66  
11 22 33 44  
4
```



完成以下程式，使輸出結果如右



```
int main(){  
    //宣告容量為3的QueueArray  
    //用函式檢查是否為空  
    //若結果為空則用迴圈放入11、22、33、44、55、66  
    //直到容量全滿後，先印出再將容量變為2倍  
    //繼續放入剩下的元素，再印出  
    //將11和22取出，然後用函式求得陣列中有元素數量和top的元素  
}
```

```
11 22 33  
11 22 33 44 55 66  
33 44 55 66  
4
```



```
int capacity;    // 陣列的總容量
int front;       // 隊頭指標 (指向第一個元素)
int back;        // 隊尾指標 (指向最後一個元素的下一個位置)
int* queue;      // 動態陣列指標 (用來儲存資料)
```

完成以下程式，使輸出結果如右



```
struct QueueArray {
    int capacity, front, back;
    int* queue;
}
```

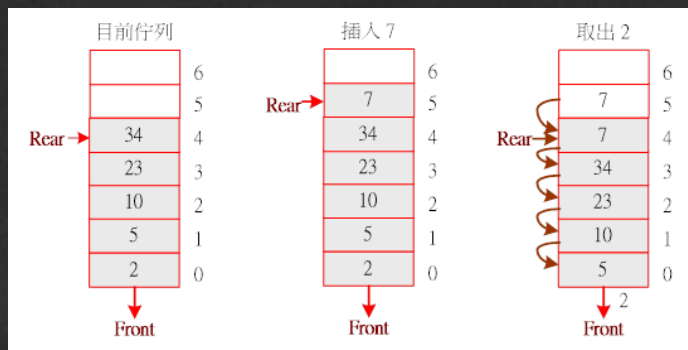
```
void DoubleCapacity()
bool IsFull()
bool IsEmpty()
void Push(int value)
void Pop()
int GetSize()
void Print()
```



Queues and Stacks List(佇列與堆疊串列)



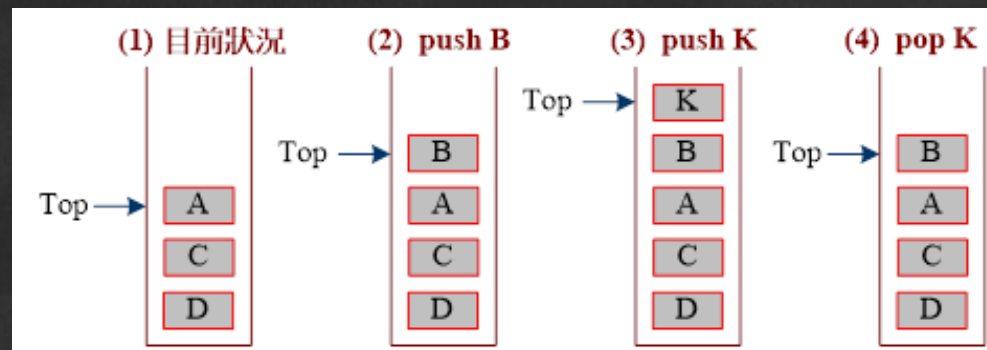
Queues(EX:排隊)



先進先出，需要兩個指標(頭&尾)



Stacks(EX:堆書)



後進先出，只有頂端需要指標

HW



Queues and Stacks List(佇列與堆疊串列)

Q:使用單向串列完成以下程式，使輸出結果如右



```
int main() {  
    StackList s;  
    s.Push(99);s.Push(88);s.Push(77); //PushBack? PushFront?  
    cout << s.Top() << endl;  
    s.Pop(); s.Pop();  
    cout << s.GetSize() << endl;  
    cout << s.Top() << endl;  
}
```

77
1
99

HW



Queues and Stacks List(佇列與堆疊串列)

Q:使用單向串列完成以下程式，使輸出結果如右



```
int main() {  
    QueueList q;  
    q.Push(99); q.Push(88); q.Push(77); //PushBack? PushFront?  
    cout << q.front->data << " " << q.back->data << "\n";  
    q.Pop();q.Pop();  
    cout << q.GetSize() << endl;  
    cout << q.GetFront() << " " << q.GetBack() << endl;  
}
```

```
99 77  
1  
77 77
```

該用 Array 還是 Linked List 實作？

Stack 和 Queue 只是「概念」，底層都可以用之前教的結構來寫：

用 Array 實作

優點：實作簡單，記憶體配置快速。

缺點：大小固定（除非用動態陣列），Queue 會有假性溢位問題（需改寫為 Circular Queue）。

用 Linked List 實作

優點：大小無限制，動態增長，不需要搬移資料。

缺點：每個節點都需要額外記憶體存指標，容易產生記憶體碎片。

結論：如果資料量未知且變動大，選 Linked List。如果大小固定且講求極速，選 Array！

Hash Table



Hash Table：如何把字串變成數字？

Array 找東西很快 $O(1)$ ，但只能用數字 Index。如果我想用字串 "Apple" 當作 Index 該怎麼辦？

1. 雜湊函數 (Hash Function)

這是一台轉換機。把 "Apple" 丟進去，它可能把字母轉成 ASCII 碼並加總，假設得到 500。

2. 取餘數 (Modulo %)

假設 Array 大小只有 10。我們將 $500 \% 10 = 0$ 。所以 "Apple" 的資料，就存在 Index 0 的位置！

"Apple"

↓
Hash Function()

Index 0



Hash table

Character	Unicode point
-----------	---------------

B	66
---	----

o	111
---	-----

b	98
---	----

Sum	275
-----	-----

addName	Sum of Unicode
---------	----------------

John	399
------	-----

Alice	478
-------	-----

Pete	398
------	-----

Hash code = $275 \% 10 = 5$

Put the "Bob" at index 5

0	1	2	3	4	5	6	7	8	9
					Bob			Alice	John

Collision(碰撞)

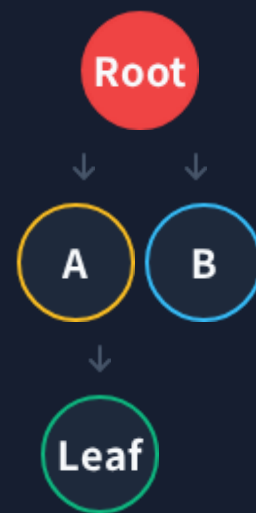
- Chaining: store the vector or link list in the vector
 - Pros: Simple to implement. The table will not "fill".
 - Cons: The linked list will become very long → Query speed will decrease
- Open Addressing: If the slot is occupied, **find another empty slot**
 - **Linear Probing** : index = (index, index+1, index+2, ...) % table_size
 - **Quadratic Probing** : index = (index ± 1², 2², 3², ...) % table_size
 - **Double Hashing**: index = (hash1(key) + i * hash2(key)) % table_size

i指次數，一開始是0，當碰撞一次i=1，當碰撞兩次i=2.....

Trees (樹狀結構)

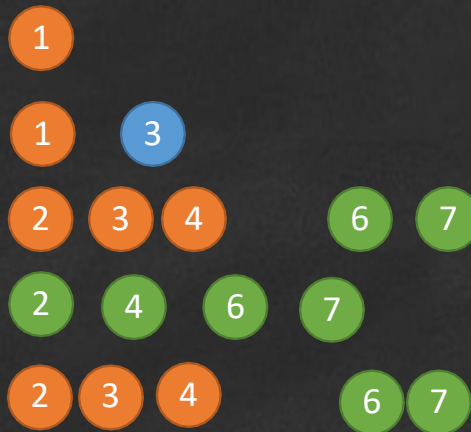
基礎術語：電腦裡的樹是倒著長的

- **Root (根節點)**：整棵樹最頂端的起點，整棵樹只有唯一一個。
- **Parent & Child (父與子)**：節點往下連接著小孩。上層的叫爸爸，下層的叫小孩。
- **Leaf (葉節點)**：位於最底層，**沒有任何小孩**的孤單節點。
- **Depth / Height (深度/高度)**：這棵樹總共有幾層。

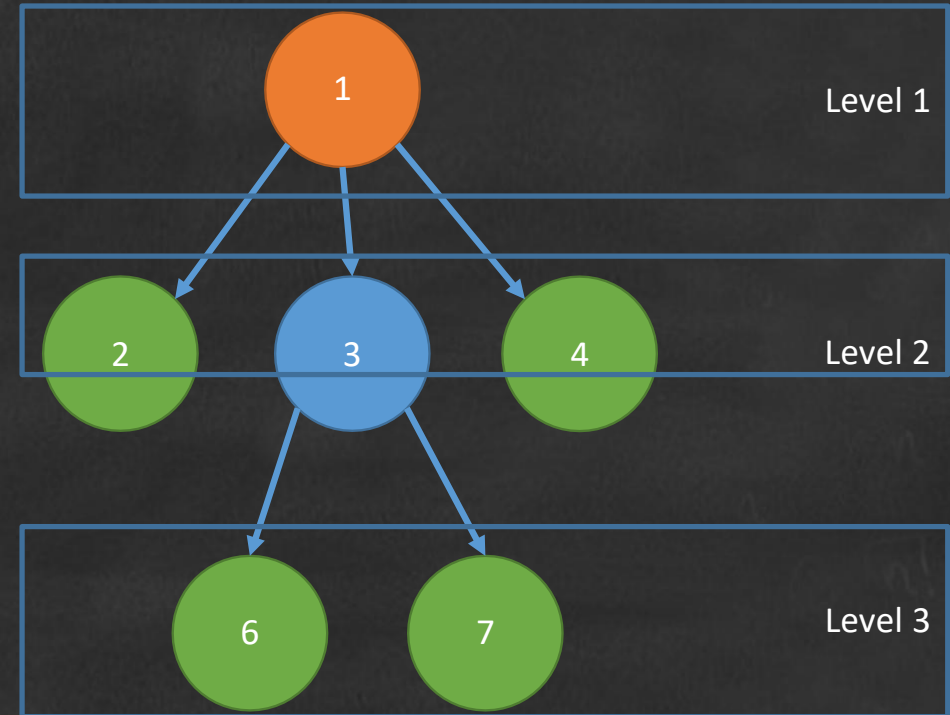


Tree

- Root
- Parent
- Child
- Leaf
- Sibling



- Degree, Level, Height



<https://medium.com/cyen6/%E8%B3%87%E6%96%99%E7%B5%90%E6%A7%8B%E7%AD%86%E8%A8%98-tree-2603a518a021>