

資料結構



演算法

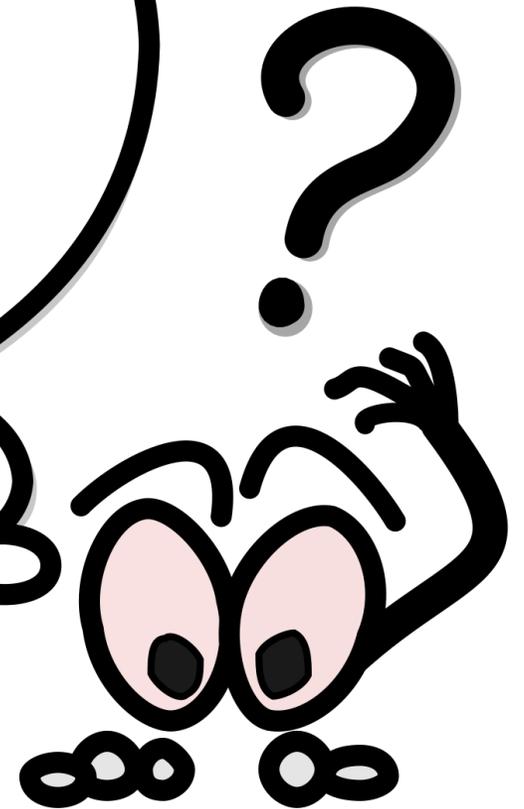
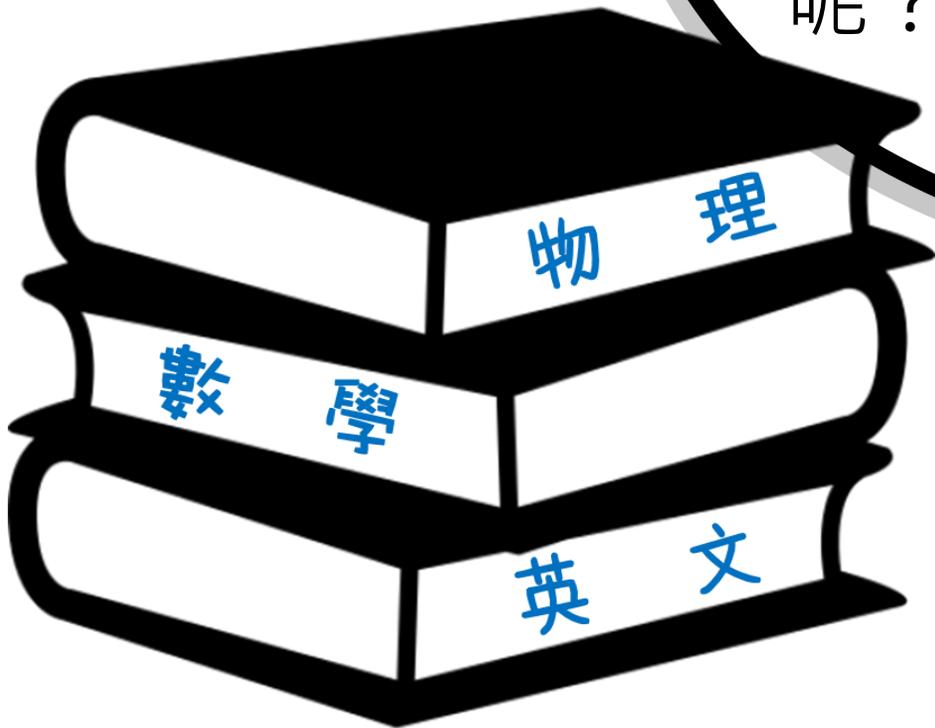
堆疊

Stack

堆疊 Stack

[十八豆教育科技](#)

桌上有依序疊上去的英文、  
數學、物理三本書，如果  
想要拿英文出來看怎麼拿  
呢？





先把最上面的物  
裡拿走，再把數學  
移走，就可以拿到  
英文了！

這就是『堆疊 stack』

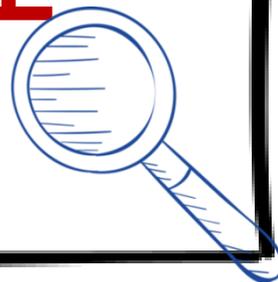


單一出入口的容器中，  
後放入的要先拿出



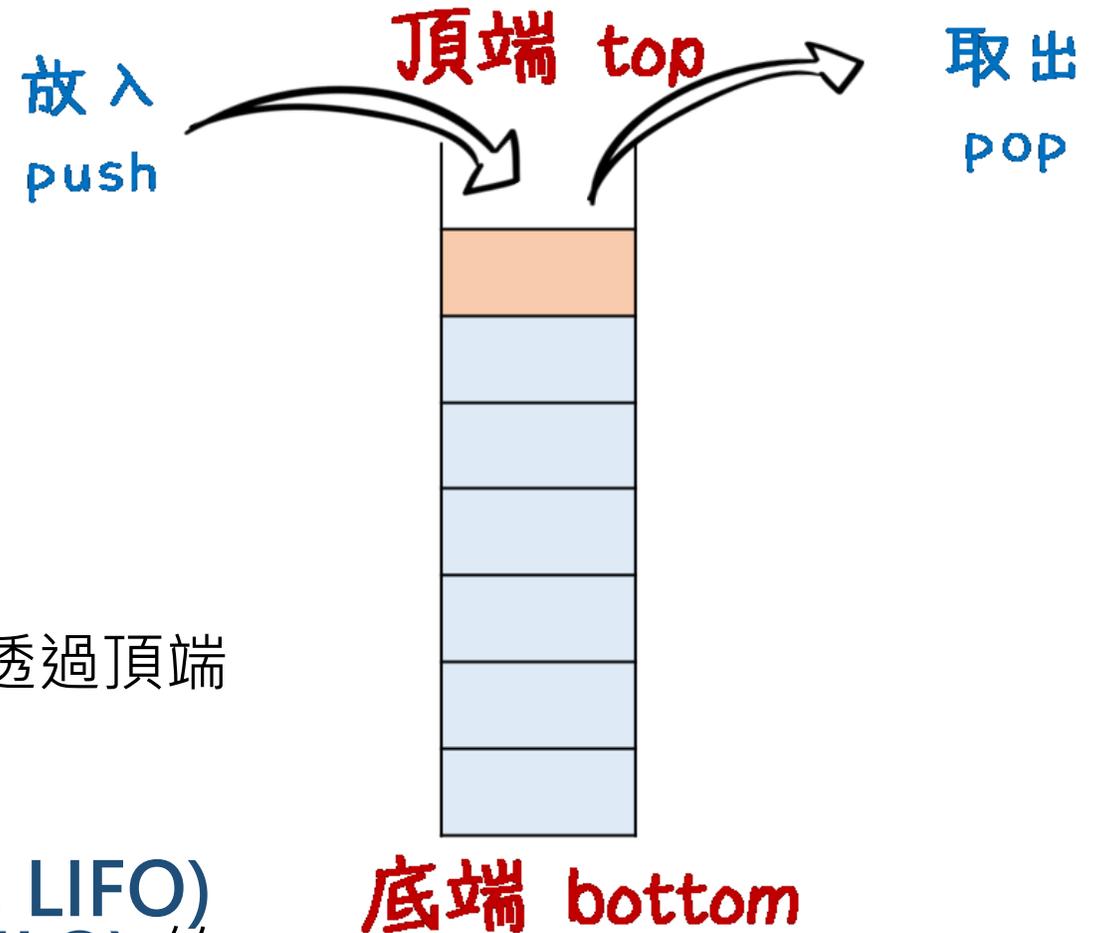
就是『堆疊』

# 堆疊特性



# 堆疊結構

- 是一種串列
- 只有一個出入口的串列
- 出入口那端稱為**頂端(top)**
  - 另一端稱為**底端(bottom)**
  - 資料的新增(放入)與移除(取出)都是透過頂端(top)操作
- 具有**後進先出(Last-In-First-Out, LIFO)**與**先進後出(First-In-Last-Out, FILO)**的特性



# LIFO 與 FILO (1)

放入  
push

頂端 top

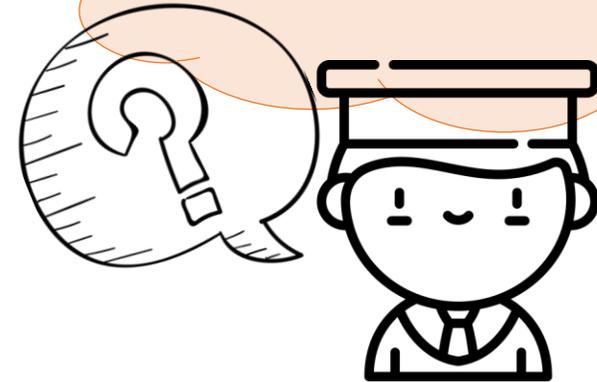
取出  
pop

依序將蘋果、草莓、  
西瓜、芒果、芭樂、  
香蕉、葡萄7種水果  
放入堆疊內

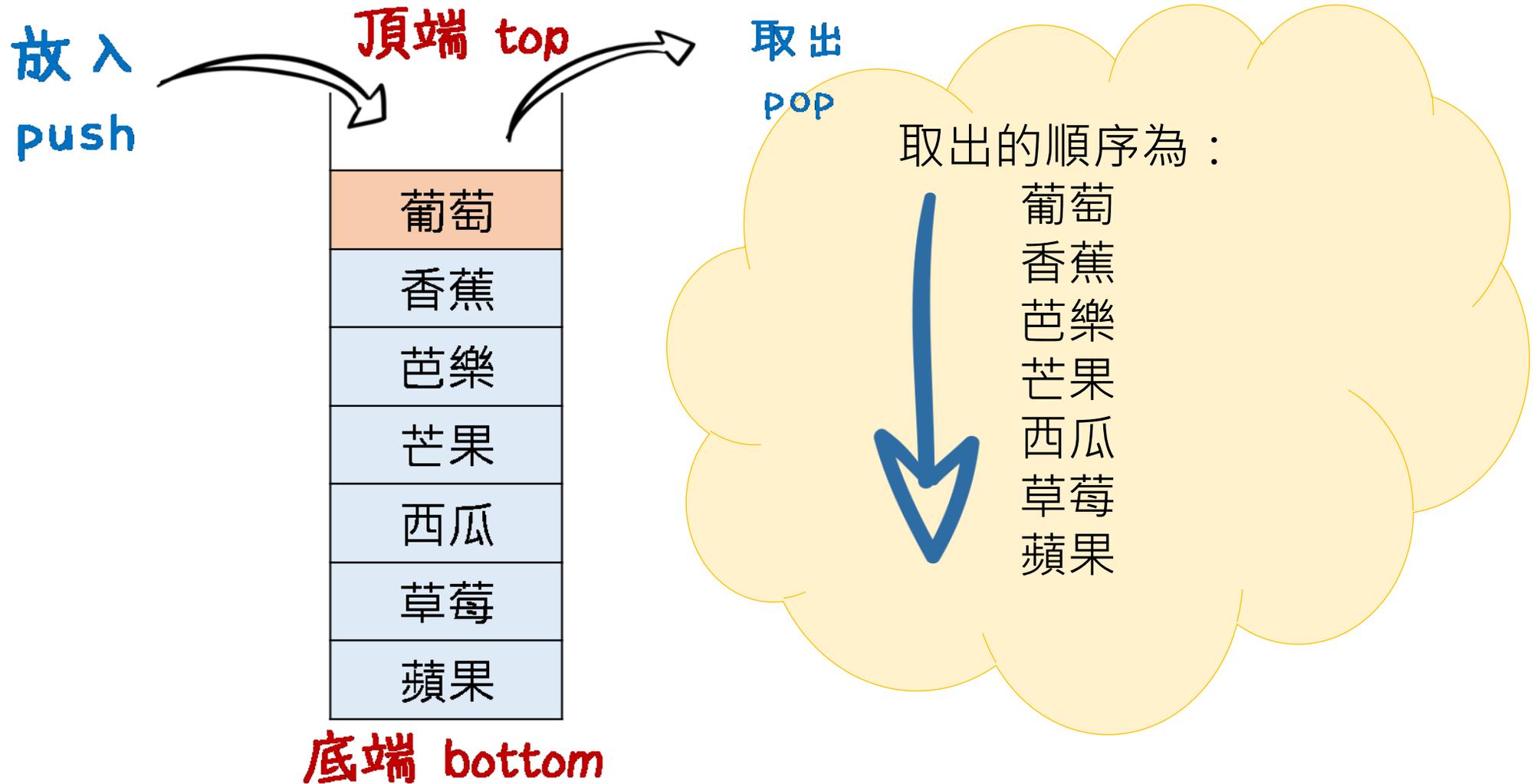


底端 bottom

知道取出的順序嗎？



# LIFO 與 FILO (2)

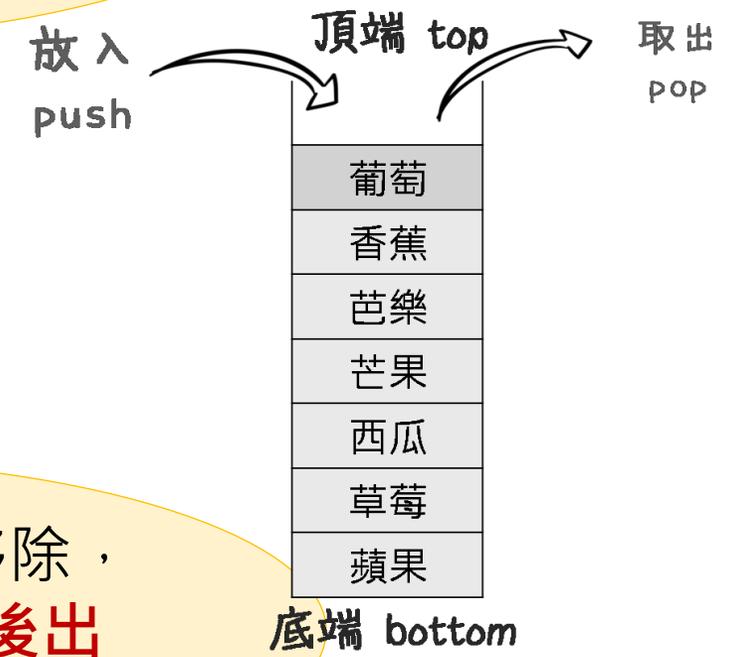


# LIFO 與 FILO (3)

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

最後放入的會最先移除，  
此種特性稱為**後進先出**  
(Last-In-First-Out, LIFO)

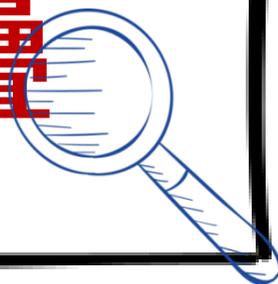
最先放入的會最後移除，  
此種特性稱為**先進後出**  
(First-In-Last-Out, FILO)



# 堆疊的實作

- 兩種資料結構可以實作堆疊
  - 陣列
    - 優點：資料操作存取快速
    - 缺點：堆疊空間是固定的
  - 鏈結串列
    - 優點：堆疊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 實作堆疊



# 用陣列實作堆疊 – 兩大元素

- 基本元素

- ① 堆疊陣列

- ② top 變數

- 記錄目前堆疊內的項目數
    - 預設初始值為 **-1**

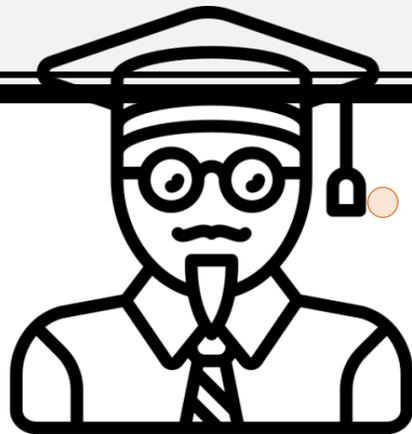
- 輔助元素

- ③ 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])`

# 用陣列實作堆疊 – 兩大元素

```
typedef struct stackStruct {  
    int *stackList;  
    int top;  
    int size;  
} stackType;
```



由於堆疊有兩個以上基本元素，因此建議使用 **struct (結構)** 儲存堆疊資料

# 用陣列實作堆疊 – 五大操作

**createStack**      建立新堆疊

**push**            新增一個項目

**pop**             移除一個項目

**isEmpty**        判斷堆疊內是否還有資料

**isFull**          判斷堆疊內的資料是否滿了

# 用陣列實作堆疊 – 建立新堆疊

```
stackType* createStack(stackType *stack, int size)
{
    stack = (stackType*)malloc(sizeof(stackType));
    newStack->stackList = (int *)malloc(sizeof(int) * size);
    newStack->top = -1;
    newStack->size = size;
    return newStack;
}
```



堆疊最重要的是用 top 變數掌握堆疊內資料存放情況，所以建立新堆疊時絕對要記得設定 top 變數



# 用陣列實作堆疊 – 新增項目

```
int push(stackType *stack, int value)
{
    if (isFull(stack)){
        return -1;
    }

    stack->top++;
    stack->stackList[stack->top] = value;

    printf("top:%d,value:%d", stack->top, stack->stackList[stack->top]);
    return stack->top;
}
```

1

新增項目時需要先檢查堆疊是否已經滿了

2

將 top 值加1，就是堆疊內要存放新資料的索引位置。



# 用陣列實作堆疊 – 移除一個項目

```
int pop(stackType *stack)
{
    if (isEmpty(stack)) {
        return -1;
    }

```

移除項目時需要先檢查  
堆疊內是否有資料

```
printf("top:%d,value:%d", stack->top, stack->stackList[stack->top]);
```

```
stack->stackList[stack->top] = 0;
stack->top--;
```

```
return stack->top;
}
```

將 top 值減1，就代表  
將那個位置視為無  
資料



# 用陣列實作堆疊 - 堆疊內是否還有資料

```
int isEmpty(stackType *stack)
{
    return (stack->top == -1) ? 1 : 0;
}
```



若 top 為 -1 就代表堆疊是空的

# 用陣列實作堆疊 – 堆疊內是否滿了

```
int isFull(stackType *stack)
{
    return (stack->top >= stack->size-1) ? 1 : 0;
}
```



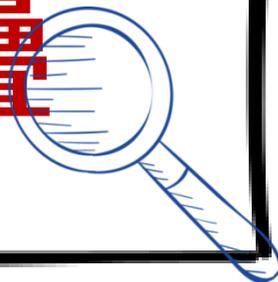
若 top 值大於等於堆疊  
預設空間數-1，就代表  
堆疊已經滿了

# 用陣列實作堆疊

- 兩大元素
  - 堆疊陣列
  - top 變數
- 五大操作

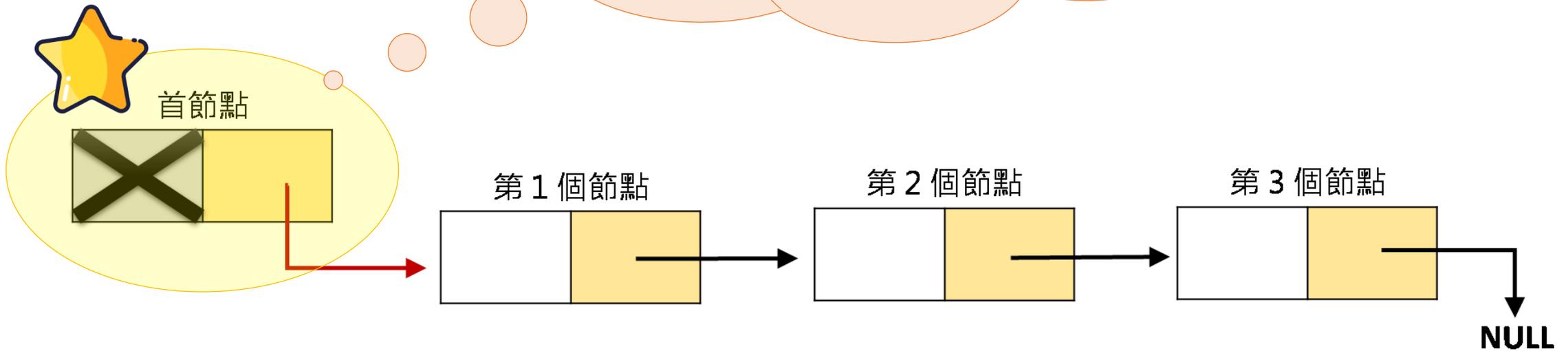
<b>createStack</b>	建立新堆疊	<b>top = -1</b>
<b>push</b>	新增項目	<b>top++</b>
<b>pop</b>	移除項目	<b>top--</b>
<b>isEmpty</b>	判斷堆疊內是否還有資料	<b>top == -1 ?</b>
<b>isFull</b>	判斷堆疊內的資料是否滿了	<b>top &gt;= stackSize - 1 ?</b>

# 用鏈結串列 實作堆疊



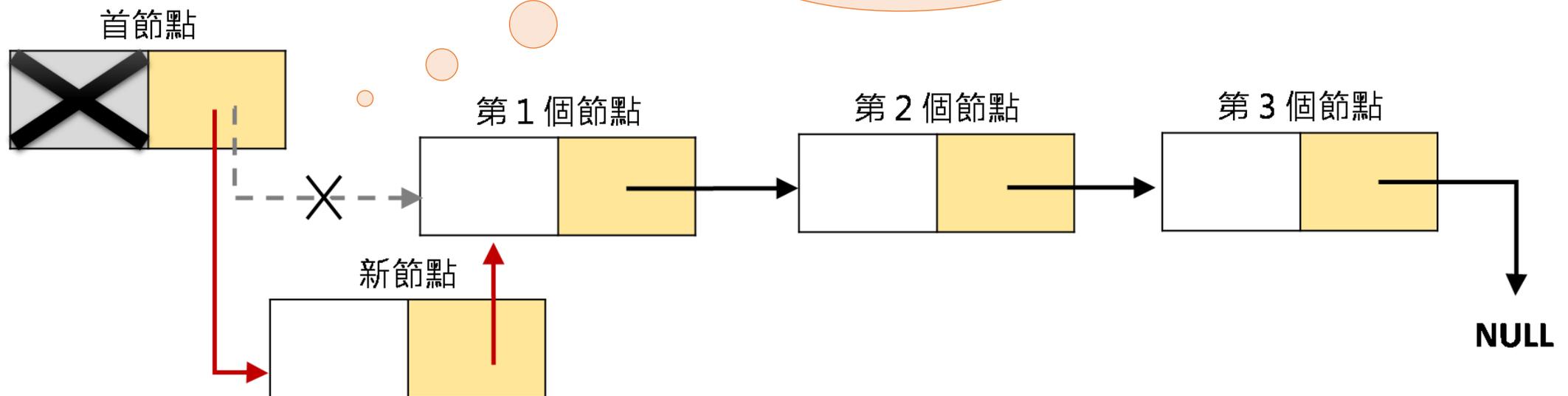
# 用鏈結串列實作堆疊：head -> top

使用鏈結串列實作堆疊時，只要將原本鏈結串列結構中的首節點（head）視為 top，不管新增或移除節點都由 head 端處理即可。



# 用鏈結串列實作堆疊 – 新增項目

新增項目時，就視為使用鏈結串列在前端新增節點的方式處理



# 用鏈結串列實作堆疊 – 新增項目

```
void push(stackNode *head, int value)
{
    stackNode *node;

    node = (stackNode *) malloc(sizeof(stackNode));
    node->next = head->next;
    node->data = value;
    head->next = node;
    printf("[push] value: %d\n", node->data);
}
```

先動態取得新結點空間，  
並設定新結點內的值與  
next 應指向的節點

1

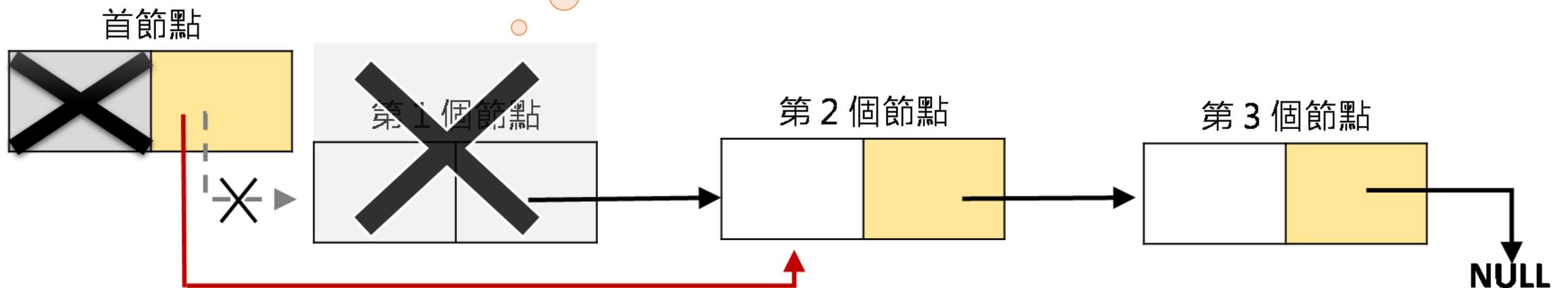
2

head (top) 節點改成  
指向新產生的 node



# 用鏈結串列實作堆疊 – 移除一個項目

移除項目時，就視為使用鏈結串列在刪除前端節點的方式處理



# 用鏈結串列實作堆疊 – 移除一個項目

```
int pop(stackNode *head)
{
    stackNode *node;
    int count = 0;

    if (isEmpty(head)){
        return -1;
    }

    node = head->next;
    head->next = node->next;

    printf("[pop] value: %d\n", node->data);
    free(node);
    return 1;
}
```

移除項目時需要先檢查  
堆疊內是否有資料

head (top) 結點改成  
指向原本的第二個節  
點



# 用鏈結串列實作堆疊 – 堆疊是否還有資料

```
int isEmpty(stackNode *head)
{
    return (head->next == NULL) ? 1 : 0;
}
```



若 head(top) 已經指向  
NULL 就代表堆疊已經空了

# 用鏈結串列實作堆疊 – 移除堆疊

```
void freeStack(stackNode *head)
{
    stackNode *node = head, *tmp;
    int count = 0;
    while (node != NULL) {
        tmp = node;
        node = node->next;
        free(tmp);
    }
}
```

使用 malloc 配置的節點資料，在資料不使用時，記得呼叫 free() 釋放記憶體



# 用鏈結串列實作堆疊 – 取得項目數量

```
int getSize(stackNode *head)
{
    stackNode *node = head->next;
    int count = 0;
    while (node != NULL) {
        node = node->next;
        count++;
    }
    return count;
}
```

直到 next 指向 NULL 前，  
逐一檢視所有結點，便可取  
得堆疊內的項目數量





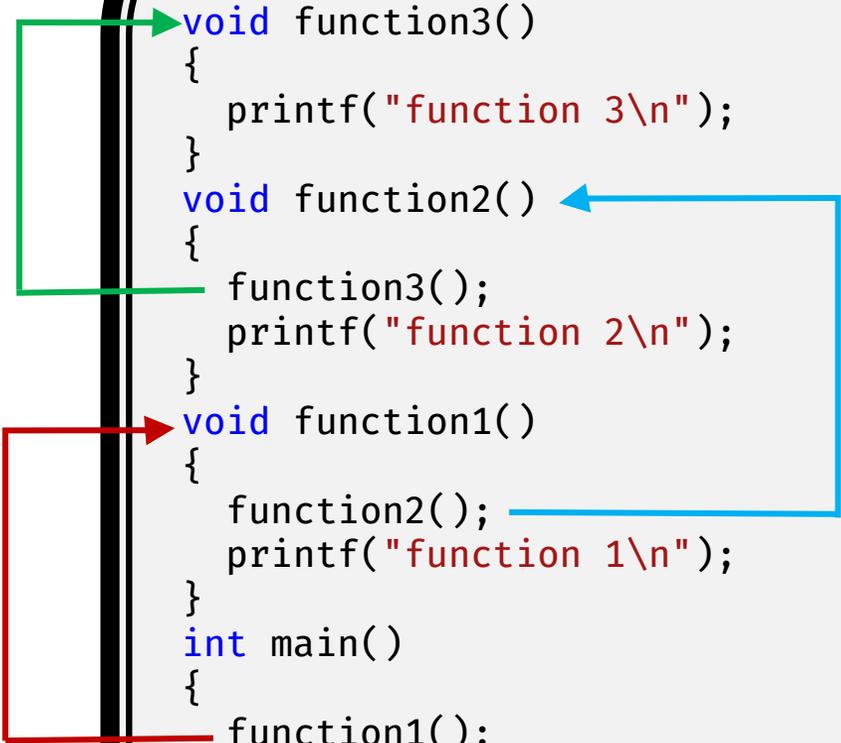
延伸的概念

# 堆疊的應用

- 日常生活：
  - 書本裝箱
  - 疊盤子
- 軟體程式：
  - 副程式的呼叫執行
  - 運算式的轉換及求值

# 堆疊的應用 - 副程式的呼叫執行

```
#include<stdio.h>
void function3()
{
    printf("function 3\n");
}
void function2()
{
    function3();
    printf("function 2\n");
}
void function1()
{
    function2();
    printf("function 1\n");
}
int main()
{
    function1();
    printf("main\n");
    return 0;
}
```



## 堆疊

```
printf("function 3\n");
printf("function 2\n");
printf("function 1\n");
printf("main\n");
```

# 堆疊的應用 - 副程式的呼叫執行

```
#include<stdio.h>
void function3()
{
    printf("function 3\n");
}
void function2()
{
    function3();
    printf("function 2\n");
}
void function1()
{
    function2();
    printf("function 1\n");
}
int main()
{
    function1();
    printf("main\n");
    return 0;
}
```



執行結果

```
dice - stack $ ./call.exe
function 3
function 2
function 1
main
```



# 堆疊的應用 - 運算式的轉換及求值

- 前序法(Prefix)
  - 運算子在運算元的前面
  - 例:  $+34.*84$ ,  $-AB$
- 中序法(Infix) 人類使用
  - 運算子在兩個運算元中間
  - 例:  $3+4$ ,  $8*4$ ,  $A-B$
- 後序法(Postfix) 電腦使用
  - 運算子在運算元的後面
  - 例:  $34+$ ,  $84*$ ,  $AB-$

電腦是使用後序法進行運算，因此就需要透過堆疊將一般人類使用的中序法轉為後序法再進行運算。

運算式的轉換可參考  
[中序式轉後序式 \(前序式\)](#)