import heapq

from collections import Counter

# 節點類別：代表 Huffman Tree 的每一個節點

class HuffmanNode:

def \_\_init\_\_(self, char=None, freq=0):

self.char = char # 字元（leaf 節點才有）

self.freq = freq # 出現頻率

self.left = None # 左子節點

self.right = None # 右子節點

def \_\_lt\_\_(self, other):

return self.freq < other.freq # 讓 heapq 能比較大小

# 建立 Huffman Tree

def build\_huffman\_tree(freq\_map):

priority\_queue = [HuffmanNode(char, freq) for char, freq in freq\_map.items()]

heapq.heapify(priority\_queue)

while len(priority\_queue) > 1:

left\_node = heapq.heappop(priority\_queue)

right\_node = heapq.heappop(priority\_queue)

merged\_node = HuffmanNode(freq=left\_node.freq + right\_node.freq)

merged\_node.left = left\_node

merged\_node.right = right\_node

heapq.heappush(priority\_queue, merged\_node)

return priority\_queue[0] # 根節點

# 建立 Huffman 編碼表

def generate\_codes(node, prefix="", huffman\_codes=None):

if huffman\_codes is None:

huffman\_codes = {}

if node:

if node.char: # 是葉節點

huffman\_codes[node.char] = prefix

generate\_codes(node.left, prefix + "0", huffman\_codes)

generate\_codes(node.right, prefix + "1", huffman\_codes)

return huffman\_codes

# 壓縮文字：轉為 Huffman 編碼

def compress(input\_text, huffman\_codes):

return ''.join(huffman\_codes[char] for char in input\_text)

# 解壓縮 Huffman 編碼為原始文字

def decompress(encoded\_binary, root):

result = []

current = root

for bit in encoded\_binary:

current = current.left if bit == '0' else current.right

if current.char is not None:

result.append(current.char)

current = root # 回到根節點繼續解碼

return ''.join(result)

# === 測試 ===

input\_text = "SeatA3Ticket12"

freq\_map = Counter(input\_text)

root\_node = build\_huffman\_tree(freq\_map)

huffman\_codes = generate\_codes(root\_node)

encoded\_binary = compress(input\_text, huffman\_codes)

decoded\_text = decompress(encoded\_binary, root\_node)

# === 輸出結果 ===

print("🔹 Original Text:", input\_text)

print("🔸 Huffman Codes:")

for char, code in huffman\_codes.items():

print(f" '{char}': {code}")

print("🔹 Encoded Binary:", encoded\_binary)

print("🔹 Decoded Text: ", decoded\_text)

# === 壓縮比例 ===

original\_bits = len(input\_text) \* 8

compressed\_bits = len(encoded\_binary)

print("\n📊 Compression Ratio:")

print(f" Original size: {original\_bits} bits")

print(f" Compressed size: {compressed\_bits} bits")

print(f" Compression Rate: {compressed\_bits / original\_bits:.2%}")

#--------------------------------------------------------------------------

# Conway's Game of Life（逐步版本，無動畫）

# 顯示每一代結果，讓使用者決定是否繼續

# 初始 10x20 網格，中央顯示 "AI"

grid = [

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

[0,0,1,1,0,0,0,0,1,0,0,0,0,1,1,1,0,0,0,0],

[0,1,0,0,1,0,0,1,0,1,0,0,1,0,0,0,1,0,0,0],

[0,1,1,1,1,0,0,1,1,1,0,0,1,0,0,0,1,0,0,0],

[0,1,0,0,1,0,0,1,0,1,0,0,1,0,0,0,1,0,0,0],

[0,1,0,0,1,0,0,1,0,1,0,0,1,1,1,1,1,0,0,0],

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],

]

rows = len(grid)

cols = len(grid[0])

# 計算某格子的活鄰居數量

def count\_neighbors(y, x):

directions = [(-1,-1), (-1,0), (-1,1),

(0,-1), (0,1),

(1,-1), (1,0), (1,1)]

count = 0

for dy, dx in directions:

ny, nx = y + dy, x + dx

if 0 <= ny < rows and 0 <= nx < cols:

count += grid[ny][nx]

return count

# 更新一次格子狀態

def update():

global grid

new\_grid = [[0]\*cols for \_ in range(rows)]

for y in range(rows):

for x in range(cols):

neighbors = count\_neighbors(y, x)

if grid[y][x] == 1:

if neighbors in [2, 3]:

new\_grid[y][x] = 1 # 活著

else:

if neighbors == 3:

new\_grid[y][x] = 1 # 復活

grid = new\_grid

# 顯示格子狀態

def display():

for row in grid:

print(' '.join(['\*' if cell else '0' for cell in row]))

# 主執行流程

generation = 1

while True:

print(f"\n=== 第 {generation} 代 ===")

display()

user\_input = input("是否繼續下一代？(y/n): ").strip().lower()

if user\_input != 'y':

print("模擬結束。")

break

update()

generation += 1

#-------------------------------------------------------

#student1

class StudentNode:

def \_\_init\_\_(self, name, chinese, math, english):

self.name = name

self.chinese = chinese

self.math = math

self.english = english

self.left = None

self.right = None

def insert(root, node):

if root is None:

return node

if node.name < root.name:

root.left = insert(root.left, node)

else:

root.right = insert(root.right, node)

return root

def search(root, name):

if root is None or root.name == name:

return root

if name < root.name:

return search(root.left, name)

else:

return search(root.right, name)

def inorder(root):

if root is not None:

inorder(root.left)

print(f"姓名: {root.name}, 國文: {root.chinese}, 數學: {root.math}, 英文: {root.english}")

inorder(root.right)

def calc\_avg(root):

result = []

def dfs(node):

if node:

dfs(node.left)

result.append((node.chinese, node.math, node.english))

dfs(node.right)

dfs(root)

if not result:

print("沒有學生資料")

return

total = [sum(x) for x in zip(\*result)]

count = len(result)

print(f"全班國文平均: {total[0]/count:.1f}")

print(f"全班數學平均: {total[1]/count:.1f}")

print(f"全班英文平均: {total[2]/count:.1f}")

def get\_int\_input(prompt):

while True:

value = input(prompt)

try:

num = int(value)

if 0 <= num <= 100:

return num

else:

print("成績必須在 0 到 100 之間，請重新輸入！")

except ValueError:

print("請輸入正確的數字！")

def delete(root, name):

if root is None:

return root

if name < root.name:

root.left = delete(root.left, name)

elif name > root.name:

root.right = delete(root.right, name)

else:

# 找到要刪除的節點

if root.left is None:

return root.right

elif root.right is None:

return root.left

# 左右子樹都在，找右子樹最小者替代

min\_larger\_node = root.right

while min\_larger\_node.left:

min\_larger\_node = min\_larger\_node.left

# 複製資料

root.name = min\_larger\_node.name

root.chinese = min\_larger\_node.chinese

root.math = min\_larger\_node.math

root.english = min\_larger\_node.english

# 刪除右子樹的最小節點

root.right = delete(root.right, min\_larger\_node.name)

return root

root = None

while True:

print("\n1.輸入 2.修改 3.學生成績 4.班平均 5.刪除學生 6.離開")

choice = input("請輸入選項: ")

if choice == "1":

name = input("請輸入名字: ")

chinese = get\_int\_input("請輸入國文成績: ")

math = get\_int\_input("請輸入數學成績: ")

english = get\_int\_input("請輸入英文成績: ")

node = StudentNode(name, chinese, math, english)

root = insert(root, node)

elif choice == "2":

if root is None:

print("目前沒有學生資料，請先輸入。")

continue

name = input("請輸入要修改的學生名字: ")

student = search(root, name)

if student:

chinese = get\_int\_input("請輸入國文成績: ")

math = get\_int\_input("請輸入數學成績: ")

english = get\_int\_input("請輸入英文成績: ")

student.chinese = chinese

student.math = math

student.english = english

print("修改完成")

else:

print("查無此人")

elif choice == "3":

if root is None:

print("目前沒有學生資料，請先輸入。")

continue

inorder(root)

elif choice == "4":

if root is None:

print("目前沒有學生資料，請先輸入。")

continue

calc\_avg(root)

elif choice == "5":

if root is None:

print("目前沒有學生資料，請先輸入。")

continue

name = input("請輸入要刪除的學生名字: ")

if search(root, name):

root = delete(root, name)

print("已刪除", name)

else:

print("查無此人")

elif choice == "6":

print("再見！")

break

else:

print("無效選項，請重新輸入")

#-------------------------------------------------------

class StudentNode:

def \_\_init\_\_(self, student\_id, score):

self.id = student\_id

self.score = score

self.left = None

self.right = None

def insert(root, student\_id, score):

if root is None:

return StudentNode(student\_id, score)

if score < root.score:

root.left = insert(root.left, student\_id, score)

else:

root.right = insert(root.right, student\_id, score)

return root

def search(root, score):

if root is None:

return None

if score == root.score:

return root

elif score < root.score:

return search(root.left, score)

else:

return search(root.right, score)

def find\_min(node):

current = node

while current.left is not None:

current = current.left

return current

def delete(root, score):

if root is None:

return None

if score < root.score:

root.left = delete(root.left, score)

elif score > root.score:

root.right = delete(root.right, score)

else:

# Case 1: No child

if root.left is None and root.right is None:

return None

# Case 2: One child

elif root.left is None:

return root.right

elif root.right is None:

return root.left

# Case 3: Two children

else:

min\_larger\_node = find\_min(root.right)

root.score = min\_larger\_node.score

root.id = min\_larger\_node.id

root.right = delete(root.right, min\_larger\_node.score)

return root

def inorder\_traversal(root):

if root:

inorder\_traversal(root.left)

print(f"ID: {root.id}, Score: {root.score}")

inorder\_traversal(root.right)

# ----------------------------

# Example usage

# ----------------------------

if \_\_name\_\_ == "\_\_main\_\_":

# Create BST and insert students

root = None

root = insert(root, 1001, 85)

root = insert(root, 1002, 92)

root = insert(root, 1003, 76)

root = insert(root, 1004, 89)

print("Original BST (in-order):")

inorder\_traversal(root)

# Search example

target\_score = 89

result = search(root, target\_score)

if result:

print(f"\n✅ Found student with score {target\_score}: ID = {result.id}")

else:

print(f"\n❌ Score {target\_score} not found.")

# Delete example

print("\nDeleting score 85...")

root = delete(root, 85)

print("BST after deletion (in-order):")

inorder\_traversal(root)

#-------------------------------------------------------

# (1/1) × √1 + (1/2) × √2 + (1/3) × √3 + ... + (1/n) × √n

# 輸入使用者要計算的項數 n

n = int(input("請輸入要前幾項："))

# 初始化總和

total = 0

# 使用 for 迴圈計算每一項並加總

for i in range(1, n + 1):

term = (1 / i) \* (i \*\* 0.5) # (1/i) × √i

total += term

# 輸出總和，可選擇保留 4 位小數

print(f"總和為：{total:.4f}")

#-------------------------------------------------------

# -\*- coding: utf-8 -\*-

"""

Program Function:

1. Prompt the user to enter an infix expression (e.g., 3+4\*2/(1-5)\*\*2)

2. Calculate and display the result of the expression

"""

# Prompt the user to input an expression

expr = input("Please enter an infix expression (e.g., 3+4\*2): ")

try:

# Use Python's built-in eval function to compute the result

result = eval(expr)

# Display the full expression and the result

print(f"{expr} = {result}")

except Exception as e:

# Catch any input errors and display a warning

print("Calculation failed. Please make sure your input is a valid arithmetic expression.")

print("Error message:", e)

#-------------------------------------------------

# 定義節點類別，用於儲存每個文字欄位

class Node:

def \_\_init\_\_(self, text):

self.text = text # 欄位內容

self.next = None # 指向下一個節點

# 定義單向連結串列類別，實作文字欄位的操作

class LinkedList:

def \_\_init\_\_(self):

self.head = None # 串列起始節點

def append(self, text):

"""尾端新增一個欄位節點"""

new\_node = Node(text)

if not self.head:

# 如果目前是空串列，直接設為 head

self.head = new\_node

else:

# 否則跑到最後一個節點，再接上 new\_node

current = self.head

while current.next:

current = current.next

current.next = new\_node

def insert(self, index, text):

"""在指定位置插入一個欄位節點

index=0 表示插入到最前端，其餘位置依序往後推

"""

new\_node = Node(text)

# 插入到最前面

if index <= 0 or not self.head:

new\_node.next = self.head

self.head = new\_node

return

# 找到要插入前一個位置

current = self.head

prev = None

i = 0

while current and i < index:

prev = current

current = current.next

i += 1

# 連接 prev → new\_node → current

prev.next = new\_node

new\_node.next = current

def delete(self, index):

"""刪除指定位置的欄位節點

index=0 表示刪除最前端節點

"""

if not self.head:

return # 空串列無事

# 刪除最前端

if index <= 0:

self.head = self.head.next

return

# 找到欲刪除節點的前一個節點

current = self.head

prev = None

i = 0

while current.next and i < index:

prev = current

current = current.next

i += 1

# 跳過 current 節點

if prev and prev.next:

prev.next = current.next

def edit(self, index, new\_text):

"""編輯指定位置的欄位內容"""

current = self.head

i = 0

while current and i < index:

current = current.next

i += 1

if current:

current.text = new\_text

def display(self):

"""列出所有欄位內容及其索引"""

current = self.head

idx = 0

print("=== 文件內容 ===")

while current:

print(f"{idx}: {current.text}")

current = current.next

idx += 1

print("================\n")

if \_\_name\_\_ == "\_\_main\_\_":

# 建立文件物件

doc = LinkedList()

# 初始加入至少 5 筆筆記

doc.append("姓名: Aaron，地址: Kinmen, Taiwan")

doc.append("好友: Bob (遊戲名: OverwatchPlayer)")

doc.append("寵物: Lucky (狗)")

doc.append("興趣: Computer Science & AI")

doc.append("備註: 喜歡閱讀論文")

# 顯示初始內容

print("初始文件：")

doc.display()

# 範例：在第 2 筆之後插入新筆記

doc.insert(2, "好友: Carol (遊戲名: ZeldaFan)")

print("插入後（在索引 2 處插入）：")

doc.display()

# 範例：編輯第 0 筆筆記

doc.edit(0, "姓名: Aaron，地址: 金門大學")

print("編輯後（索引 0 修改）：")

doc.display()

# 範例：刪除第 3 筆筆記

doc.delete(3)

print("刪除後（移除索引 3 的筆記）：")

doc.display()

#------------------------------------------

import heapq

# 1. 定義病人資料：每筆為 (姓名, 優先級, 到院順序)

# 假設 A~I 依序到達，arr\_num 分別為 1~9

patients = [

('A', 3, 1),

('B', 2, 2),

('C', 1, 3),

('D', 3, 4),

('E', 4, 5),

('F', 2, 6),

('G', 2, 7),

('H', 3, 8),

('I', 1, 9),

]

# 2. 計算優先值：value = priority \* 1000 + 800 - arrival\_num

patients\_with\_value = []

for name, priority, arrival in patients:

value = priority \* 1000 + 800 - arrival

patients\_with\_value.append((name, priority, arrival, value))

# 2a. 列印每位病人的計算結果

print("病人 優先級 到院順序 計算值")

for name, priority, arrival, value in patients\_with\_value:

print(f"{name:>2} {priority:>2} {arrival:>2} {value}")

# 3. 根據計算值降序排序，決定治療順序

sorted\_patients = sorted(patients\_with\_value, key=lambda x: x[3], reverse=True)

print("\n── 根據優先值降序排序後的治療順序 ──")

for idx, (name, \_, \_, value) in enumerate(sorted\_patients, 1):

print(f"{idx:>2}. 病人{name} (值={value})")

# 4. 建立最大堆 (Max-Heap)

# Python 的 heapq 預設為最小堆，因此推入時使用 -value

heap = []

for name, \_, \_, value in patients\_with\_value:

heapq.heappush(heap, (-value, name))

# 4a. 列印堆的內部列表結構 (neg\_value, name)

print("\n堆的內部列表表示 (neg\_value, name)：")

print(heap)

# 4b. 從最大堆中依序彈出，得到真正的優先治療順序

print("\n── 從最大堆彈出得到的治療順序 ──")

order = []

while heap:

neg\_value, name = heapq.heappop(heap)

order.append((name, -neg\_value))

for idx, (name, value) in enumerate(order, 1):

print(f"{idx:>2}. 病人{name} (值={value})")

#---------------------------------------------

# -\*- coding: utf-8 -\*-

"""

Program Function:

1. Prompt the user to enter an infix expression (e.g., 3+4\*2/(1-5))

2. Display prefix, infix, postfix representations

3. Calculate and display the result

"""

# Function to define operator precedence

def precedence(op):

if op == '+' or op == '-':

return 1

if op == '\*' or op == '/':

return 2

return 0

# ========== 原本的：中序 (infix) → 後序 (postfix) 演算法 ==========

def infix\_to\_postfix(expression):

output = []

stack = []

i = 0

while i < len(expression):

char = expression[i]

if char == ' ':

i += 1

continue

# 如果是數字或小數點，處理多位數

if char.isdigit() or char == '.':

num = []

# 收集連續的 digit 或 '.'（支援浮點數）

while i < len(expression) and (expression[i].isdigit() or expression[i] == '.'):

num.append(expression[i])

i += 1

output.append(''.join(num))

continue

# 如果是 '('，直接推到 stack

elif char == '(':

stack.append(char)

# 如果是 ')'，就把 stack pop 出來直到遇到 '('

elif char == ')':

while stack and stack[-1] != '(':

output.append(stack.pop())

# 移除那個 '('

if stack:

stack.pop()

# 如果是四則運算子

elif char in '+-\*/':

# 如果 stack 裡頂端運算子優先度 >= 自己，就先 pop 出來丟到 output

while stack and stack[-1] != '(' and precedence(stack[-1]) >= precedence(char):

output.append(stack.pop())

stack.append(char)

i += 1

# 最後把 stack 裡剩下的運算子全部 pop 出來

while stack:

output.append(stack.pop())

return output

# ========== 新增：用後序演算法衍生前序 (prefix) 的函式 ==========

def infix\_to\_prefix(expression):

"""

利用「反轉後處理再反轉」技巧，把中序轉成前序：

1. 先把整串 expression 反轉（string reverse）

2. 反轉後，遇到 '(' → 換成 ')'，遇到 ')' → 換成 '('

3. 以反轉過後的字串去呼叫 infix\_to\_postfix()，得到一個暫時的後序 list

4. 最後把這個暫時的後序 list 逆序，變成 prefix list

"""

# 1. 去除空格（不影響邏輯，但方便後續處理）

expr = expression.replace(' ', '')

# 2. 把字串反轉

rev = expr[::-1]

# 3. 調換括號

tmp = []

for ch in rev:

if ch == '(':

tmp.append(')')

elif ch == ')':

tmp.append('(')

else:

tmp.append(ch)

rev = ''.join(tmp)

# 4. 對上述 rev 字串呼叫後序函式，得到一個 list（postfix of the reversed）

postfix\_of\_rev = infix\_to\_postfix(rev)

# 5. 把那個 list 反過來，就變成 prefix

prefix\_list = postfix\_of\_rev[::-1]

return prefix\_list

# ========== 原本的：計算後序 (postfix) ==========

def evaluate\_postfix(postfix\_tokens):

stack = []

for token in postfix\_tokens:

# 如果 token 裡面除了第一個小數點之外，全都是 digit → 視為數字

if token.replace('.', '', 1).isdigit():

stack.append(float(token))

else:

# 否則就是運算子，pop 兩個數做計算

b = stack.pop()

a = stack.pop()

if token == '+':

stack.append(a + b)

elif token == '-':

stack.append(a - b)

elif token == '\*':

stack.append(a \* b)

elif token == '/':

stack.append(a / b)

# 最後 stack 裡剩下的就是答案

return stack[0]

# ========== 主程式 ==========

if \_\_name\_\_ == '\_\_main\_\_':

expr = input("Enter infix expression (e.g., 2 + 3 \* 4): ").strip()

# 顯示「中序」── 就是使用者輸入的那條字串

print("Infix: ", expr)

# Step 1: 轉成 Prefix

prefix\_tokens = infix\_to\_prefix(expr)

# 以空格分隔 token 印出

print("Prefix: ", ' '.join(prefix\_tokens))

# Step 2: 轉成 Postfix

postfix\_tokens = infix\_to\_postfix(expr)

print("Postfix: ", ' '.join(postfix\_tokens))

# Step 3: 計算 Postfix（或你也可以寫 prefix evaluator，但這裡直接重用原來的後序計算）

try:

result = evaluate\_postfix(postfix\_tokens)

# 如果你想把整數顯示成整數格式，可以再檢查一下小數部分是否為 .0

if abs(result - int(result)) < 1e-9:

# 幾乎是整數，就以整數形式顯示

print("Result: ", int(result))

else:

# 否則就顯示浮點數

print("Result: ", result)

except Exception as e:

print("Evaluation error. Please check your expression. Error:", e)

#------------------------------------------------

from collections import deque

# 建立一個票號隊列，從 1 到 20

ticket\_queue = deque(range(1, 21))

# 定義呼叫下一位顧客的函式

def call\_next(queue, counter\_id):

"""

從隊列 pop 出下一張票，並指派到指定的櫃台

如果隊列已空，印出「目前無候號」

"""

if queue:

num = queue.popleft()

print(f"呼叫號碼 {num} → 櫃台 {counter\_id}")

else:

print(f"櫃台 {counter\_id}：目前無候號")

#── 模擬 Case 1 ──

# 狀況：只有櫃台 1 空閒（假設櫃台 2、3 都忙碌中）

print("Case 1：只有櫃台 1 空閒")

# 呼叫下一位到櫃台 1

call\_next(ticket\_queue, 1)

print("剩餘隊伍：", list(ticket\_queue), "\n")

#── 模擬 Case 2 ──

# 狀況：只有櫃台 2 空閒（假設櫃台 1、3 都忙碌中）

print("Case 2：只有櫃台 2 空閒")

# 呼叫下一位到櫃台 2

call\_next(ticket\_queue, 2)

print("剩餘隊伍：", list(ticket\_queue), "\n")

#── 模擬 Case 3 ──

# 狀況：櫃台 1 和 2 都空閒（假設櫃台 3 忙碌中）

# 要依照「櫃台編號由小到大」依序呼叫

print("Case 3：櫃台 1、2 都空閒，先呼叫編號較小的櫃台")

for counter in sorted([1, 2]):

call\_next(ticket\_queue, counter)

print("剩餘隊伍：", list(ticket\_queue), "\n")

#------------------------------------------------

from collections import deque

from datetime import datetime, timedelta

class Customer:

def \_\_init\_\_(self, arrival\_time: datetime, order\_details: str, amount: float):

"""

arrival\_time : datetime 顧客進店時間

order\_details : str 點餐內容

amount : float 消費金額

estimated\_wait : int|None 預估等待時間（分鐘），一開始為 None

"""

self.arrival\_time = arrival\_time

self.order\_details = order\_details

self.amount = amount

self.estimated\_wait = None # Will be set on arrival

def \_\_repr\_\_(self):

return (f"Customer(time={self.arrival\_time.strftime('%H:%M')}, "

f"order={self.order\_details}, "

f"amt={self.amount}, "

f"wait={self.estimated\_wait}m)")

class SnackShop:

def \_\_init\_\_(self, max\_per\_hour: int = 10):

"""

max\_per\_hour : int 每小時最大接待人數

current\_hour : datetime 目前所屬小時（整點）

hourly\_count : int 當前小時已到店並直接進店的人數

waiting\_queue : deque 排隊等候的顧客（FIFO）

current\_customers : list 正在店內消費的顧客

served\_stack : list 已服務完成顧客的堆疊（LIFO，可查最近服務紀錄）

"""

self.max\_per\_hour = max\_per\_hour

# 初始化 current\_hour 為當前整點

now = datetime.now()

self.current\_hour = now.replace(minute=0, second=0, microsecond=0)

self.hourly\_count = 0

self.waiting\_queue = deque()

self.current\_customers = []

self.served\_stack = []

def \_reset\_hour\_if\_needed(self):

"""如果跨到新小時或新日期，重置 hourly\_count 與 current\_hour"""

now = datetime.now()

# 檢查是否換小時或換日

if now.hour != self.current\_hour.hour or now.date() != self.current\_hour.date():

self.current\_hour = now.replace(minute=0, second=0, microsecond=0)

self.hourly\_count = 0

def arrive\_customer(self, customer: Customer):

"""

新顧客到達：

- 若本小時接待量未滿，直接加入 current\_customers，estimated\_wait = 0

- 否則加入 waiting\_queue，並依排隊順位估算等待時間

"""

self.\_reset\_hour\_if\_needed()

if self.hourly\_count < self.max\_per\_hour:

# 直接進店

customer.estimated\_wait = 0

self.current\_customers.append(customer)

self.hourly\_count += 1

else:

# 加入排隊

position = len(self.waiting\_queue)

# 假設每小時可服務 max\_per\_hour 人，估算等待小時數

hours\_wait = position // self.max\_per\_hour + 1

customer.estimated\_wait = hours\_wait \* 60 # 轉為分鐘

self.waiting\_queue.append(customer)

def process\_next(self):

"""

處理下一位完成消費的顧客：

- 從 current\_customers 取出最早進店（FIFO），推入 served\_stack

- 再從 waiting\_queue 補位，如有則設 estimated\_wait = 0

回傳完成服務的 Customer 物件

"""

if not self.current\_customers:

return None

# 完成服務：取出最先加入的顧客

finished = self.current\_customers.pop(0)

self.served\_stack.append(finished)

# 若有排隊顧客，補位進店

if self.waiting\_queue:

next\_cust = self.waiting\_queue.popleft()

next\_cust.estimated\_wait = 0

self.current\_customers.append(next\_cust)

return finished

def get\_waiting\_list(self):

"""回傳目前所有在排隊的顧客清單（複製後回傳）"""

return list(self.waiting\_queue)

def get\_current\_customers(self):

"""回傳目前正在店內的顧客清單（複製後回傳）"""

return list(self.current\_customers)

def total\_sales(self) -> float:

"""計算今日已完成服務之顧客總營業額"""

return sum(c.amount for c in self.served\_stack)

def average\_spending(self) -> float:

"""計算今日每位已服務顧客平均消費金額"""

count = len(self.served\_stack)

return (self.total\_sales() / count) if count else 0.0

# 範例使用

if \_\_name\_\_ == "\_\_main\_\_":

shop = SnackShop(max\_per\_hour=10)

# 模擬 12 位顧客陸續到店

for i in range(12):

cust = Customer(

arrival\_time=datetime.now(),

order\_details=f"Snack #{i+1}",

amount=50 + i # 金額 50,51,...

)

shop.arrive\_customer(cust)

print("Current in-shop customers:")

print(shop.get\_current\_customers())

print("\nWaiting queue:")

print(shop.get\_waiting\_list())

# 處理掉 3 位消費完顧客

for \_ in range(3):

done = shop.process\_next()

print("\nServed:", done)

print("\nToday's total sales:", shop.total\_sales())

print("Average spending per customer:", shop.average\_spending())

#------------------------------------------------------

class AVLNode:

def \_\_init\_\_(self, key, weight):

# key: 道路名稱或識別碼

# weight: 該道路目前流量（越大表示越壅塞）

self.key = key

self.weight = weight

self.height = 1

self.left = None

self.right = None

class AVLTree:

def \_\_init\_\_(self, bf\_threshold=1):

# bf\_threshold: 當節點平衡因子絕對值超過此值時，視為該路段壅塞

self.root = None

self.bf\_threshold = bf\_threshold

# ---------- 公用方法 ----------

def get\_height(self, node):

if not node:

return 0

return node.height

def get\_balance(self, node):

# 平衡因子 = 左子樹高度 - 右子樹高度

if not node:

return 0

return self.get\_height(node.left) - self.get\_height(node.right)

# ---------- 旋轉方法 ----------

def right\_rotate(self, y):

x = y.left

T2 = x.right

# 執行旋轉

x.right = y

y.left = T2

# 更新高度

y.height = max(self.get\_height(y.left), self.get\_height(y.right)) + 1

x.height = max(self.get\_height(x.left), self.get\_height(x.right)) + 1

print(f"執行右旋 (RR / LL) 節點 {y.key} → {x.key}")

return x # 新根

def left\_rotate(self, x):

y = x.right

T2 = y.left

# 執行旋轉

y.left = x

x.right = T2

# 更新高度

x.height = max(self.get\_height(x.left), self.get\_height(x.right)) + 1

y.height = max(self.get\_height(y.left), self.get\_height(y.right)) + 1

print(f"執行左旋 (LL / RR) 節點 {x.key} → {y.key}")

return y # 新根

# ---------- 插入節點 ----------

def insert(self, node, key, weight):

# 1. 標準 BST 插入

if not node:

return AVLNode(key, weight)

if weight < node.weight:

node.left = self.insert(node.left, key, weight)

else:

node.right = self.insert(node.right, key, weight)

# 2. 更新高度

node.height = max(self.get\_height(node.left), self.get\_height(node.right)) + 1

# 3. 計算並檢查平衡因子

balance = self.get\_balance(node)

# 4. 處理四種失衡情境

# LL

if balance > 1 and weight < node.left.weight:

return self.right\_rotate(node)

# RR

if balance < -1 and weight >= node.right.weight:

return self.left\_rotate(node)

# LR

if balance > 1 and weight >= node.left.weight:

node.left = self.left\_rotate(node.left)

print(f"先左旋再右旋 (LR) 在節點 {node.key}")

return self.right\_rotate(node)

# RL

if balance < -1 and weight < node.right.weight:

node.right = self.right\_rotate(node.right)

print(f"先右旋再左旋 (RL) 在節點 {node.key}")

return self.left\_rotate(node)

# 5. 若節點壅塞 (平衡因子超過門檻)，呼叫推薦替代路線

if abs(balance) > self.bf\_threshold:

print(f"警告：道路 {node.key} 壅塞 (BF={balance})，推薦替代路線：")

self.recommend\_alternative(node)

return node

# ---------- 刪除節點 ----------

def delete(self, node, weight):

if not node:

return node

if weight < node.weight:

node.left = self.delete(node.left, weight)

elif weight > node.weight:

node.right = self.delete(node.right, weight)

else:

# 找到要刪除的節點

if not node.left:

return node.right

elif not node.right:

return node.left

# 兩子樹皆有：取右子樹最小值頂替

temp = self.get\_min\_value\_node(node.right)

node.key, node.weight = temp.key, temp.weight

node.right = self.delete(node.right, temp.weight)

# 更新高度

node.height = max(self.get\_height(node.left), self.get\_height(node.right)) + 1

balance = self.get\_balance(node)

# 同樣處理四種失衡並平衡

if balance > 1 and self.get\_balance(node.left) >= 0:

return self.right\_rotate(node)

if balance > 1 and self.get\_balance(node.left) < 0:

node.left = self.left\_rotate(node.left)

return self.right\_rotate(node)

if balance < -1 and self.get\_balance(node.right) <= 0:

return self.left\_rotate(node)

if balance < -1 and self.get\_balance(node.right) > 0:

node.right = self.right\_rotate(node.right)

return self.left\_rotate(node)

return node

def get\_min\_value\_node(self, node):

current = node

while current.left:

current = current.left

return current

# ---------- 替代路線推薦 ----------

def recommend\_alternative(self, congested\_node):

# 遍歷整棵樹，找出流量低於 congested\_node.weight 的節點

alternatives = []

def dfs(n):

if not n:

return

if n.weight < congested\_node.weight:

alternatives.append((n.key, n.weight))

dfs(n.left)

dfs(n.right)

dfs(self.root)

# 根據流量由低到高排序，取前 5 筆

alternatives.sort(key=lambda x: x[1])

for road, w in alternatives[:5]:

print(f" • 道路 {road} (流量={w})")

# ---------- 外部介面 ----------

def add\_road(self, key, weight):

"""新增道路 (插入節點)"""

self.root = self.insert(self.root, key, weight)

def remove\_road(self, weight):

"""關閉道路 (刪除節點)，以流量當識別"""

self.root = self.delete(self.root, weight)

def inorder(self, node):

"""中序列印 (可用於偵錯)"""

if not node:

return

self.inorder(node.left)

print(f"{node.key}:{node.weight}", end=" ")

self.inorder(node.right)

# ------------------- 使用範例 -------------------

if \_\_name\_\_ == "\_\_main\_\_":

tree = AVLTree(bf\_threshold=1)

# 模擬道路流量動態

road\_flows = [

("A路", 50), ("B路", 30), ("C路", 70),

("D路", 20), ("E路", 40), ("F路", 60), ("G路", 80)

]

for name, flow in road\_flows:

print(f"\n新增道路 {name} (流量={flow})")

tree.add\_road(name, flow)

print("現況 (中序):", end=" ")

tree.inorder(tree.root)

print("\n" + "-"\*40)

#-------------------------------------------------

class MaxHeap:

def \_\_init\_\_(self, initial\_data=None):

"""Initialize the MaxHeap. If initial\_data is provided, build the heap."""

# The heap list stores elements as [candidate\_name, support\_value]

self.heap = []

# A dictionary to map candidate names to their index in the heap list

self.position = {}

if initial\_data:

self.build\_heap(initial\_data)

def build\_heap(self, data\_list):

"""Build the heap from a list of (candidate\_name, support\_value) tuples."""

self.heap = []

self.position = {}

# Insert all elements into heap list and record their positions

for name, support in data\_list:

self.heap.append([name, support])

self.position[name] = len(self.heap) - 1

# Heapify process: call \_heapify\_down on all non-leaf nodes

n = len(self.heap)

for index in reversed(range(n // 2)):

self.\_heapify\_down(index)

def insert\_or\_update(self, name, support):

"""Insert a new candidate or update the support of an existing one."""

if name in self.position:

# Update existing candidate's support and adjust heap

index = self.position[name]

old\_support = self.heap[index][1]

self.heap[index][1] = support

# Decide whether to sift up or down based on new support

if support > old\_support:

self.\_heapify\_up(index)

else:

self.\_heapify\_down(index)

else:

# Insert new candidate

self.heap.append([name, support])

index = len(self.heap) - 1

self.position[name] = index

self.\_heapify\_up(index)

def get\_top(self):

"""Return the candidate with the highest support without removing it."""

if not self.heap:

return None

name, support = self.heap[0]

return name, support

def pop\_top(self):

"""Remove and return the candidate with the highest support."""

if not self.heap:

return None

top\_node = self.heap[0]

last\_node = self.heap.pop()

del self.position[top\_node[0]]

if self.heap:

# Move last node to root and heapify down

self.heap[0] = last\_node

self.position[last\_node[0]] = 0

self.\_heapify\_down(0)

return top\_node

def display\_heap(self):

"""Return the current heap array (level-order representation)."""

# Return a list of tuples for readability

return [(name, support) for name, support in self.heap]

def get\_sorted\_candidates(self):

"""Return a list of all candidates sorted by support (highest first) without modifying the original heap."""

# Copy heap elements locally and sort by support descending

return sorted(self.heap, key=lambda x: x[1], reverse=True)

def \_heapify\_up(self, index):

"""Maintain heap property by moving the node at index up as needed."""

while index > 0:

parent\_index = (index - 1) // 2

# Compare support values: if child > parent, swap

if self.heap[index][1] > self.heap[parent\_index][1]:

self.\_swap(index, parent\_index)

index = parent\_index

else:

break

def \_heapify\_down(self, index):

"""Maintain heap property by moving the node at index down as needed."""

n = len(self.heap)

while True:

left = 2 \* index + 1

right = 2 \* index + 2

largest = index

# Check if left child exists and is greater than current largest

if left < n and self.heap[left][1] > self.heap[largest][1]:

largest = left

# Check if right child exists and is greater than current largest

if right < n and self.heap[right][1] > self.heap[largest][1]:

largest = right

if largest != index:

self.\_swap(index, largest)

index = largest

else:

break

def \_swap(self, i, j):

"""Swap two nodes in the heap and update their positions."""

self.position[self.heap[i][0]], self.position[self.heap[j][0]] = j, i

self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

if \_\_name\_\_ == "\_\_main\_\_":

# Example usage: simulate a five-person primary election

# Initial candidates with their support values

initial\_candidates = [

("Alice", 250),

("Bob", 300),

("Charlie", 150),

("Diana", 280),

("Ethan", 200)

]

# Create a MaxHeap with initial data

heap = MaxHeap(initial\_candidates)

# Display the heap structure after building

print("Initial heap (level-order):", heap.display\_heap())

# Show current ranking of candidates by support

print("Current ranking:", heap.get\_sorted\_candidates())

# Insert a new candidate

heap.insert\_or\_update("Fiona", 270)

print("After inserting Fiona:", heap.display\_heap())

# Update an existing candidate's support

heap.insert\_or\_update("Charlie", 320)

print("After updating Charlie's support to 320:", heap.display\_heap())

# Get the candidate most likely to win

top\_candidate = heap.get\_top()

print("Current top candidate:", top\_candidate)

# Simulate support fluctuation: Bob's support drops

heap.insert\_or\_update("Bob", 180)

print("After Bob's support drops to 180:", heap.display\_heap())

# Pop the top candidate (e.g., declare winner)

winner = heap.pop\_top()

print("Winner:", winner)

print("Heap after popping top:", heap.display\_heap())

#---------------------------------------------------------

# Node class for BST

class Node:

def \_\_init\_\_(self, name, score):

self.name = name

self.score = score

self.left = None

self.right = None

# Function to insert a new node into BST

def insert(root, name, score):

"""

Insert a node with given name and score into BST.

The BST is ordered by score.

"""

if root is None:

return Node(name, score)

if score < root.score:

root.left = insert(root.left, name, score)

else:

root.right = insert(root.right, name, score)

return root

# Function to find the node with minimum score in BST (used in deletion)

def find\_min(node):

"""

Find the node with minimum score in BST.

"""

current = node

while current.left is not None:

current = current.left

return current

# Function to delete a node by score

def delete\_node(root, score):

"""

Delete the node with given score from BST.

"""

if root is None:

return None

if score < root.score:

root.left = delete\_node(root.left, score)

elif score > root.score:

root.right = delete\_node(root.right, score)

else:

# Node to be deleted found

# Case 1: No child or one child

if root.left is None:

return root.right

elif root.right is None:

return root.left

# Case 2: Two children: get inorder successor (smallest in right subtree)

temp = find\_min(root.right)

# Copy the inorder successor's content to this node

root.name = temp.name

root.score = temp.score

# Delete the inorder successor

root.right = delete\_node(root.right, temp.score)

return root

# Function to print BST structure in text (sideways)

def print\_tree(node, level=0):

"""

Print the BST structure in a sideways textual format.

Right subtree is printed above and left subtree below with indentation.

"""

if node is not None:

# Print right subtree

print\_tree(node.right, level + 1)

# Print current node with indentation

print(' ' \* level + f'({node.name}, {node.score})')

# Print left subtree

print\_tree(node.left, level + 1)

# Main code

if \_\_name\_\_ == "\_\_main\_\_":

# Insert 7 student records (name, score)

student\_data = [

("Alice", 85),

("Bob", 92),

("Charlie", 78),

("Diana", 90),

("Eve", 88),

("Frank", 95),

("Grace", 80)

]

root = None

# Build the BST with student data

for name, score in student\_data:

root = insert(root, name, score)

# Insert an extra data with your own seat number

# Replace the value of my\_seat\_number with your actual seat number

my\_seat\_number = 15

root = insert(root, "MySeat", my\_seat\_number)

# Print the original BST structure

print("Original BST:")

print\_tree(root)

# Delete a node (e.g., delete the node with score 92)

root = delete\_node(root, 92)

# Print the BST structure after deletion

print("\nBST after deleting node with score 92:")

print\_tree(root)

#------------------------------------------

以下是加入**中文註解**的完整 Python 程式碼，模擬政黨初選使用 **最大堆（Max Heap）** 結構來預測誰最有可能勝出：

class MaxHeap:

def \_\_init\_\_(self, initial\_data=None):

"""初始化最大堆，如果有初始資料就建立堆"""

self.heap = [] # 儲存堆的列表，元素為 [候選人名字, 支持度]

self.position = {} # 用來記錄每個候選人在堆中的索引位置（方便更新）

if initial\_data:

self.build\_heap(initial\_data)

def build\_heap(self, data\_list):

"""從初始候選名單建立堆結構"""

self.heap = []

self.position = {}

for name, support in data\_list:

self.heap.append([name, support])

self.position[name] = len(self.heap) - 1

# 對每個非葉節點從下往上做 heapify\_down 以建立正確的最大堆

for index in reversed(range(len(self.heap) // 2)):

self.\_heapify\_down(index)

def insert\_or\_update(self, name, support):

"""插入新候選人，或更新已有候選人的支持度"""

if name in self.position:

# 更新支持度後根據新值決定是往上調整還是往下調整

index = self.position[name]

old\_support = self.heap[index][1]

self.heap[index][1] = support

if support > old\_support:

self.\_heapify\_up(index)

else:

self.\_heapify\_down(index)

else:

# 新增候選人至堆尾，再往上調整以維持堆性質

self.heap.append([name, support])

index = len(self.heap) - 1

self.position[name] = index

self.\_heapify\_up(index)

def get\_top(self):

"""取得目前支持度最高的候選人（不移除）"""

if not self.heap:

return None

name, support = self.heap[0]

return name, support

def pop\_top(self):

"""移除並回傳支持度最高的候選人"""

if not self.heap:

return None

top\_node = self.heap[0]

last\_node = self.heap.pop()

del self.position[top\_node[0]]

if self.heap:

self.heap[0] = last\_node

self.position[last\_node[0]] = 0

self.\_heapify\_down(0)

return top\_node

def display\_heap(self):

"""回傳目前堆積的內容（用 level-order 顯示）"""

return [(name, support) for name, support in self.heap]

def get\_sorted\_candidates(self):

"""取得所有候選人按支持度排序（不改變原堆）"""

return sorted(self.heap, key=lambda x: x[1], reverse=True)

def \_heapify\_up(self, index):

"""從 index 向上調整以維持最大堆性質"""

while index > 0:

parent\_index = (index - 1) // 2

if self.heap[index][1] > self.heap[parent\_index][1]:

self.\_swap(index, parent\_index)

index = parent\_index

else:

break

def \_heapify\_down(self, index):

"""從 index 向下調整以維持最大堆性質"""

n = len(self.heap)

while True:

left = 2 \* index + 1

right = 2 \* index + 2

largest = index

if left < n and self.heap[left][1] > self.heap[largest][1]:

largest = left

if right < n and self.heap[right][1] > self.heap[largest][1]:

largest = right

if largest != index:

self.\_swap(index, largest)

index = largest

else:

break

def \_swap(self, i, j):

"""交換堆中兩個節點，並更新 position 字典"""

self.position[self.heap[i][0]], self.position[self.heap[j][0]] = j, i

self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

if \_\_name\_\_ == "\_\_main\_\_":

# 初始候選人名單與支持度

initial\_candidates = [

("Alice", 250),

("Bob", 300),

("Charlie", 150),

("Diana", 280),

("Ethan", 200)

]

# 建立最大堆

heap = MaxHeap(initial\_candidates)

print("初始堆結構:", heap.display\_heap())

print("目前支持度排序:", heap.get\_sorted\_candidates())

heap.insert\_or\_update("Fiona", 270)

print("插入 Fiona 後:", heap.display\_heap())

heap.insert\_or\_update("Charlie", 320)

print("更新 Charlie 支持度至 320 後:", heap.display\_heap())

top\_candidate = heap.get\_top()

print("目前最可能勝選者:", top\_candidate)

heap.insert\_or\_update("Bob", 180)

print("Bob 支持度降至 180 後:", heap.display\_heap())

winner = heap.pop\_top()

print("勝出者:", winner)

print("移除勝出者後的堆:", heap.display\_heap())

**📌 重點函式心得總結：**

| **函式名稱** | **功能** | **心得重點** |
| --- | --- | --- |
| insert\_or\_update() | 插入新候選人或更新支持度 | ❗ 支持度改變後需判斷方向（up/down），避免錯誤排序。 |
| \_heapify\_up() | 向上調整堆結構 | 適用於新加入或支持度上升情境，確保父節點比子節點大。 |
| \_heapify\_down() | 向下調整堆結構 | 常用於刪除頂端或支持度下降，維持最大堆性質。 |
| get\_top() | 取得目前支持度最高者 | 📊 可作為民調預測「即時第一名」的指標。 |
| pop\_top() | 移除並回傳第一名 | 🗳️ 模擬「勝出者已確定、退出初選」的情境。 |
| display\_heap() | 回傳目前堆結構（level order） | 👀 有助於視覺化堆結構變化與 debug。 |
| get\_sorted\_candidates() | 輸出全部候選人依支持度排序 | 📈 可模擬完整初選排行榜，用於分析趨勢。 |

以下針對專案中各題（各模組）的主要函式，提出關鍵功能說明與心得反思：

## 1. Huffman 編碼與壓縮

* **build\_huffman\_tree(freq\_map)**
  + **功能**：從字元頻率表建立最小佇列，依序合併頻率最小的兩個節點，直到剩下一棵 Huffman 樹的根節點。
  + **心得**：利用 heapq 處理最小堆極為直覺，整體流程清晰易懂。注意到 HuffmanNode 的 \_\_lt\_\_ 定義，使節點可直接比較。若輸入字符集過大時，可考慮改用更高效的優先佇列實作（如 Fibonacci Heap），不過對一般文字壓縮而言已足夠。
* **generate\_codes(node, prefix=””, huffman\_codes=None)**
  + **功能**：遞迴遍歷 Huffman 樹，把左子節點路徑加 “0”、右子節點路徑加 “1”，收集所有葉節點對應的編碼。
  + **心得**：遞迴實作簡潔，但注意如果樹非常深（極端情況下），可能導致遞迴深度過大。可以考慮改為顯式堆疊的 DFS，但以一般文字分佈而言，樹深度不會太誇張，遞迴即可接受。
* **compress(input\_text, huffman\_codes)**
  + **功能**：將輸入文字中每個字元直接對應到編碼串，拼接成最終二進制字串（以字串形式儲存）。
  + **心得**：實作簡單，核心在於查表速度。若輸入包含未在 huffman\_codes 的字元，需額外處理異常情況。目前假定所有字元皆已預先計算頻率，是典型做法。
* **decompress(encoded\_binary, root)**
  + **功能**：從根節點出發，逐位讀取 “0/1” 選擇左或右子節點，直到到達葉節點，還原字元並回到根繼續解碼。
  + **心得**：典型的 Huffman 解碼流程，關鍵在能正確重建原樹並無二義性。若遇到資料損毀（例如 bit 欄位遺失），需額外驗錯；此版本則專注於「正確輸入下」的解碼。

## 2. Conway’s Game of Life（逐步版本）

* **count\_neighbors(y, x)**
  + **功能**：計算指定格子（y, x）周圍八個相鄰格子的活細胞數量。
  + **心得**：八向迭代邏輯清晰，直接讀 global grid。若要擴充到更大維度或環形邊界，可將邊界判斷邏輯抽成參數或改寫為模組化函式。
* **update()**
  + **功能**：根據 Game of Life 的規則（存活格子有 2 或 3 個活鄰居則繼續存活，死格子若恰好有 3 個活鄰居則復活），建構新一代 new\_grid，最後更新 grid。
  + **心得**：採用兩層巢狀迴圈（列×行）遍歷，每格都重新計算。對 10×20 的尺寸效能無虞，但若欲擴充到大規模網格，可考慮只更新活動範圍（有改變可能的格子），或使用稀疏矩陣表示法減少計算量。
* **display()**
  + **功能**：以文字形式在終端顯示當前 grid，活細胞顯示 \*，死細胞顯示 0。
  + **心得**：簡潔易讀，若未來想加上視覺效果，可以改用 curses 庫或直接用圖形化套件（如 Pygame、matplotlib）。
* **主執行迴圈 (while True 部分)**
  + **功能**：負責印出當前世代、呼叫 display()，提示使用者是否繼續到下一代，若否則結束。
  + **心得**：這種互動式控制方便用戶步進觀察演化過程，但若想「自動演算多代」則可將 prompt 搜除，或加上延遲（time.sleep()）與清螢幕功能。若加上無限滾動或動畫化，使用者體驗更佳。

## 3. 學生成績管理系統（BST 以姓名排序）

* **class StudentNode**
  + **功能**：節點屬性包含 name、各科成績、left/right 子節點指標。
  + **心得**：以姓名作為鍵值（字典序）實作二元搜尋樹 (BST)，若名字相同的情況尚未處理（目前直接插到右子樹）；可視需求加上相同 key 的合併或拒絕機制。
* **insert(root, node)**
  + **功能**：標準 BST 插入，根據 node.name 比較，往左或右插入。
  + **心得**：若未來需要平衡樹或大量查詢，可改用平衡樹（如 AVL、紅黑樹）或換成 hash table。當只有少量學生資料時，這樣的實作夠用且邏輯簡單。
* **search(root, name)**
  + **功能**：遞迴搜尋節點，透過姓名比較逐層往下。
  + **心得**：回傳節點物件後可直接修改其成績。若 BST 不平衡，最壞情況成為 O(n)。適合在學生數少或姓名分布均勻時使用。
* **inorder(root)**
  + **功能**：中序走訪，依姓名字典序列印所有學生與成績。
  + **心得**：中序遍歷 BST 即自動排序；如果要匯出結果給視覺化介面，可將 print 換成把資料收到 list 中再做後續處理。
* **calc\_avg(root)**
  + **功能**：透過深度優先搜尋 (DFS) 收集所有節點的成績，計算全班國文、數學、英文平均。
  + **心得**：先用兩層 DFS（走訪左→加入資料→走訪右），取得 result list，再一次做統計。可以簡化為單次走訪時計算累加值及計數，減少記憶體使用。
* **get\_int\_input(prompt)**
  + **功能**：反覆提示輸入，直到使用者輸入合法的 0～100 整數成績為止。
  + **心得**：在互動程式中常見的輸入檢驗邏輯，格式簡潔；未來若想統一錯誤處理，可改以 exception 或自定義驗證函式。
* **delete(root, name)**
  + **功能**：在 BST 中找到關鍵字為 name 的節點並刪除，處理三種情況：無子、單子、雙子，用右子樹最小值替代。
  + **心得**：典型 BST 刪除流程，如果學生姓名重複，可能只刪除找到的第一筆。可視需求改進為以 (姓名＋其他唯一 ID) 組成複合鍵，避免歧義。

## 4. 以分數為 Key 的學生 BST 實作

* **class StudentNode (重複定義，但此處 id 與 score)**
  + **功能**：節點屬性包含 id（學號）、score（分數），以及 left/right。
  + **心得**：與前一題不同，用分數做 BST 的排序依據，若多個學生同分，程式將插到右子樹；若需處理相同分數多筆，可改用額外結構（如 list）或在 score 上加細節（e.g., (score, id)）。
* **insert(root, student\_id, score)**
  + **功能**：依 score 大小插入 BST，維持所有節點依分數排序。
  + **心得**：以分數排序的樹方便按照成績做查詢（e.g., 印出前 N 名）。若人数太多、且分數集中在少數值時，BST 可能退化，若需要穩定高效查詢，可考慮改為平衡樹。
* **search(root, score)**
  + **功能**：依分數值在 BST 中搜尋對應節點，回傳節點或 None。
  + **心得**：只能透過分數找到某位學生，但若有多位同分，只回傳第一個碰到的節點。實際使用中或需複數比對，可改成返回包含所有同分學生的 list。
* **find\_min(node)**
  + **功能**：回傳以 node 為根的子樹中最小分數的節點（最左葉）。
  + **心得**：用於刪除時取右子樹最小值替代，流程標準。
* **delete(root, score)**
  + **功能**：刪除 BST 中 score 相符的節點，處理無子、單子、雙子三種情況。
  + **心得**：整體與前一版 BST 刪除雷同，僅 key 值不同。同樣，如果有多筆相同 score，只刪除第一筆。若需刪除全部相同分數，可改寫遞迴邏輯。
* **inorder\_traversal(root)**
  + **功能**：中序遍歷，依分數從小到大印出 ID 與分數。
  + **心得**：可快速列出全班分數排序，若要反向（從大到小）列印，只需先走訪右子樹。
* **主程式範例使用**
  + **功能**：示範插入四筆學生資料、搜尋指定分數、刪除某分數並印出結果。
  + **心得**：範例清楚展示 BST 操作流程，適合作為學習或測試用途。若未來需要更完整的互動介面，可將 I/O 與邏輯分離。

## 5. 無限級數加總：∑ (1/i) × √i

* **主要邏輯（無函式封裝）**
  + 使用 for i in range(1, n+1) 迴圈，計算每項 (1 / i) \* (i \*\* 0.5)，累加到 total，最後印出結果。
  + **心得**：直觀且易讀。如果希望重複呼叫，可以將這部分改為 def series\_sum(n): 函式，回傳總和，方便後續程式整合或單元測試。同時，可考慮在 n 很大時累加誤差問題，用更高精度的 decimal 模組。

## 6. 簡易中序表示式計算器

* **主要邏輯（無函式封裝）**
  + 讀取使用者輸入的中序算術字串 expr，直接用 Python eval(expr) 計算並顯示結果；若有例外，印出錯誤訊息。
  + **心得**：用 eval 實作最簡潔，但存在安全風險——若使用者輸入惡意程式碼（如 \_\_import\_\_('os').system('rm -rf /')），將直接執行。實際應用須做輸入過濾或改用安全的解析與計算方式（例如前述的 postfix evaluator 模組）。若僅作學習用途，此實作無疑是最快速的方法。

## 7. 文字處理系統（單向鏈結串列）

* **class Node**
  + **功能**：代表單個文字欄位節點，儲存 text 內容及 next 指標。
  + **心得**：結構單純，而 text 以字串方式存欄位資料，適合做文件編輯示範。
* **class LinkedList (方法：append, insert, delete, edit, display)**
  + **append(text)**
    - **功能**：在串列尾端加入新節點。
    - **心得**：如果使用頻繁，可加上 tail 指標，避免每次迴圈到最後才插入，將時間複雜度由 O(n) 降為 O(1)。
  + **insert(index, text)**
    - **功能**：在指定索引位置插入新節點，若 index <= 0 或串列為空，則插到最前端。
    - **心得**：邏輯簡潔，但如果 index 超過串列長度，會插到末端。可根據需求明確定義行為：要麼超出拋錯，要麼直接 append。
  + **delete(index)**
    - **功能**：刪除指定索引位置的節點，若 index <= 0 則刪首。
    - **心得**：注意如果 index 大於最後一個有效索引，程式跳過，不做刪除。若希望拋出錯誤或通知用戶，可加上邊界檢查提示。
  + **edit(index, new\_text)**
    - **功能**：找到第 index 個節點並修改其 text。
    - **心得**：若 index 超過範圍，程式靜默不修改且不提示。可改進為回傳布林值表示是否修改成功，或拋例外告知「索引無效」。
  + **display()**
    - **功能**：從頭節點依序印出每個節點索引與內容，便於檢視整份文件。
    - **心得**：對於測試與示範非常直觀；若未來想支援更複雜的操作（如搜尋、儲存檔案），可改為收集內容到 list 再做格式化輸出。

## 8. 醫院病人優先處理（Heap 排序）

* **無明確函式定義，但關鍵邏輯可視為以下步驟**
  1. **計算優先值 (value = priority \* 1000 + 800 - arrival\_num)**
     + **心得**：將 priority 欄位放大 1000 倍，再加上固定偏移並扣除到院順序，讓高優先值總是在排序中最前。設計巧妙地把「緊急度」與「到院早晚」結合為單一數值。
  2. **sorted\_patients = sorted(..., key=lambda x: x[3], reverse=True)**
     + **心得**：直接用 Python 內建排序，也可得優先治療順序。簡單易懂，但若要動態新增或刪除病人，或隨時查詢下一位，需要改用堆結構才更有效率。
  3. **建立 Max-Heap：heapq.heappush(heap, (-value, name))**
     + **心得**：Python 的 heapq 是最小堆，故用 -value 來模擬最大堆。此做法很常見，也能在 O(log n) 時間內取得下一位要治療的病人。
  4. **while heap: heapq.heappop()**
     + **心得**：依序彈出最高優先值的病人，列印治療順序。結構邏輯顯示掌握 Python 堆的基本使用模式。

## 9. 中序→前序/後序轉換與計算

* **precedence(op)**
  + **功能**：返回運算子優先順序（+/- 為 1，\*// 為 2，其他為 0）。
  + **心得**：清楚定義算術優先級，用於後續中序轉後序的比較判斷；如要擴充到指數運算、括號優先級或更多運算符，可補充邏輯。
* **infix\_to\_postfix(expression)**
  + **功能**：將中序運算式轉為後序 (postfix) 表示法，支援多位數與浮點數。
  + **心得**：採「運算子堆疊 + 動態讀取數字」方式，對於浮點數支援完備（遇小數點繼續讀取），遞迴深度不高、清晰易懂。若要擴充支援更多運算符或函式，可改寫成 Tokenizer + Shunting-yard 演算法更通用。
* **infix\_to\_prefix(expression)**
  + **功能**：先將中序字串反轉、交換括號，再呼叫 infix\_to\_postfix，最後把得到的後序令牌逆序取得前序。
  + **心得**：「反轉→後序→再反轉」是經典技巧，優化開發成本，不必重寫完整的中序→前序演算法。注意在反轉過程中需正確交換左右括號，程式內有正確處理。
* **evaluate\_postfix(postfix\_tokens)**
  + **功能**：以堆疊計算後序運算式結果，將數字推入堆疊、遇運算子就 pop 兩個進行計算並推回，最後堆疊剩結果。
  + **心得**：邏輯簡潔且具一般性，只要 postfix\_tokens 內容合法，即可正確計算；若要增加指數 (^) 或函式 (sin, cos)，只需在運算子判斷段加上對應邏輯即可。
* **主程式 (if \_\_name\_\_ == '\_\_main\_\_')**
  + **功能**：提示使用者輸入中序字串，依序印出中序、前序、後序表示，並利用 evaluate\_postfix 計算結果。
  + **心得**：透過函式串接呈現完整工作流程，操作體驗連貫；若未來想把輸出封裝成 GUI 或 Web 介面，只要把 print 換成回傳或回呼即可。

## 10. 銀行號碼叫號系統（Queue 模擬）

* **call\_next(queue, counter\_id)**
  + **功能**：從 deque 隊列彈出下一號碼，並印出「號碼 → 柜台 ID」。若隊列為空，印「目前無候號」。
  + **心得**：簡潔地反映「哪個櫃台空閒就呼叫下一位」的邏輯。若要統一管理多個櫃台狀態，或讓系統自動在多閒時同時呼叫，可以將 call\_next 包成回傳下一號碼由呼叫者決定在哪個柜台處理。
* **示範 Case 1、2、3**
  + **功能**：分別模擬只有某一個或多個櫃台空閒的情況，呼叫下一號並印出剩餘隊列。
  + **心得**：簡單明瞭，展示如何使用排序 sorted([1,2]) 讓多櫃台同時呼叫時依照編號順序取號。若實務上要考慮「多櫃台同時閒」，或「櫃台處理時間不同」，可擴充為事件驅動 (event-driven) 模擬。

## 11. 小吃店顧客排隊與營收系統

* **class Customer**
  + **功能**：儲存顧客到達時間 (arrival\_time)、訂單細節 (order\_details)、消費金額 (amount) 以及 estimated\_wait（預估等待分鐘數）。
  + **心得**：設計將到達時間與預估等待時間分離，若要做更精準的排程（例如考慮平均用餐時間），可在 SnackShop 中根據實際離店時間更新 estimated\_wait。
* **class SnackShop**
  + **\_reset\_hour\_if\_needed(self)**
    - **功能**：若進入新小時或跨日，重置 hourly\_count（本小時已直接進店人數），並更新 current\_hour。
    - **心得**：這段時間檢查是關鍵，保證每一小時接客上限重置。如果系統需長時間運行，可能要考慮跨日切換時的特殊邏輯（例如每天總量限制）。
  + **arrive\_customer(self, customer)**
    - **功能**：新顧客到達時，若本小時接待人數未滿 (hourly\_count < max\_per\_hour)，則 estimated\_wait = 0，直接加入 current\_customers；否則將顧客加入 waiting\_queue，並依序估算等待小時數（hours\_wait = position // max\_per\_hour + 1），換算成分鐘。
    - **心得**：估算等待時間時僅以「位置 ÷ 每小時可接待」做粗略換算，若想更精確，可加入「平均用餐時間」與「實際離開時間」動態更新。另外，若要實做「顧客放棄」或「優先客戶」等機制，可在這裡修改對 waiting\_queue 的處理順序。
  + **process\_next(self)**
    - **功能**：模擬有顧客完成用餐，即從 current\_customers 取出最早進店的顧客 (pop(0))，推入 served\_stack；如果 waiting\_queue 非空，則將隊首顧客移入 current\_customers，並設 estimated\_wait = 0。
    - **心得**：「先進先出」地處理在店顧客，動作簡潔。若要考慮「服務時間不一」或「顧客離開順序不固定」，可以依據顧客到達時間 + 平均用餐時間建立 min-heap 型結構管理離店時機，讓 process\_next 自動在正確時間觸發。
  + **get\_waiting\_list(self) / get\_current\_customers(self)**
    - **功能**：回傳目前排隊名單或正在店內的顧客清單。
    - **心得**：回傳複製的 list 避免外部直接修改內部結構。如果要匯出給前端或 UI，這兩個方法提供了必要資訊。
  + **total\_sales(self) / average\_spending(self)**
    - **功能**：分別計算當日已服務顧客的總營收與平均消費。
    - **心得**：直接對 served\_stack 做累加與平均，若需更複雜報表（例如時段營收統計），可加入時間欄位篩選或額外累計欄位。
* **範例使用**
  + **功能**：模擬 12 位顧客陸續到店（for i in range(12)），示範如何呼叫 arrive\_customer、列印目前在店與排隊名單，並透過 process\_next 處理完成顧客，最後印出總營業額與平均消費。
  + **心得**：範例涵蓋了「接待 → 排隊 → 服務完成」三大流程，結構完整。如果要改成定時自動化排程，可結合 threading.Timer 或 asyncio，定期呼叫 arrive\_customer／process\_next。

## 12. 交通壅塞繞路系統（AVL Tree）

* **class AVLNode**
  + **功能**：節點屬性包含 key（道路名稱）、weight（流量）、height、left/right 子節點。
  + **心得**：weight 作為 AVL Tree 的排序依據，且同時也是「流量」值，用於判斷平衡因子與是否壅塞。此設計讓「結構平衡」與「流量監控」二合一。
* **class AVLTree**
  + **get\_height(self, node) / get\_balance(self, node)**
    - **功能**：分別回傳節點高度與平衡因子 (左高 − 右高)。
    - **心得**：基礎且必要，確保插入/刪除後能判斷失衡並旋轉。在大型圖形系統中，如果需要同時查詢「節點流量」與「平衡狀態」，這兩個方法能及時提供。
  + **right\_rotate(self, y) / left\_rotate(self, x)**
    - **功能**：分別實作右旋（LL / RR case）與左旋（RR / LL case），並更新節點高度，最後回傳新根。
    - **心得**：旋轉時會印出提示訊息（如「執行右旋 (RR / LL) 節點 Y → X」），可協助 debug。若要在 GUI 中可視化旋轉過程，建議改為回傳變動節點資訊給呼叫方，再由上層繪製。
  + **insert(self, node, key, weight)**
    - **功能**：先依 weight 做標準 BST 插入；再更新高度，計算平衡因子。若失衡，根據四種情況（LL、RR、LR、RL）呼叫對應旋轉；最後若平衡因子絕對值超過 bf\_threshold，則印出「道路壅塞，推薦替代路線」並呼叫 `recommend\_alternative()」。
    - **心得**：把「平衡樹」與「流量監控」結合在一起：當流量導致 BF 超過門檻，就直接觸發替代路線推薦。此設計直觀且實用；若未來要擴充「多源道路圖」，可能要把整顆樹抽象成圖結構，或改用 Dijkstra/A\* 做路徑規劃，但目前這個單樹版本已能示範基本思路。
  + **delete(self, node, weight)**
    - **功能**：先以 weight 找到節點並刪除，處理三種情況（無子、單子、雙子，用右子樹最小值替代）；刪除後更新高度、檢查平衡因子並做旋轉。
    - **心得**：與插入類似，保證刪除後仍能維持 AVL 平衡。如果道路關閉（刪除節點），同時確保剩餘結構仍能正確做替代路線推薦。
  + **get\_min\_value\_node(self, node)**
    - **功能**：回傳 node 子樹中最小 weight 節點（一路往左深度遍歷）。
    - **心得**：支援刪除時用最小值替代，維持 BST 性質。
  + **recommend\_alternative(self, congested\_node)**
    - **功能**：遍歷整棵樹(dfs)，找出所有 weight 小於 congested\_node.weight 的節點，根據流量由低到高排序，列出前 5 條路線做替代建議。
    - **心得**：這個方法把「流量比較」與「遞迴樹遍歷」結合，很直接；如果要做更精細的「最短路徑」或「考慮道路距離」等因素，須改用圖演算法。此版本適合「同一量測標準下推薦流量較低的路線」，足夠展示概念。
  + **add\_road(self, key, weight) / remove\_road(self, weight)**
    - **功能**：分別呼叫 insert() 或 delete()，外部介面簡化使用者操作。
    - **心得**：在呼叫這兩個方法時，會自動完成所有平衡、壅塞檢測與替代推薦；如果要建立更完整的系統，還可加上「查詢指定道路是否壅塞」或「查詢特定流量範圍的道路清單」等查詢介面。
  + **inorder(self, node)**
    - **功能**：中序列印整棵樹節點的 (key:weight)，方便 debug。
    - **心得**：中序走訪即能依流量從低到高列印，適合作為日常檢視或日誌輸出；未來若要可視化整顆AV L結構，可把走訪結果轉成圖形資料再輸出至前端。
* **範例使用 (if \_\_name\_\_ == "\_\_main\_\_")**
  + **功能**：依序新增多條道路（名稱 + 流量），印出每次插入後的中序結果，若有壅塞即顯示替代路線。
  + **心得**：範例說明了當 weight 越大（流量越高），會進行旋轉與平衡調整，同時觸發 recommend\_alternative。可作為壅塞監控系統的最簡版範例，如果想做成動態可視化儀表板，可把每次插入/刪除事件與結果推送給前端。

### 總結

整份專案涵蓋了多種常見資料結構、演算法與實際應用案例，每個模組的主要函式都各自聚焦在特定領域邏輯：

* **Huffman、堆與隊列**：展現了經典壓縮與優先/排隊處理思維。
* **Game of Life、無限級數、基本運算器**：示範了演算法思維與數值計算。
* **BST、AVL Tree**：從無平衡到平衡二元樹的升級過程，並結合「流量監控／壅塞推薦」。
* **單向連結串列、簡易 I/O 系統**：提供最基礎的資料結構操作範例。
* **SnackShop 排隊與營收**：結合真實場景下的排隊邏輯與統計需求。

如果有任何具體模組想更深入討論或優化，隨時告訴我！