



# 資料結構簡介

# Data Structure

鄭博宇 Benjamin Cheng

EMILY in IM



# Agenda

- 什麼是資料結構
- 機器需要做什麼？
- big O
- Pointer (link-based串列)
- Set & Dictionary
- 樹 Tree



# 課程進行

- 這是一堂單純的講座課（多發言，一起思考）
- Python Lecture 的複習
- 自學資源？網路上查關鍵字有很多筆記



# 講師介紹

Benjamin 鄭博宇

- 資管雙經濟 大二
- IG: @benjami8\_24n



# 這堂課希望…

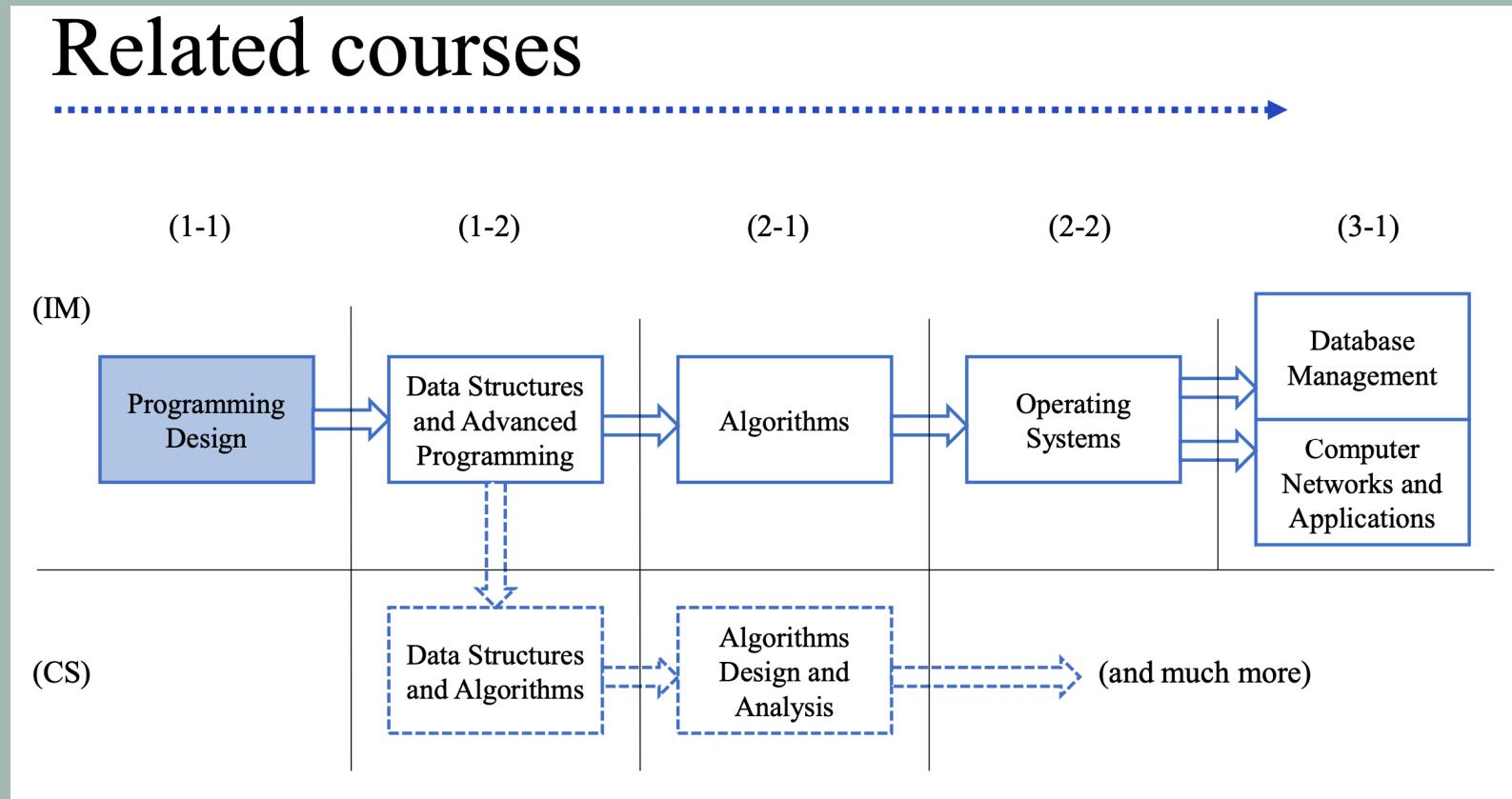
1. 讓大家了解學資訊的人，都在學什麼？
2. 資料結構為什麼很重要？
3. 有哪些常見的專有名詞？



# 台大資管系的必修

(Credit: 孔令傑教授 / PD\_2023 Finale)

## Related courses





# 演算法 vs. 資料結構

兩堂 資管系的必修：

- **演算法 (Algorithm)**：解決特定問題的有限步驟集合，透過明確的運算、邏輯推理或數學操作，將輸入轉換為期望的輸出，並強調效率與正確性。
- **資料結構 (Data Structure)**：用於組織、存儲和管理數據的方式，透過特定的結構設計（如陣列、鏈結串列、樹、圖等），提升數據操作的效率，並影響演算法的執行效能。



# 機器需要做什麼？

EMILY in IM



# 讀取與存取

EMILY in IM



# 讀取與存取

- 索引存取 (Indexing)：透過索引快速找到特定元素（如 List 中的 `list[i]`）。
- 順序存取 (Sequential Access)：依序讀取資料（如 讀取鏈結串列）。
- 區間查詢 (Range Query)：查找某個範圍內的資料（如 找出成績在 80~90 分的學生）。



# 搜尋與查找

EMILY in IM



# 搜尋與查找

- 線性搜尋 (Linear Search)：逐一檢查所有資料（如在無序資料集中找特定數字）。
- 二分搜尋 (Binary Search)：對已排序的資料進行高效查找（如搜尋電話簿）。



# 搜尋與查找

- 模式匹配 (Pattern Matching)：在字串或文本中查找特定模式  
(如 Ctrl + F 搜尋)。
- 最近鄰搜尋 (Nearest Neighbor Search)：在圖中找出最接近的  
資料點 (如 GPS 定位的最近加油站)。



# 插入與刪除

EMILY in IM



# 插入與刪除

- 尾部插入 (Append)：在結尾新增元素（如 `list.append(x)`）。
- 中間插入 (Insert at Position)：在特定位置插入資料（如 在 List 索引 3 插入值）。
- 頭部插入 (Prepend)：在開頭插入元素（如 `deque.push_front(x)`）。



# 插入與刪除

- 刪除特定元素 (Remove Element)：移除特定值（如刪除 Linked List 中的某個節點）。
- 批量刪除 (Bulk Deletion)：一次刪除多個符合條件的元素（如刪除所有過期郵件）。



# 排序與重組

EMILY in IM



# 排序與重組

- 整體排序 (Global Sorting)：對整個集合排序（如 quick sort, merge sort）。
- 部分排序 (Partial Sorting)：只排序部分元素（如 只找出前 10 名）。
- 洗牌 (Shuffling)：隨機打亂順序。
- 反轉 (Reversing)：將順序顛倒（如 `array.reverse()`）。



# 統計

EMILY in IM



# 統計

- 計數 (Counting)：計算某個條件符合的元素數量（如統計有多少人考 100 分）。
- 求和 (Summation)：計算所有元素的總和（如 `sum(arr)`）。
- 平均值 (Mean)：計算數據的平均值（如 `sum(arr) / len(arr)`）。



# 統計

- 最大值 / 最小值 (Max / Min) : 找出數據中的極值 (如 `max(arr)`)。
- 中位數 (Median) : 找出排序後的中間值 (如 `median(arr)`)。
- 眇數 (Mode) : 找出出現次數最多的值。



# 集合

EMILY in IM



# 集合

- 交集 (Intersection)：找出兩個集合共有的元素（如  $A \cap B$ ）。
- 聯集 (Union)：合併兩個集合的所有元素（如  $A \cup B$ ）。
- 差集 (Difference)：找出  $A$  有但  $B$  沒有的元素（如  $A - B$ ）。
- 子集判斷 (Subset Check)：確認一個集合是否包含於另一個集合（如  $A \subseteq B$ ）。



# 一些常見針對特定資料結構的操作

- 堆積操作 (Heap Operations)：插入、刪除並維持最小/最大堆。
- 樹遍歷 (Tree Traversal)：DFS, BFS, Inorder, Preorder, Postorder (如 遍歷二元樹)。
- 圖搜尋 (Graph Search)：Dijkstra 搜尋最短路徑 (如 Google Maps 導航)。
- 拓撲排序 (Topological Sorting)：有向無環圖 (DAG, Directed Acyclic Graph) 的順序。



# 複雜度 big O

EMILY in IM



# 兩種複雜度

(credit: 李根逸 / DSAP Chapter7)

We typically care most about two types of resources: time and space.

- Time refers to the amount of computation used (CPU).
- Space refers to the amount of memory space used (Memory).

In modern software development, network and power consumption can also be considered as important resources.



# Asymptotic Behavior

- Syntactic sugar 語法糖 (以 Python 舉例)

```
for i in range(n):
    print("Hello", "\n", end="")
```

```
i = 0                      # t1
while i < n:
    print("Hello", end="")
    print("\n", end="")
    i += 1                  # t5
```

$$T(n) = t1 + n * (t2 + t3 + t4 + t5) + t2$$

$$T(n) = a * n + b \text{ for } a, b \geq 0$$



# Asymptotic Behavior

- Syntactic sugar 語法糖

「語法糖」的核心概念，是在語法層面提供更簡單或直觀的寫法，而不影響實際的執行結果。

- 在 C++ 中，更強調編譯器 (Compiler) 產生的「機器語言」相同。
- Python 使用直譯器 (intepreter)，因此通常指「讓程式碼更簡潔，並且可能優化執行方式」



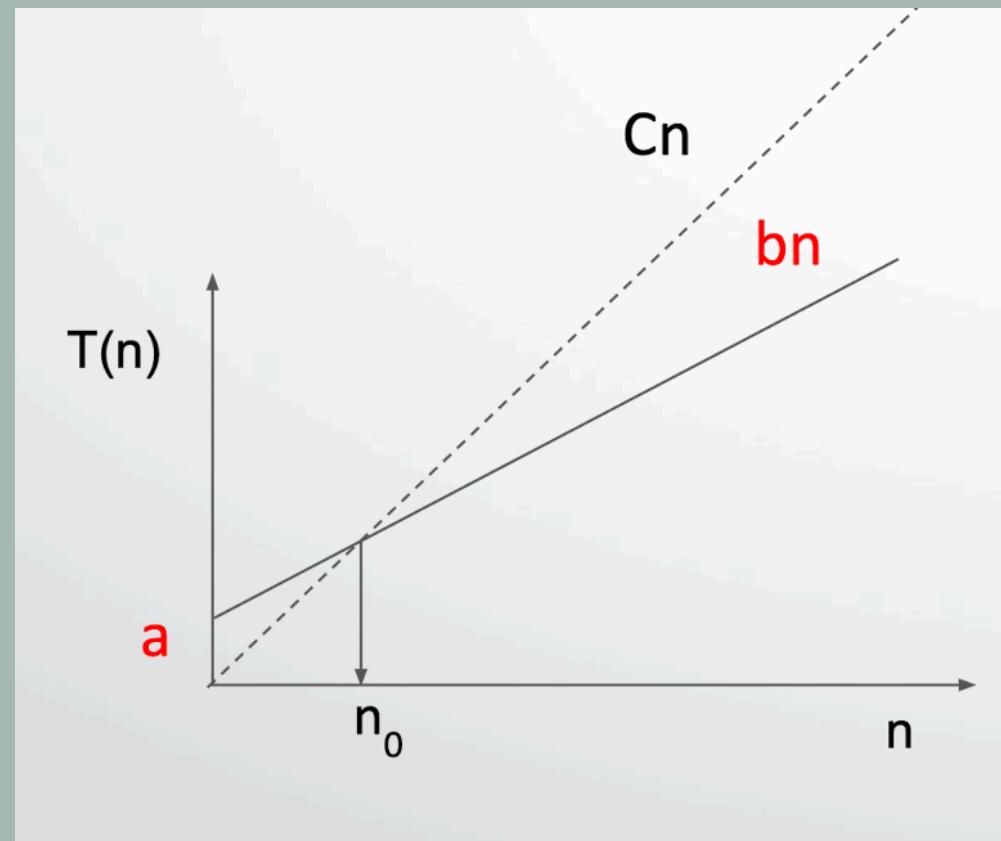
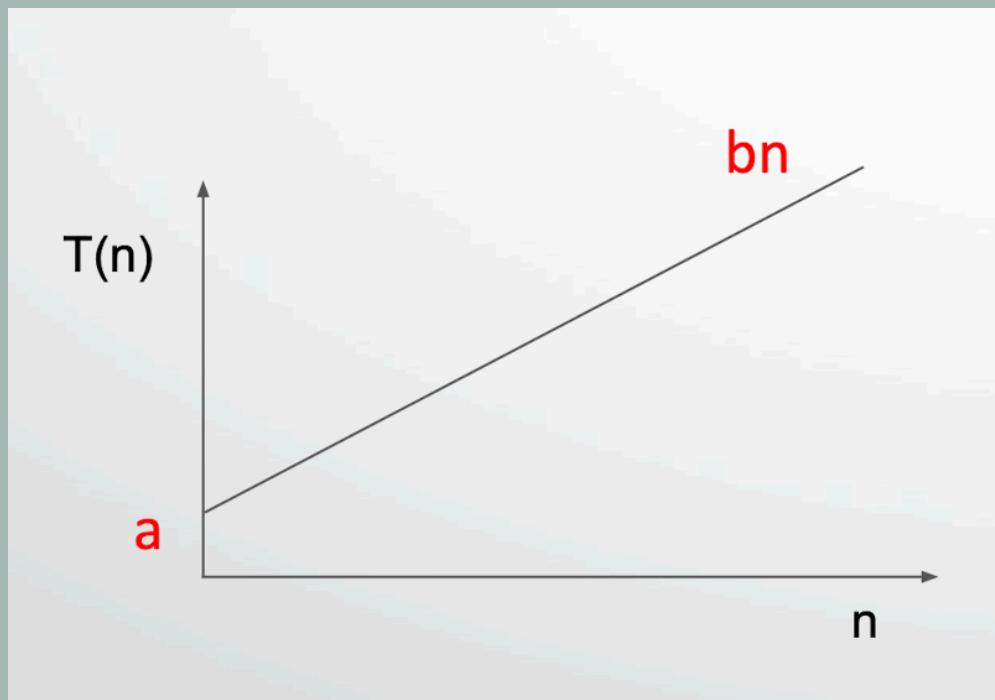
# Big O 定義

$f(n)$  is  $O(g(n))$  if there exists positive constants  $C$  and  $n_0$  such that  $0 \leq f(n) \leq C * g(n)$  for all  $n \geq n_0$ .

So,  $T(n)$  is  $O(n)$ .

只要在某個點之後，都能比  $T(n)$  大就行！

->  $T(n)$  在最糟情況下，頂多就是  $Cn$





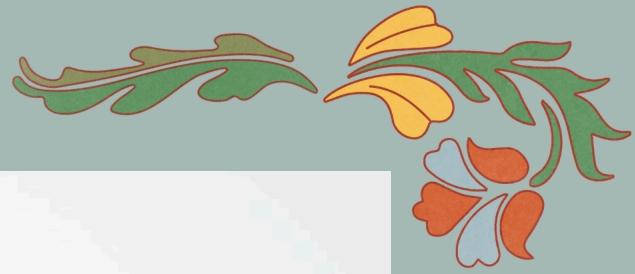
# 更多例子

- $T(n) = n = O(n)$
- $T(n) = n^2 = O(n^2)$
- $T(n) = n + 3 = O(n)$
- $T(n) = 3n + 5 = O(n)$



# 更多例子

- $T(n) = 39 = O(1)$
- $T(n) = n^2 + n + 1 = O(n^2)$
- $T(n) = 10n^2 + 100n + 1 = O(n^2)$
- $T(n) = 9n + 3n^2 + n^3 + 9 = O(n^3)$



## Cut in half ?

```
6 |     while (n > 1) {  
7 |         cout << "Hello" << endl;  
8 |         n /= 2;  
9 |     }
```

O(?)

$$\begin{array}{c} \text{T times} \\ \searrow \quad \swarrow \\ n / 2 / 2 / 2 / 2 / \dots / 2 > 1 \end{array}$$

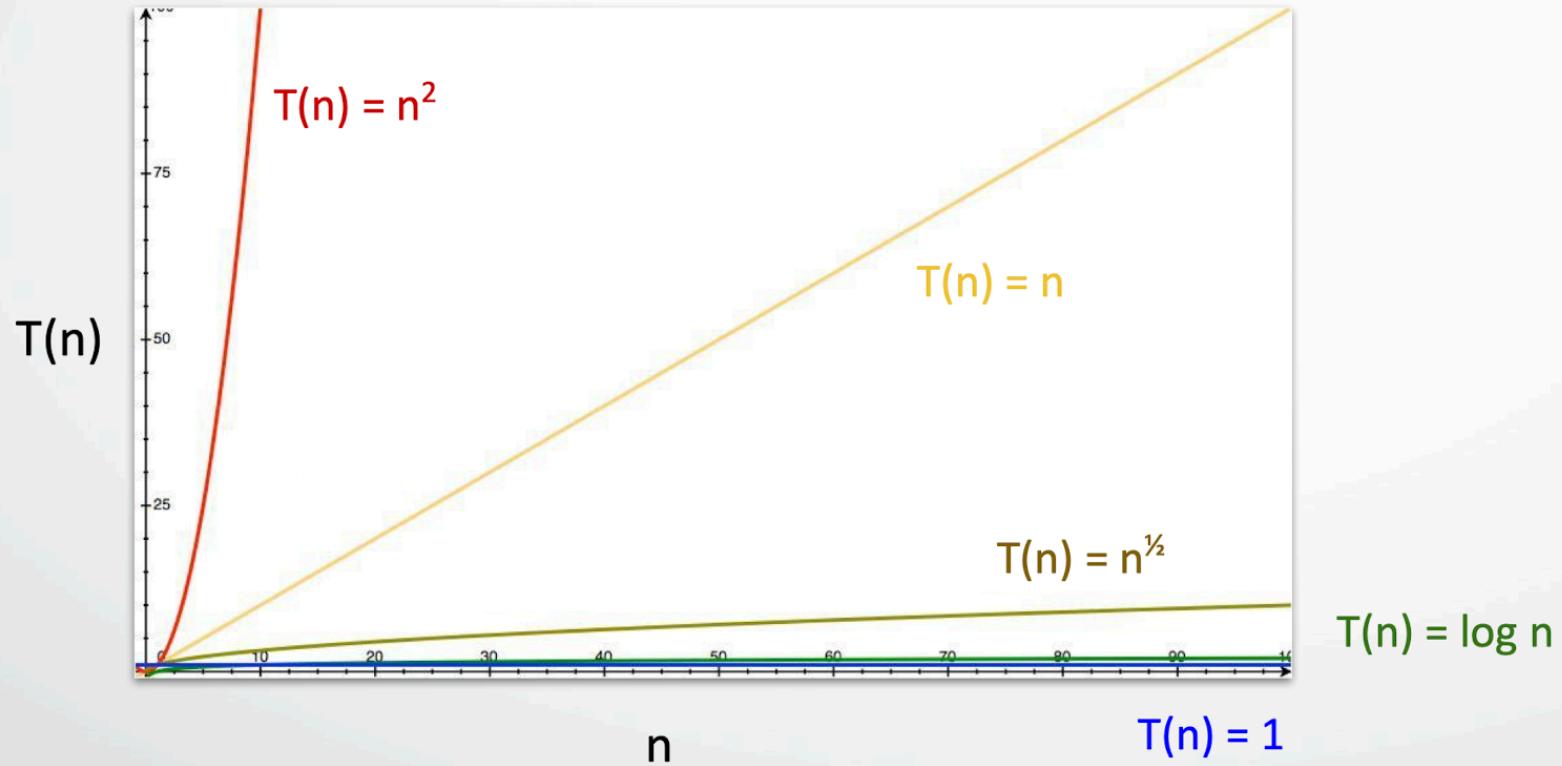
$$n / 2^T > 1$$

$$2^T < n$$

$$T < \log_2 n = O(\log_2 n) = O(\log n) \text{ [Change of Base Formula: } \log_2 n = \log n / \log 2]$$



# Different functions





# 運算

(Credit: 蔡益坤 / Algorithms 03 Analysis)

💡 We can add and multiply with  $O$ .

## Lemma (3.2)

1. If  $f(n) = O(s(n))$  and  $g(n) = O(r(n))$ , then  
 $f(n) + g(n) = O(s(n) + r(n))$ .
2. If  $f(n) = O(s(n))$  and  $g(n) = O(r(n))$ , then  
 $f(n) \cdot g(n) = O(s(n) \cdot r(n))$ .

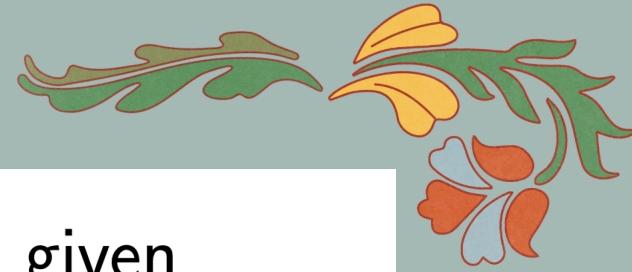
💡 However, we cannot subtract or divide with  $O$ . (Why?)



# Answer

However, we cannot subtract or divide with  $O$ . (Why?)

- ➊  $2n = O(n)$ ,  $n = O(n)$ , and  $2n - n = n \neq O(n - n)$ .
- ➋  $n^2 = O(n^2)$ ,  $n = O(n^2)$ , and  $n^2/n = n \neq O(n^2/n^2)$ .



- ➊ Let  $T(n)$  be the number of steps required to solve a given problem for input size  $n$ .
- ➋ We say that  $T(n) = \Omega(g(n))$  or the problem has a lower bound of  $\Omega(g(n))$  if there exist constants  $c$  and  $N$  such that, for all  $n \geq N$ ,  $T(n) \geq cg(n)$ .
- ➌ If a certain function  $f(n)$  satisfies both  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ , then we say that  $f(n) = \Theta(g(n))$ .
- ➍ We say that  $f(n) = o(g(n))$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ .



# List 複習 (Python Lecture 01)



# 串列 List - 動機

當我們今天想存 3 個類似的數值時...

```
a = 5  
b = 7  
c = 6
```



# 串列 List - 動機

如果今天變成 50 個？

如果當中有兩個變數要 交換順序？

如果我需要從中間 拿掉一個，其他往前遞補？

如果臨時有人從 中間插入？



# 串列 List - 動機

我們需要一個 系統化的容器：

1. 將一串元素們按照順序擺在一起
2. 元素可以各式各樣：整數、字母、字串、布林值
3. 有一些設計好的功能，方便我們使用（例如：變換順序、拿掉其中一個、從中間插入...）



# 串列 List - 宣告

在 Python 中，我們使用 [中括號] 表達串列

創建一個串列可以透過...

```
1. new_list = [0, 1, 2, 3, 4]
```

```
2. another_list = list()
```

兩種方法。



## 試試看

艾蜜莉想要幫自己的三次段考成績，用一個 list 來記錄，成績分別是：95、47、68。

請創建一個名為 score 的串列，將成績儲存起來。



# 串列 List - 取值

在 Python 中，我們使用 [中括號] 表達串列

我們想要取得串列中索引值 = i 的內容...

```
print(new_list)
print(new_list[2]) #注意！誰被印出了？
```



## 想法

在程式中，我們從 0 開始計數，  
所以索引值從 0 開始算。

## 試試看

請幫艾蜜莉印出第二次段考的成績！  
(提示：第二次段考的索引值是多少？)



# 串列 List - 索引值專論

在程式中，我們從 0 開始計數，所以索引值從 0 開始算。

- 一般情況下， $[i]$  表達 第  $i + 1$  個 元素
- 從後方開始取值，使用  $[-i]$  表達 倒數第  $i$  個 元素
- $[i:j]$  表示 從  $i$  (包含) 到  $j$  (不包含) 的元素



試試看

如果是 [i: (留白) ] 以及 [ (留白) :j] 時，範圍是在哪裡？

試試看

當今天變成 [i:j:k] 時，k 有什麼意義？



# 串列 List - 插入資料

從尾端 插入資料：

1. `append( element )` — 將另一個 元素 加在後面
2. `extend( list )` — 將另一個 串列 加在後面



```
# 使用 append
fruits = ['apple', 'banana', 'cherry']
fruits.append('orange')
print(fruits) # 輸出: ['apple', 'banana', 'cherry', 'orange']

# 使用 extend
fruits = ['apple', 'banana', 'cherry']
more_fruits = ['mango', 'pineapple', 'grape']
fruits.extend(more_fruits)
print(fruits) # 輸出: ['apple', 'banana', 'cherry', 'mango', 'pineapple', 'grape']
```



## 試試看

在 `my_list` 的後方加上新的元素 `"New_element"`。

## 試試看

建立一個新的串列 `another_list`，內容物自訂，並且將這個新串列接在原本的 `my_list` 後面。



## 注意

如果使用 `append()` 插入一個 串列 會發生什麼事？

我們將會擁有一個新元素，該元素是一個串列。

```
[1, 2, 3, [4, 5, 6], 87, 0]
```



# 串列 List - 插入資料

從 中間 插入資料：

```
insert( index, element )
```

給定 索引值 `index` 以及 想插入的元素 `element` 作為參數，就可以  
從中間插入新資料！

補充

其餘的資料將會往後順延一格



## 試試看

在每兩次段考之間，還會有一次隨堂小考。

艾蜜莉兩次小考成績分別是：99、107。

請依照時間順序，將小考成績插入 score 串列中，並且將最後的  
score 結果列印出來。



# 串列 List - 移除資料

給定 索引值 移除資料：

```
pop( index )
```

給定 索引值 `index` 作為參數，該位置的資料會被刪除

補充

其餘的資料將會往前順延一格

若未給定參數，預設刪除最後一格的元素



# 試試看

`pop()` 其實會回傳資料！

試試看將 `pop( index )` 放在等號後面，有什麼東西被回傳了？



# 串列 List - 移除資料

給定 想移除的內容：

```
remove( element )
```

給定 元素 element 作為參數，第一個出現的該元素會被刪除

補充

其餘的資料將會往前順延一格



# 串列 List - 複製資料

我們有兩種複製資料的方法：

## 1. Shallow Copy

我們做 `B = A`，看似將串列 A 複製了一份給 B，但是實際上，  
真的如此嗎？



## 試試看

將 `my_list` 按照上頁的方法複製一份給 `copy_list`。

若我們將 `my_list[0]` 的內容移除，試試看將 `copy_list` 列印出來，觀察看看發生什麼問題了？



# 串列 List - 複製資料

我們有兩種複製資料的方法：

## 1. Shallow Copy

我們做 `B = A`，看似將串列 A 複製了一份給 B，但是實際上，  
並不是如此。

A 與 B 此時共用著同一個串列，我們將兩者 引用 到同一個地方。  
因此，改變其中一個串列，會影響到另外一個串列。



# 串列 List - 複製資料

因此，我們介紹第二種複製資料的方法：

## 2. Deep Copy

`B = A[ : ]`，將串列 A 中的每一個元素，都依序複製到一個新的串列 B 中。



# 串列 List - 還有一些功能值得你探索...

## 1. 排列 `list.sort(reverse=True/False)`

我們能夠使用 **sort()** 來排序，其中 **參數** 預設為 `False`，會將串列重新由小到大排序；若我們將參數設定為 `True`，則會由大排到小。



# 串列 List - 還有一些功能值得你探索...

## 2. 長度 `len( list )`

我們能夠藉此方便取得 串列的長度 ，對於後續迴圈的操作十分  
重要！



# 串列 List - 還有一些功能值得你探索...

## 3. 次數 `list.count( element )`

在參數中放入 **元素element**，將會回傳該元素在串列中的 出現  
次數



# 串列 List - Recap

1. Python 的串列，設計了很多好用的功能
2. 但背後的運行邏輯，導致效率極差。
3. 在其他程式語言中，通常將這種 連續記憶體 的串列稱為 陣列  
**(Array)**



# 情境

假設我們要管理一個學校的學生資料，並且不斷地有學生加入或離開。每次加入或刪除學生資料時，我們可能希望儘量不讓程式的運行變得很慢。

還記得語法糖嗎？Python看似簡潔，實際要耗費大量時間。



# 情境

假設現在有 100 個學生資料，我們需要每次插入一名新學生資料到開頭，或者從開頭刪除學生資料。

- 如果使用 Python 的串列 (List)：當使用 `insert()` 或 `pop(0)` 這類型的操作時，Python 需要將所有後續的資料移動一格，才會騰出位置或更新資料。這會使得每次操作的時間複雜度是  $O(n)$ ，也就是 資料量越大，操作越慢。



## 想法

Python 提供許多簡單易用的語法，比如串列推導式、`append()` 方法等，讓我們寫程式變得更簡潔。但這些簡潔的寫法有時候會隱藏背後的運行成本。

因此，雖然使用 `insert()` 和 `pop(0)` 操作看似簡單，但在實際執行時，可能會因為需要移動大量資料而使效能變差。



## 思考

如果我們每次都需要從開頭加入或刪除資料，這會導致什麼情況呢？

如何修改儲存資料的方式，避免運行效率被拖垮呢？



# 指標 Pointer

EMILY in IM



# 指標 Pointer

指標（Pointer）是程式語言中的一個概念，通常出現在像是 C、C++ 的程式語言中。

指標就像是記憶體位置的「地址」，它保存了一個記憶體位置，而這個位置指向某個資料。



# 指標 Pointer

Python 隱藏了大部分內部的記憶體管理，讓我們不需要直接處理指標。

當我們創建一個新變數時，Python 會自動處理如何儲存這些資料，而我們只需要注意變數名稱。



# 鏈結串列 Linked List

EMILY in IM



# 鏈結串列 (Linked List) 簡介

鏈結串列是一種 動態資料結構，它由一系列的 節點 (Node) 組成，每個節點包含兩部分：

1. 資料部分：儲存實際的資料。
2. 指標（或稱為鏈接）部分：指向下一個節點的位置。

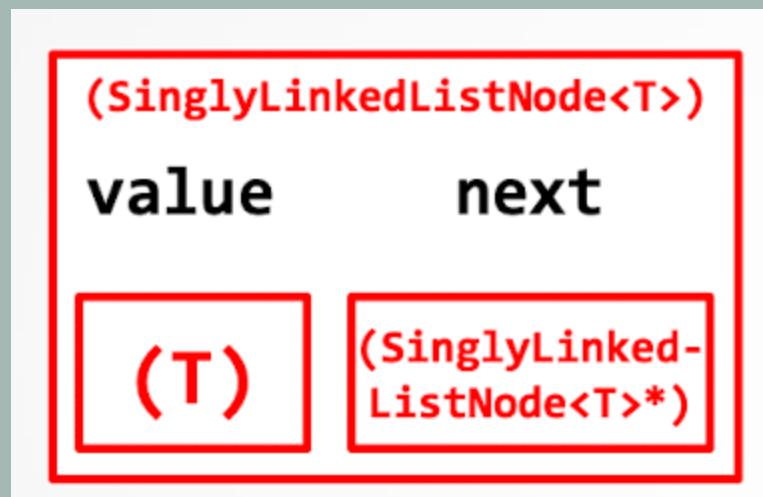


# 基本結構

每個節點有兩個部分：資料和指向下一個節點的指標。

首節點 (Head) 是鏈結串列的開始，它指向第一個節點。

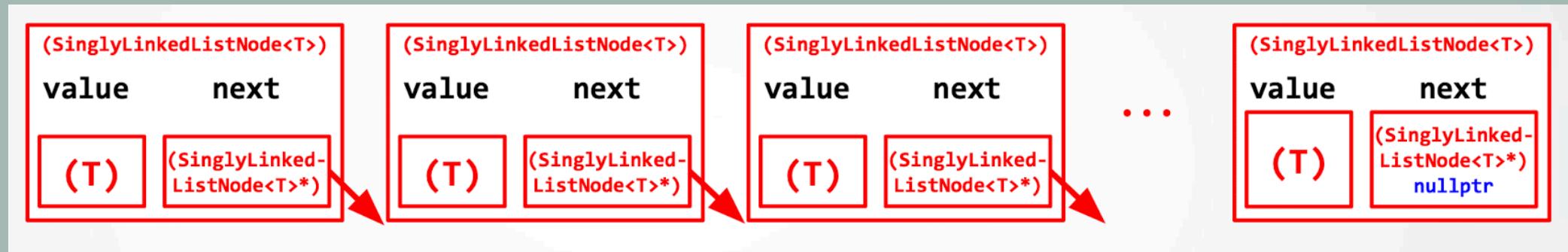
尾節點 (Tail) 指向最後一個節點，且其指標通常為 None，表示鏈結串列結束。



EMILY in IM



# 鏈結：指向下一個節點的位置





# 主要操作

插入 (Insert) : 可以在鏈結串列的任意位置插入新的節點 (例如：在開頭、中間、尾部) 。

- 插入新的節點需要修改前一個節點的指標，讓它指向新插入的節點。



# 主要操作

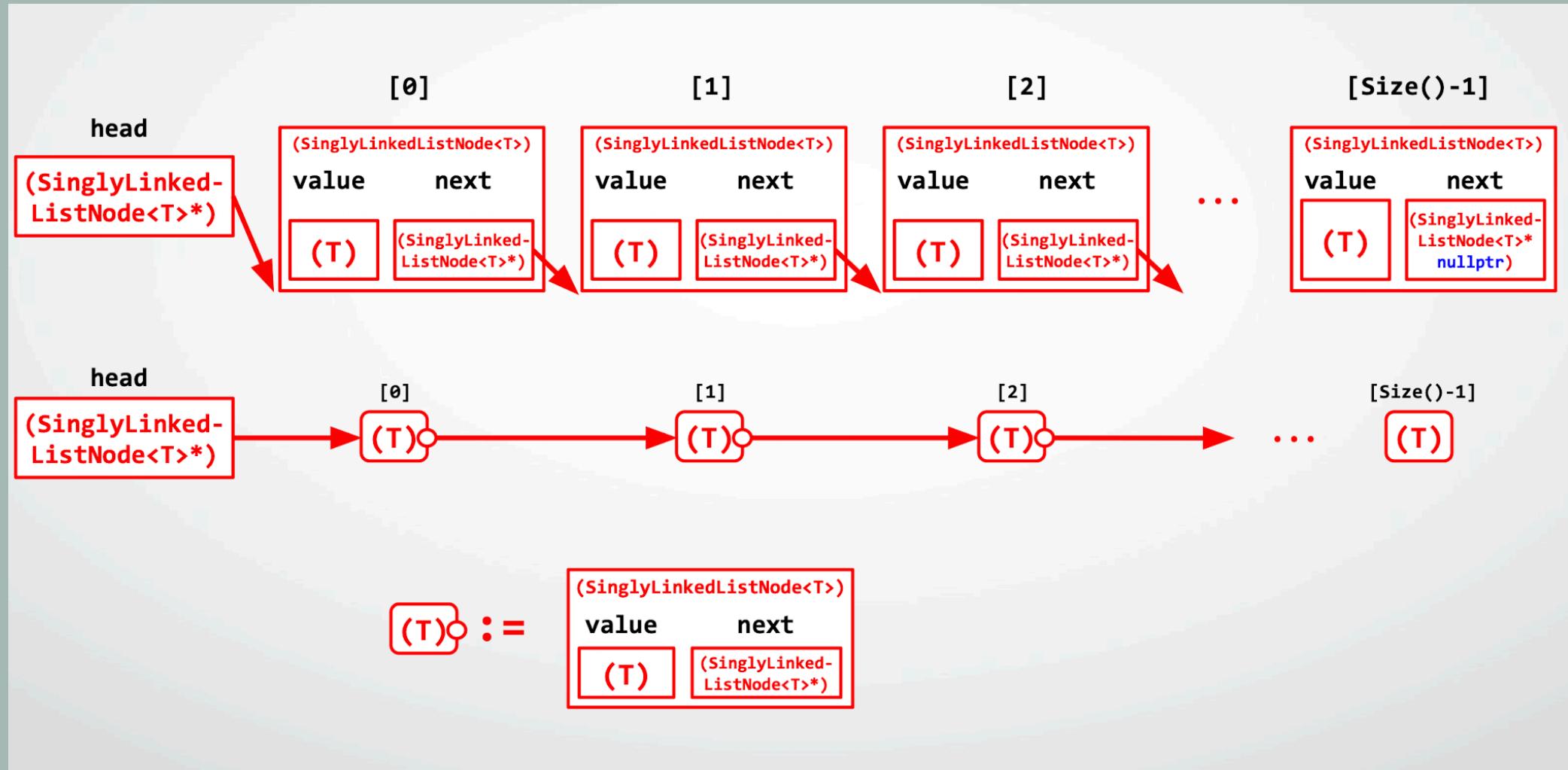
刪除 (Delete)：可以從鏈結串列的任意位置刪除節點，並且調整指標來保持鏈結串列的連貫性。

- 刪除節點後，上一個節點的指標需要更新為指向被刪除節點的下一個節點。



# 主要操作

遍歷 (Traverse)：從首節點開始，依次訪問每一個節點，直到遇到 None (尾節點) 為止。





# 優缺點？

EMILY in IM



# 優點

1. 動態大小：與陣列不同，鏈結串列的大小不是固定的，可以根據需要動態地增加或減少節點。
2. 插入和刪除操作效率高：在鏈結串列中，插入或刪除節點的時間複雜度通常為  $O(1)$ ，只需要改變指標，而不需要移動其他資料。



# 缺點

1. 隨機存取不方便：不像陣列那樣能直接通過索引存取資料，必須從頭節點開始一個一個遍歷。
2. 額外的記憶體開銷：每個節點都需要存儲資料和指標，增加了額外的記憶體開銷。



# Recap

串列 (List) 有兩種實作方法：

- 陣列式 (Array-based List)
- 鏈結式 (Linked-based List)



# 陣列式 (Array-based List)

- 連續記憶體，存取速度快 ( $O(1)$ )
- 索引 (Indexing) 容易，可直接存取元素
- 插入/刪除慢，需要搬移元素 ( $O(n)$ )
- 大小固定，需事先定義容量

例如：Python list (底層實作是動態陣列)



# 鏈結式 (Linked-based List)

- 動態大小，不需事先分配空間
- 插入/刪除快 ( $O(1)$ ，僅需改變指標)
- 存取慢，需從頭開始逐個搜尋 ( $O(n)$ )
- 額外指標空間，比陣列佔用更多記憶體



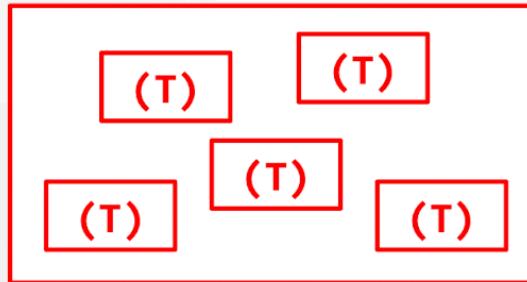
# 一些進階的資料結構

## Set & Dictionary



# Recap (Credit: 李根逸 / DSAP Chapter5)

- 集合 Set
- Data:
  - A finite number of **distinct** objects, having the same data type





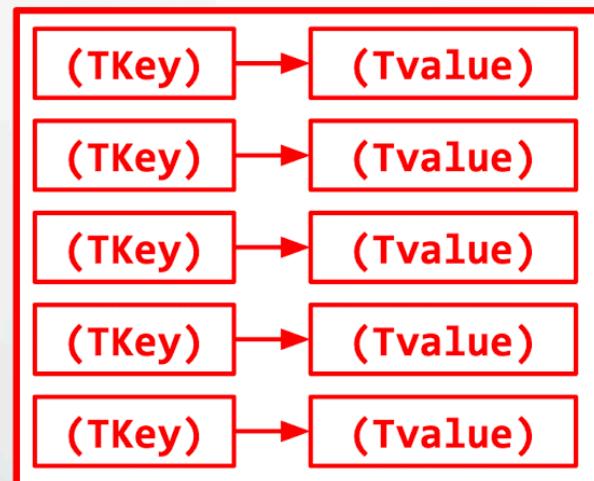
# Recap (Credit: 李根逸 / DSAP Chapter5)

- 集合 Set
- Operations:
  - **Build(objs)**: Create a set from a list of objects.
  - **Empty()**: See whether the set is empty.
  - **Size()**: Get the number of items currently in the set.
  - **Add(obj)**: Add a given object to this set.
  - **Remove(obj)**: Remove a particular object from this set, if possible.
  - **Clear()**: Remove all objects from this set.
  - **Contains(obj)**: Test whether this set contains a particular object.



# Recap (Credit: 李根逸 / DSAP Chapter5)

- 字典 Dictionary
- Data:
  - A finite number of objects, each associated with a distinct search **key**





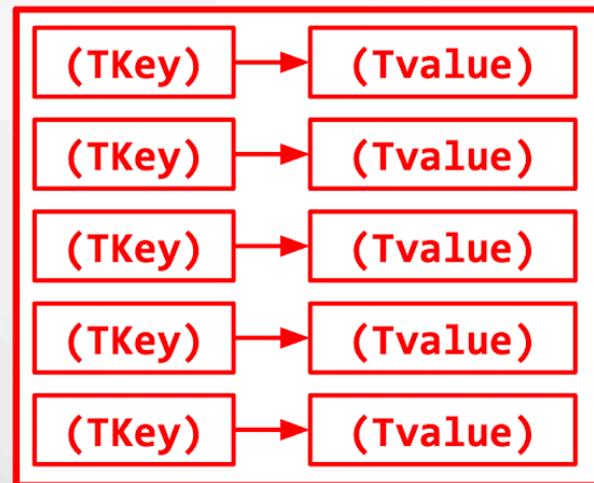
# Recap (Credit: 李根逸 / DSAP Chapter5)

- 字典 Dictionary
- Operations:
  - **Build(kvps)**: Create a dictionary from a list of key-value pairs.
  - **Empty()**: See whether this dictionary is empty.
  - **Size()**: Get the number of items in this dictionary.
  - **Add(key, item)**: Insert an item into this dictionary according to the item's search key.
  - **Remove(key)**: Remove the item with the given search key from this dictionary.
  - **Clear()**: Remove all entries from this dictionary.
  - **Contains(key)**: See whether this dictionary contains an item with a given search key.



# Recap (Credit: 李根逸 / DSAP Chapter5)

- 字典 Dictionary
- Operations:
  - **GetItem(key):** Get an item with a given search key from this dictionary.





# Set / Dictionary 複習 (Python Lecture 02)

By 林杰 @2025臺大資管營



# 元組 Tuple

EMILY in IM



# 元組 Tuple - 動機

tuple 和 list 是非常相似的東西，不過 tuple 在建立之後就無法更改裡面的資料。

為什麼不直接用 list 就好？

1. Tuple 是不可變的，可以防止資料被意外更改。
2. Tuple 可以作為字典的鍵 (key)，在下一部分會說明。



# Tuple 的應用

我們可能會將什麼東西存進 tuple 中呢？

1. 平面座標： coordinate = (2, 4)

2. RGB 顏色表示： color = (255, 255, 0)

另一個可以認識 tuple 的原因：

tuple 這個結構和離散數學與自動機理論中的許多理論定義與操作有關，適合表達「不需改變」的數學結構。



# 元組 Tuple - 宣告賦值

在 Python 中，我們使用（小括號）表達元組。

創建一個元組可以透過...

1. `a_tuple = tuple()`

2. `a_tuple = ()`

3. `coordinate = (0, 0)`



# 元組 Tuple - 取值

在 Python 中，我們使用 (小括號) 表達元組。

取得元組中索引值 **i** 的內容...

```
coordinate = (0, 0)

print(coordinate) # 印出 (0, 0)
print(coordinate[0]) # 印出 0
```



# 元組 Tuple - 注意事項

tuple 和 list 是非常相似的東西，不過 tuple 在建立之後就無法更改裡面的資料。

所以我們不能對一個 tuple 新增元素、刪除元素。



# 集合 Set

EMILY in IM



# 集合 Set - 動機

集合是數學中一個十分常用的概念，它用來描述一組明確而互不重複的對象。

在 Python 中，set 是一個裝資料的大容器：

- 同樣的值只出現一次
- 裡面裝的元素之間是沒有位置關係的
- 學習用 Python 操作 Set，順便複習集合 :)



# 集合 Set - **add()**

使用 `add()` 可以讓我們在集合中加入一筆資料。

```
a_set = set() # 空集合  
a_set.add(10) # 添加元素 10  
a_set.add(20) # 添加元素 20  
a_set.add(10) # 再次添加 10(無效，集合不允許重複)  
a_set # ???
```

## Q&A

上面 `a_set` 這個集合中，有哪些元素？



# 集合 Set - `remove()`

`remove()` 方法從集合中刪除指定的元素。如果該元素不存在，會拋出 `KeyError`。

```
a_set = {1, 2, 3, 4}  
a_set.remove(3) # 移除元素 3  
print(a_set) # {1, 2, 4}  
a_set.remove(5) # KeyError
```



# 集合 Set - **discard()**

`discard()` 方法從集合中刪除指定的元素。如果該元素不存在，不會拋出錯誤。

```
a_set = {1, 2, 4}  
a_set.discard(5) # 沒有錯誤
```



# 集合 Set - 聯集與交集

Factorial Image



# 集合 Set - **union()**

`union()` 方法用於返回兩個集合的聯集，包含兩個集合所有唯一元素的集合。

```
a_set = {1, 2, 3}  
b_set = {3, 4, 5}  
result = a_set.union(b_set) # 返回兩集合的聯集  
print(result) # {1, 2, 3, 4, 5}
```



# 集合 Set - intersection()

intersection() 方法用於返回兩個集合的交集，即同時存在於兩個集合中的元素。

```
a_set = {1, 2, 3}  
b_set = {3, 4, 5}  
result = a_set.intersection(b_set) # 返回兩集合的交集  
print(result) # {3}
```



# 字典 Dictionary

EMILY in IM



# 字典 Dictionary - 介紹

- 字典是由鍵值對 (key-value pairs) 組成的無序集合。
- 鍵是唯一的，而值可以是任何資料類型。
- 可以通過鍵來存取對應的值。

## Mapping (keys → values)

- "Name" → "Tran"
- "Age" → 37
- "Address" → "Vietnam" **EMILY in IM**



# 字典 Dictionary - 宣告賦值

在 Python 中，我們使用 {大括號} 表達元組。

- 創建一個空字典： my\_dict = {} # 空字典
- 使用鍵值對初始化字典：

```
person = {"name": "Alice",
           "age": 25,
           "city": "New York"}
print(person) # {'name': 'Alice', 'age': 25, 'city': 'New York'}
```



# 字典 Dictionary - 取值

```
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

當我們想要取出某個 key 所對應到的 value，我們可以用  
`dict[key]`。

例如，如果想要求出 name 這個 key 所對應的 value，我們可以這樣寫：

```
person['name'] # 程式就會回傳 'Alice'
```



# 字典 Dictionary - 更新

```
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

字典更改和增加的語法一模一樣！

如果我們想要將 age 的值改為 30，可以這樣寫：

```
person["age"] = 30 # 更改 age 的值
```

如果我們想要新增鍵值對，可以這樣寫：

```
person["school"] = "NTU"
```



# 字典 Dictionary - 更新

person 就變成了

```
{'name': 'Alice', 'age': 30, 'city': 'New York', 'school': 'NTU'}
```



# 字典 Dictionary - 刪除元素

我們有兩種方法可以刪除字典中的元素：

## 1. 使用 `del` 刪除指定鍵：

```
del person["city"] # 刪除 "city" 這個鍵及其對應的值  
print(person) # {'name': 'Alice', 'age': 30, 'school': 'NTU'}
```



# 字典 Dictionary - 刪除元素

## 2. 使用 `pop()` 刪除指定鍵：

```
age = person.pop("age") # 刪除並返回 "age" 的值  
print(person) # {'name': 'Alice', 'school': 'NTU'}  
print("Age:", age) # 30
```

現在 `person` 變成 `{'name': 'Alice', 'school': 'NTU'}`。



# 字典 Dictionary - 取出鍵或值

在某些應用中，我們只想取出字典中的鍵或值：

使用 `keys()` 取出所有鍵：

```
keys = person.keys()  
print(keys) # dict_keys(['name', 'school'])
```

在這裡，`keys` 的資料型態是 `dict_keys`。



# 字典 Dictionary - 取出鍵或值

使用 **values()** 取出所有值：

```
values = person.values()  
print(values) # dict_values(['Alice', 'NTU'])
```



# 字典 Dictionary - 取出鍵與值

在某些應用中，我們只想取出字典中的鍵或值：

使用 **items()** 取出所有鍵值對

```
items = person.items()  
print(items) # dict_items([('name', 'Alice'), ('school', 'NTU')])
```

## 備註

資料型態 `dict_keys`、`dict_values` 和 `dict_items` 都是可以迭代的物件，可以用 `for` 迴圈取出所有元素。



# 樹 Tree

EMILY in IM



# 樹 Tree

樹（Tree）是一種階層式的資料結構，由節點（Node）組成，常用來表示具有層級關係的資料，像是家譜、組織結構等。

與鏈結串列（Linked List）不同，樹的每個節點可以有多個子節點，形成立分枝結構。



# 基本概念

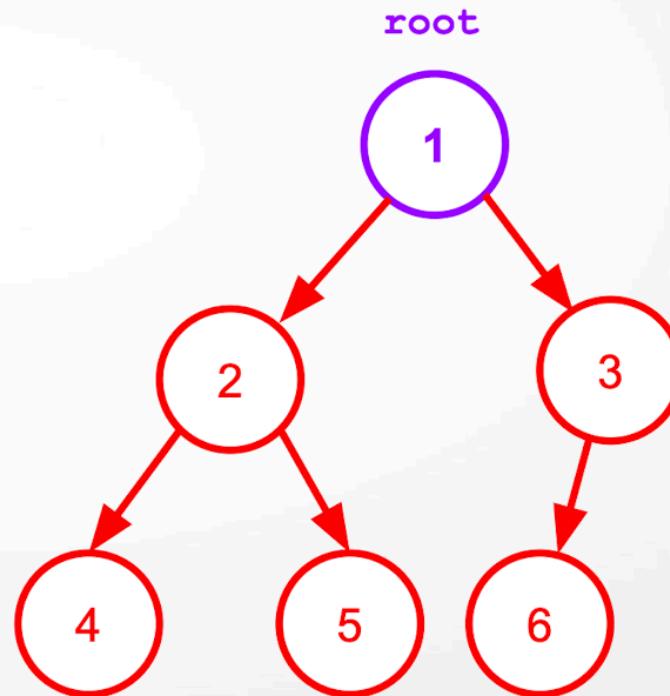
- 根節點 (Root)：樹的最上層節點，沒有父節點。
- 子節點 (Child)：從某個節點延伸出的節點。
- 父節點 (Parent)：直接連結子節點的節點。
- 葉節點 (Leaf)：沒有子節點的節點。
- 深度 (Depth)：某節點到根節點的距離（層數）。
- 高度 (Height)：從某節點到最深葉節點的距離。



# 基本概念

## The root node

- Each node in the tree can be connected to **many children**, but must be connected to exactly **one parent**, except for the **root** node.
  - The root node has no parent.

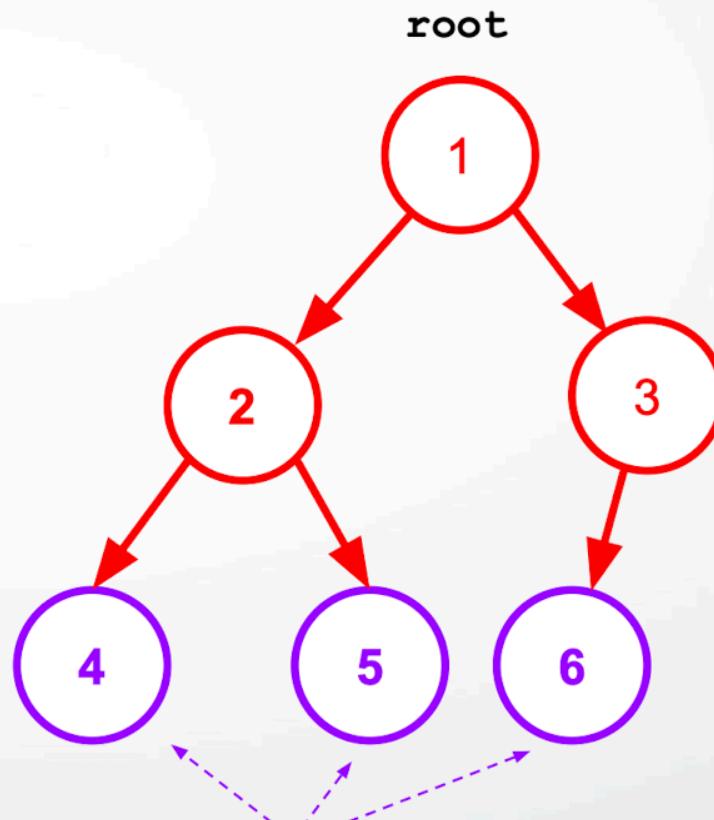




# 基本概念

## The leaf node

- Each node in the tree can be connected to **many children**, but must be connected to exactly **one parent**, except for the **root** node.
  - An leaf node (also known as an external node or terminal node) has no children.

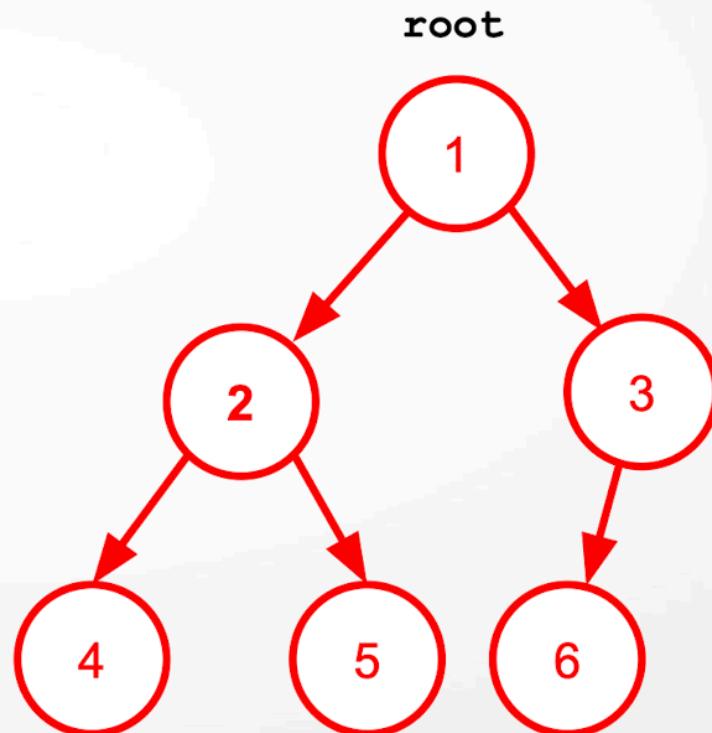




# 基本概念

## The number of edges in a tree

- Each node in the tree can be connected to **many children**, but must be connected to exactly **one parent**, except for the **root** node.
  - A tree with **N** nodes will have **N-1** edges.
    - Aside from the root, every node has an edge pointing to it.



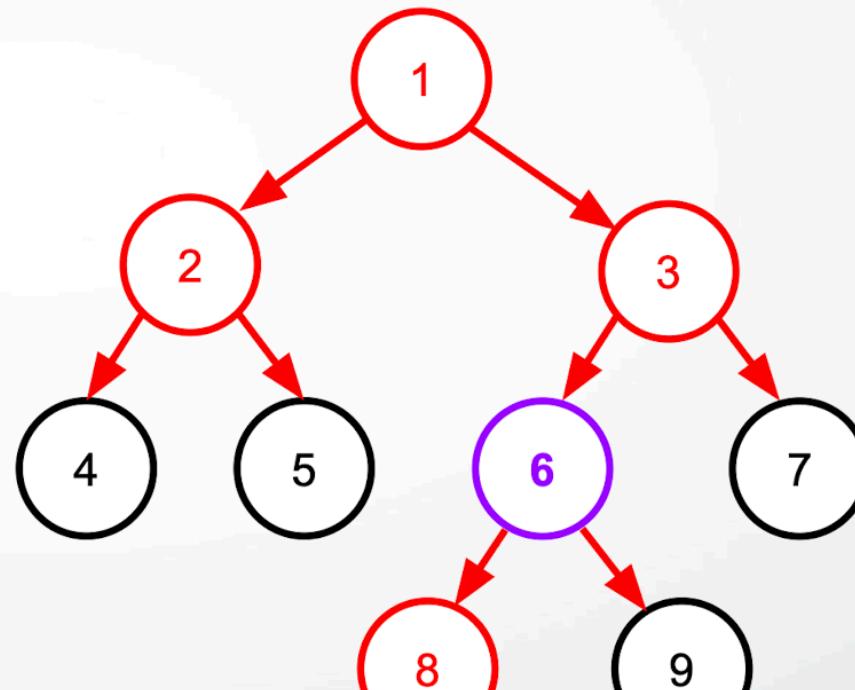


# 基本概念

- 沿著箭頭走，永遠不可能走回自己

## Relationship between nodes [3]

- There is **no cycles** in a tree:
  - No node can be **its own ancestor**.
  - There is **exactly one path** between the root and any node.



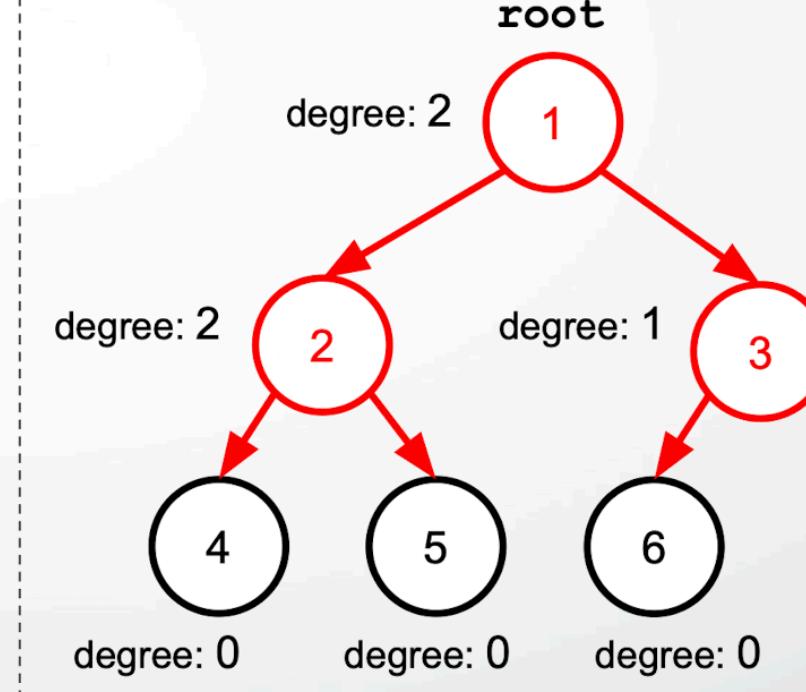


# 基本概念

## Degree

- The degree of a **node** is the number of its children.
  - A leaf has necessarily degree zero.
- The degree of a **tree** is the maximum degree of a node in the tree.

degree of the tree: 2





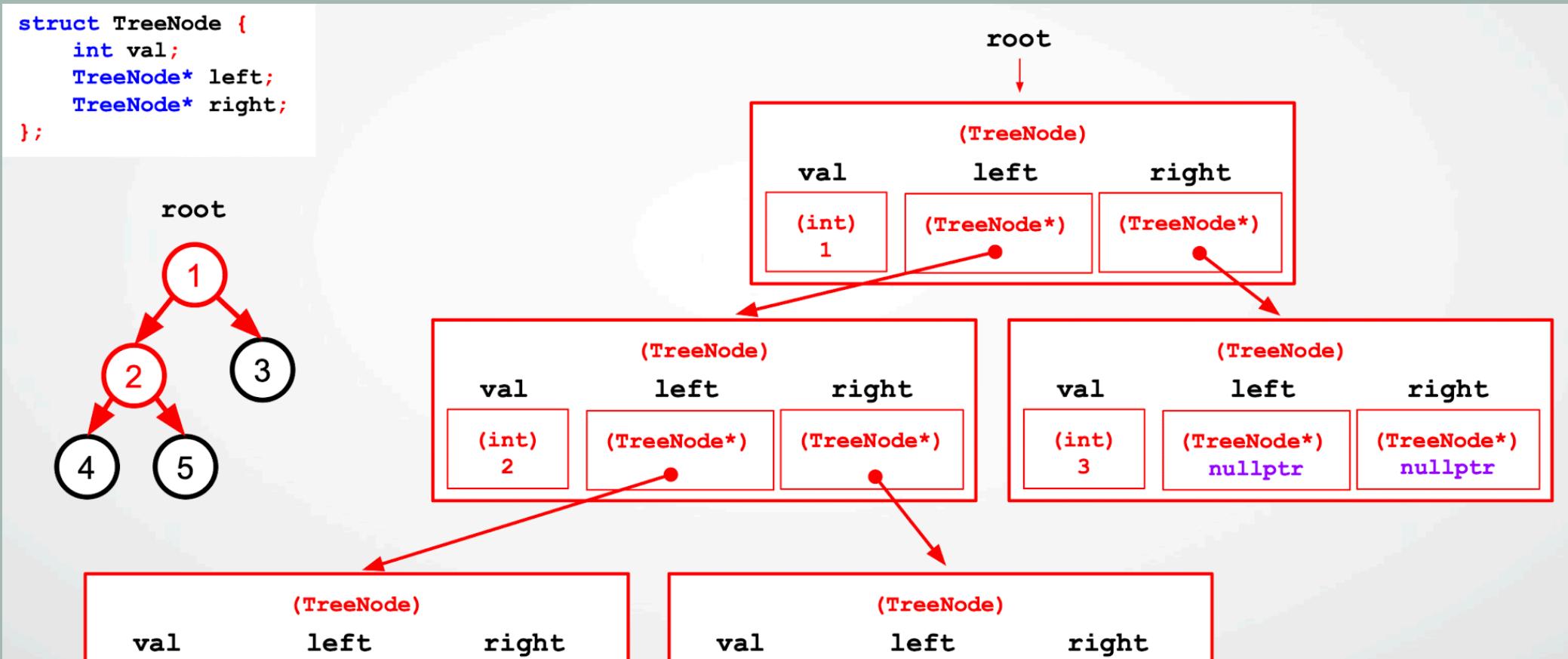
# 二元樹 Binary Tree

EMILY in IM



# 二元樹 Binary Tree

- 每個節點最多只能有兩個子節點（左子節點和右子節點）。

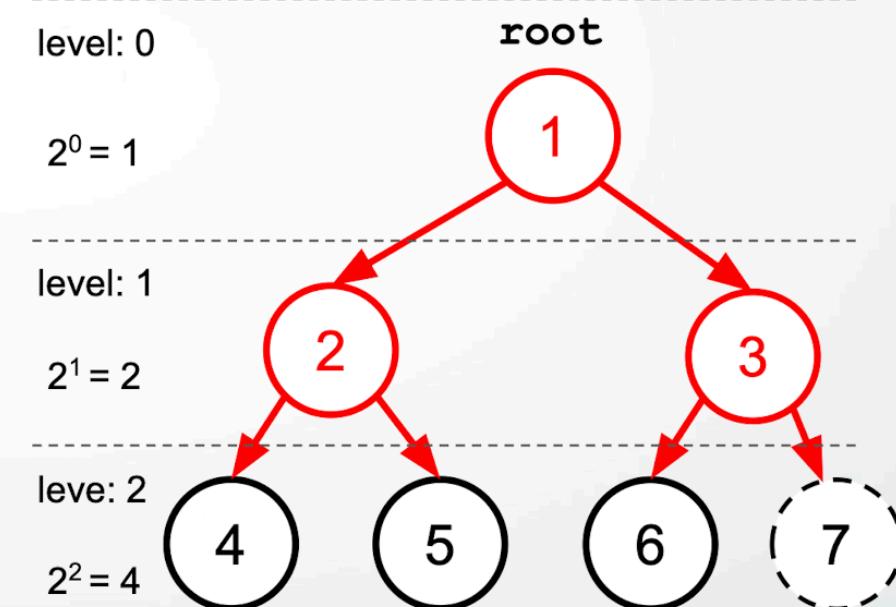




# level

## The level of a node [2]

- The level of a node is the distance between it and the root node
  - There are at most  $2^k$  nodes at level k.
  - There are at most  $2^0 + 2^1 + 2^2 + \dots + 2^k = 2^{k+1} - 1$  nodes at level k and below

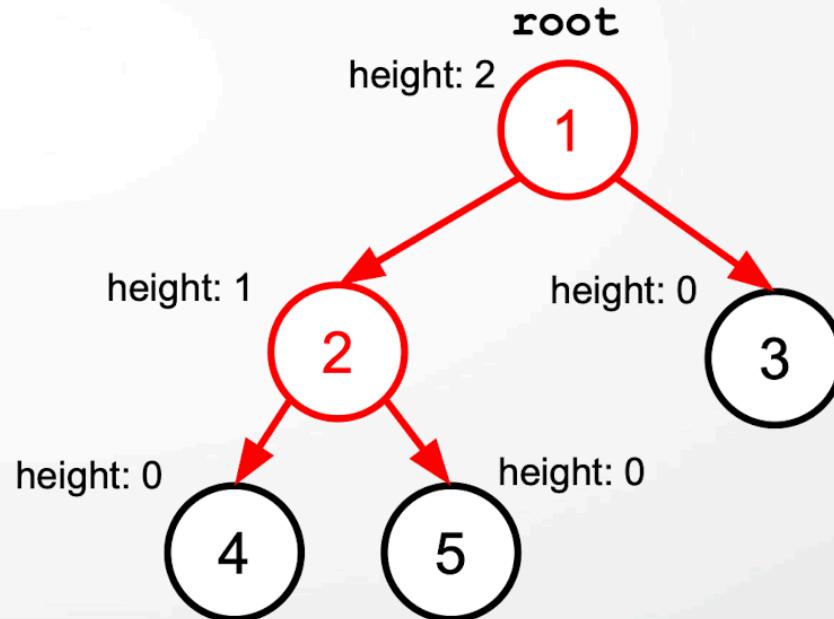




# height

## The height of a node

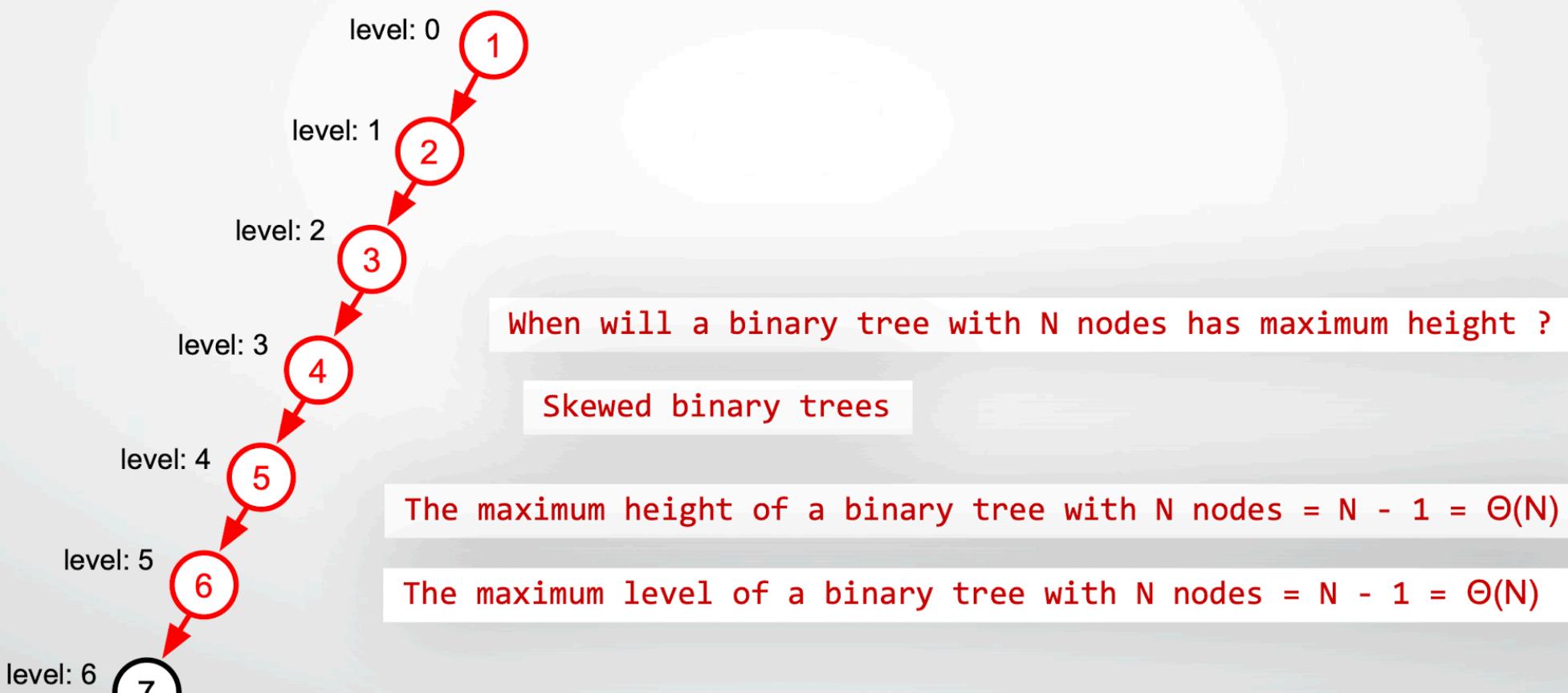
- The height of a node is the number of edges along **the longest downward path to a leaf** from that node.





# 糟糕的情况

## The maximum height of binary trees





# 如何確保效率？

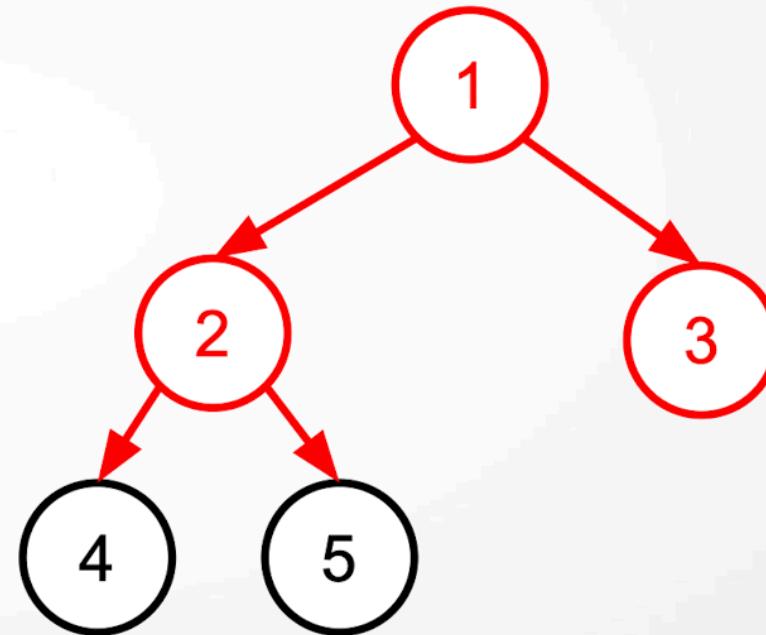
Tree 應該要盡可能保持平衡 (balanced) !

接下來我們將介紹一些特殊的二元樹，它們擁有一些設計好的規則，以完成特定的目的...



# 完全二元樹 Complete Binary Tree

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

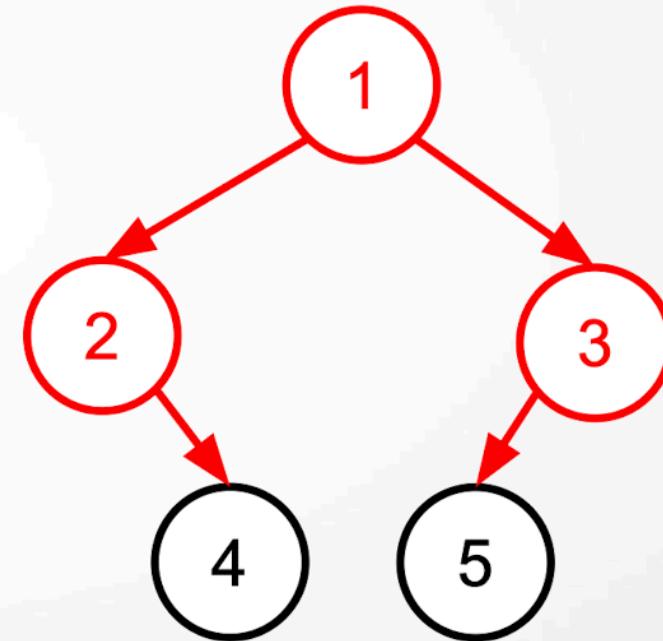


好處：可以使用 List 來處理！



# 平衡二元樹 Balanced Binary Tree

A balanced binary tree is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1



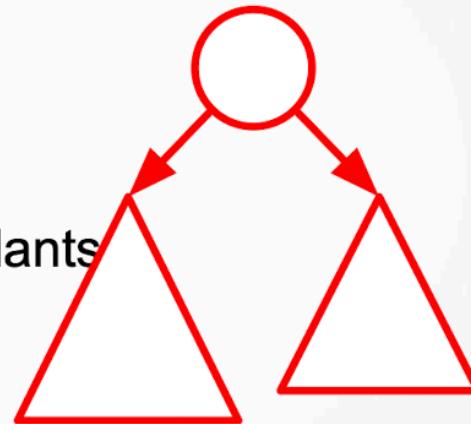
好處：可以避免出現極端的圖形！



# 堆積樹 Heap

## Heap

- A heap is a complete binary tree
- Max heap :
  - Each node in the tree is larger than all of its descendants
    - The root of a max heap is the maximum node.
- Min heap :
  - Each node in the tree is less than all of its descendants
    - The root of a min heap is the minimum node.





# Heapify

Heapify 是一種調整二元樹的方法，讓一棵樹能滿足「堆積 (Heap)」的要求。

- 前提：左右子樹已經是堆積 (heap) 了
- 讓根節點的值「向下浮動」(float down)，直到符合堆積的規則



# 二元搜尋樹 BST (Binary Search Tree)

BST 是一種特殊的 二元樹 (Binary Tree) ，遵守以下規則：

- 左子樹的值 < 根節點
- 右子樹的值 > 根節點
- 每個子樹本身也是 BST



# 二元搜尋樹 BST (Binary Search Tree)

正如同名字一樣，是為了搜尋內容而設計的。

看似效率很高，但是依舊可能會有極端情形出現。

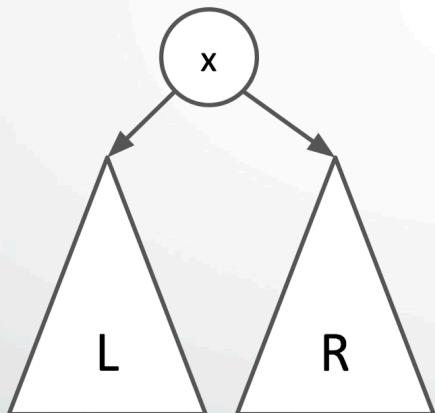


# 二元平衡搜尋樹

確保 時間複雜度 穩定、避免出現極端情形。

Height-balanced binary search tree

- For all nodes,  $| \text{Height}(L) - \text{Height}(R) | \leq 1$





# 二元平衡搜尋樹

確保 時間複雜度 穩定、避免出現極端情形。

- 左子樹、右子樹，高度最多差 1 。
- 具體實現方法：參考 **AVL Tree**、紅黑樹



# Recap

Balanced BST 具有 BST 的規則，但額外保證：

- 樹的左右子樹高度相近
- 避免變成單邊長鏈，確保  $O(\log n)$  的查找時間



# 一些結語

EMILY in IM



# 還有很多東西沒提到...

- 資料的儲存與操作方式：除了 **陣列 (Array)** 和 **鏈結串列 (Linked List)**，還有 **佇列 (Queue)** 和 **堆疊 (Stack)**，它們提供不同的資料存取規則，適用於各種應用場景。
- 樹的遍歷 (Tree Traversal)：遍歷樹的方法包括 **深度優先搜尋 (DFS)** 和 **廣度優先搜尋 (BFS)**，以及 **中序 (Inorder)**、**前序 (Preorder)**、**後序 (Postorder)** 遍歷，這些方法在處理二元樹時尤其重要。



# 還有很多東西沒提到...

- 樹只是開端，圖論（Graph）更關鍵：雖然樹是基本的階層式結構，但現實世界的許多應用，如社交網絡、地圖導航、網路連線，實際上都是圖（Graph），因此 圖論 才是現代應用的核心。



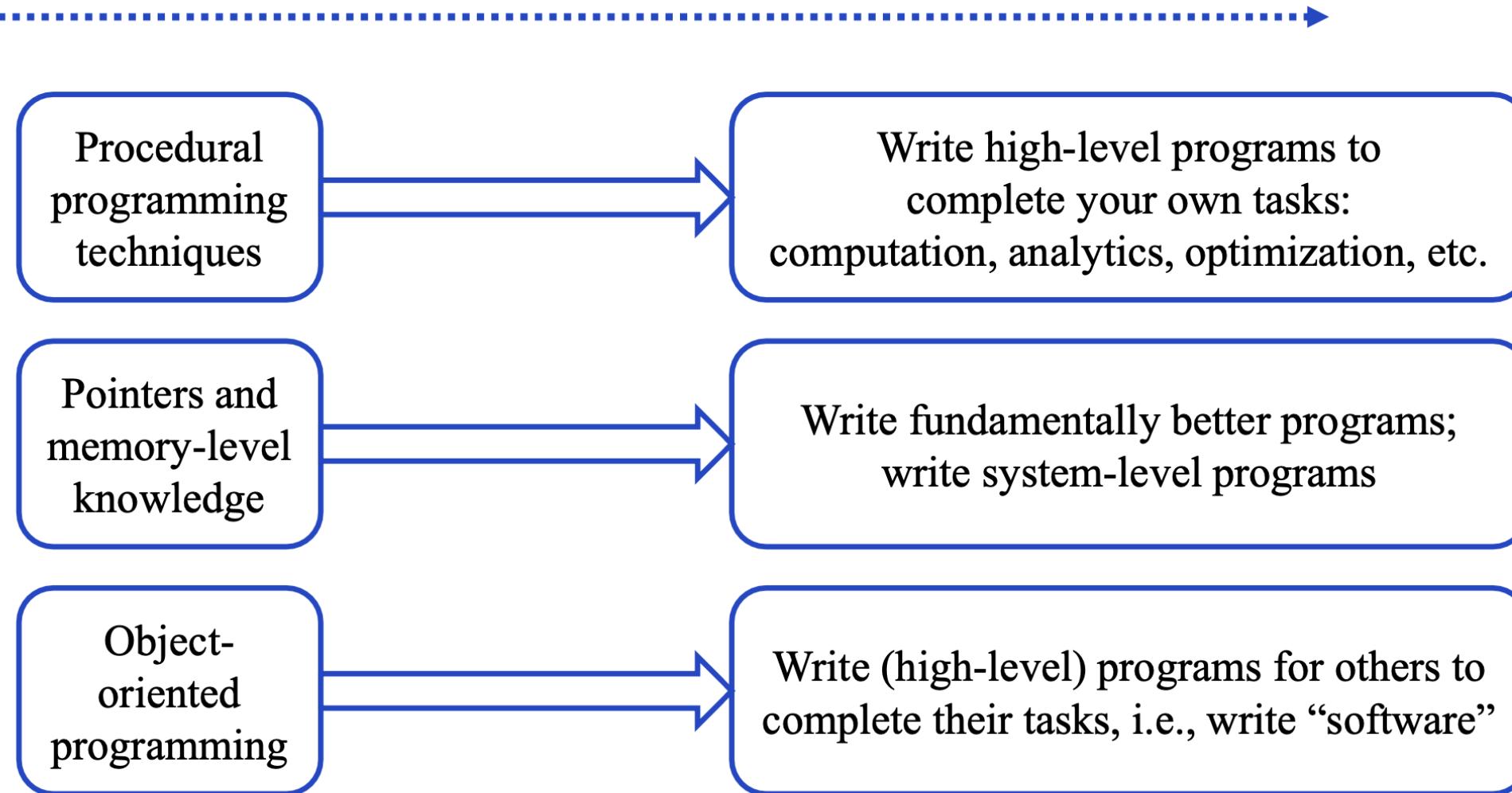
# 自學資源、後續學習

很多關鍵字，網路上都有筆記。

就像逐步拆炸彈，善用 AI 一步步學習新名詞！



# Things that you will be able to do





# The End

(Thanks!)

EMILY in IM