112 年特種考試地方政府公務人員考試試題

等　別：三等考試
類　科：資訊處理
科　目：資料結構

一、請以 C, C++, C#, Java 或 Python 撰寫 2 個方法，一個以迴圈方式，一個以遞迴方式，對存在 singular linked list 的資料進行 linearly search。假設 Node 的結構如下：（12 分）

```
class Node {
    int data=0;
    Node next=null;
    Node(int dd, Node nn) { data=dd; next=nn; }
```

【解題關鍵】
《考題難易》：★★
《破題關鍵》：本題為鏈結串列的基本操作題，只要了解鏈結串列線性搜尋的迴圈與遞迴演算法即可作答。

【擬答】

```
class Node:
    def __init__(self, data, next_node=None):
        self.data = data
        self.next = next_node

# 迴圈方式 linear search
def linear_search_iterative(head, target):
    current = head
    while current is not None:
        if current.data == target:
            return True  # 找到目標值，返回 True
        current = current.next
    return False  # 當前節點為空，表示未找到目標值，返回 False

# 遞迴方式 linear search
def linear_search_recursive(node, target):
    if node is None:
        return False  # 遞迴到底部，未找到目標值，返回 False
    if node.data == target:
        return True  # 找到目標值，返回 True
    return linear_search_recursive(node.next, target)  # 遞迴到下一個節點

# 範例使用
# 建立一個 linked list: 1 -> 2 -> 3 -> 4 -> 5
head = Node(1, Node(2, Node(3, Node(4, Node(5)))))
```

```
# 使用迴圈方式進行 linear search
target_value = 3
result_iterative = linear_search_iterative(head, target_value)
print(f"迴圈方式：找到目標值 {target_value} - {result_iterative}")

# 使用遞迴方式進行 linear search
result_recursive = linear_search_recursive(head, target_value)
print(f"遞迴方式：找到目標值 {target_value} - {result_recursive}")
```

二、請為數列 0, 10, 30, 20, 50, 80, 40, 90, 70, 60 建立 AVL tree, Min/Max heap, 2−4 tree，並依它們的性質以 yes or no 完成下表。註：所建立的 tree or heap 請以圖示，如果是 Searching Tree，請以左小右大的方式建立。（24 分）

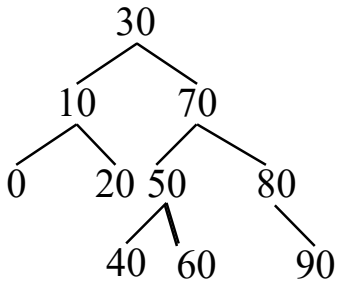
	Balance	Searching Tree
AVL tree		
Min heap		
Max heap		
2−4 tree		

【解題關鍵】
《考題難易》：★★
《破題關鍵》：本題為搜尋樹、堆積的概念題，只要了解搜尋樹、堆積、2-4 tree 的操作即可作答。

【擬答】

㈠ AVL Tree:

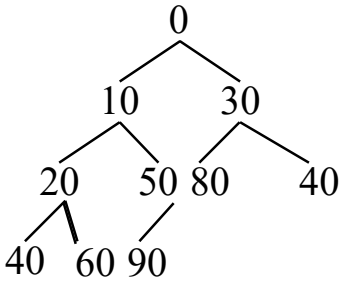
AVL tree 是一種平衡二元搜索樹，左子樹的高度和右子樹的高度最多相差 1。本題在加入 30 時會進行 RR 旋轉、加入 80 時會進行 RR 旋轉、加入 60 時會進行 RL 旋轉，最後得到下列 AVL Tree：



Balance Searching Tree: Yes

㈡ Min Heap:

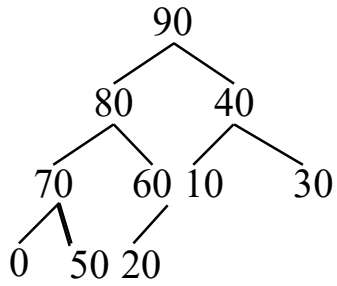
Min heap 是一種 Complete 二元樹，父節點的值小於或等於其子節點的值。依照上列順序加入後，無須調整即可得下列 Min heap：



Balance Searching Tree: No

㈢ Max Heap:

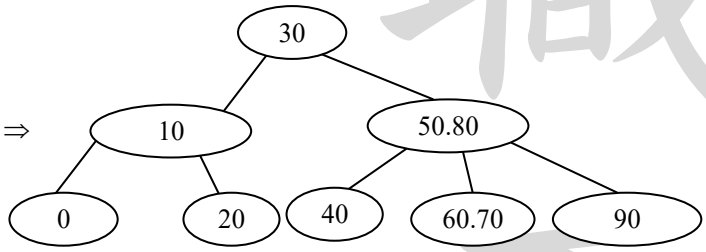
Max heap 是一種 Complete 二元樹，父節點的值大於或等於其子節點的值。依照上列順序加入後，幾乎每次加入均須調整，最後可得下列 Max heap：



Balance Searching Tree: No

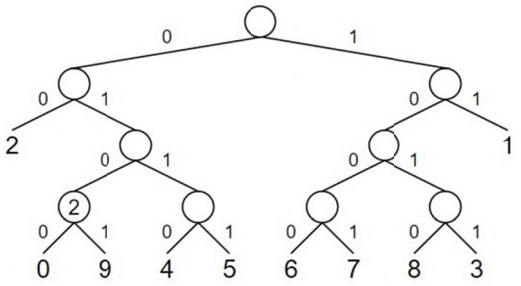
㈣ 2−4 Tree:

2−4 tree 是一種平衡的搜尋樹，是 degree=4 的 B 樹，每個節點可以包含 2 到 4 個子節點。在加入 20 時會進行分裂，加入 80 時會進行分裂，加入 90 時會進行分裂，加入 60 時會進行分裂，最後得到下列 2−4 tree



Balance Searching Tree: Yes

三、請以如下的 Huffman Tree 所做的數字編碼，解讀 01010111110100100011 編碼對應的數字。（10 分）

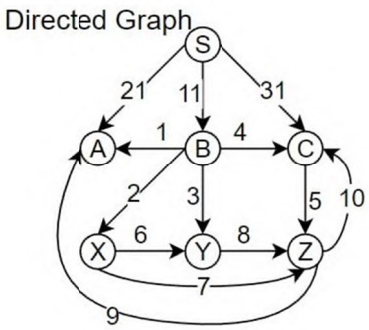


【解題關鍵】
《考題難易》：★★
《破題關鍵》：本題為 Huffman Tree 的基本應用題，只要了解 Huffman Tree 解碼過程即可作答。

【擬答】

0101(9)　0111(5) 11(1) 0100(0) 1000(6) 11(1)
因此解讀為 951061

四、針對如下的有向圖（節點為走訪對象，連線上的數字為走訪的 cost），依如下 BFS（配合 queue）與 DFS（配合 stack）演算法，進行所有節點的走訪，多個節點可以走訪時，以連線上 cost 較低者優先，結果請以迴圈內部的顯示要求，依下表形式填入（stack 垂直表示，開口在上方，queue 水平表示，出口在左，入口在右）。註：假設節點 S 為起始點。（24 分）



BFS 演算法	Loop1	…
print node		
queue		
processSet		

DFS 演算法	Loop1	…
print node		
Stack		
processSet		

BFS/DFS 演算法（/前為 BFS 使用 queue,/ 後為 DFS 使用 stack

Step1: set queue/stack to empty
　　set processSet to empty
Step2: enqueue/push S and add S into processSet
Step3: while queue/stack is not empty
Step31: dequeue/pop and print it
Step32: enqueue/push all one step neighbors which are not in processSet
　　according to the cost of edges and add them into processSet
Step33: display content of queue/stack and processSet

【解題關鍵】
《考題難易》：★★
《破題關鍵》：本題為圖形走訪基本操作題，只要了解有向圖 BFS 與 DFS 實作過程即可作答。

【擬答】依據題意，可將本題以下列 Python 程式碼實現：

```
from collections import deque
# 定義有向圖的邊
edges = [('S', 'A', 21), ('S', 'B', 11), ('S', 'C', 31),
         ('B', 'A', 1), ('B', 'C', 4), ('B', 'X', 2),
         ('B', 'Y', 3), ('C', 'Z', 5), ('X', 'Y', 6),
         ('X', 'Z', 7), ('Y', 'Z', 8), ('Z', 'C', 10),
         ('Z', 'A', 9)]
# 初始化圖的資料結構
graph = {}
```

```
    for edge in edges:
        start, end, cost = edge
        if start not in graph:
            graph[start] = []
        graph[start].append((end, cost))
    # BFS 走訪
    def bfs(start):
        queue = deque([start])
        process_set = set([start])
        i = 0
        while queue:
            current_node = queue.popleft()
            i += 1
            print(f'Loop ',i)
            print(f"print node: {current_node}")
            neighbors = sorted(graph.get(current_node, []), key=lambda x: x[1])
            for neighbor, _ in neighbors:
                if neighbor not in process_set:
                    queue.append(neighbor)
                    process_set.add(neighbor)
            print(f"queue: {', '.join(queue)}")
            print(f"processSet: {', '.join(process_set)}\n")
    # DFS 走訪
    def dfs(start):
        stack = [start]
        process_set = set()
        i = 0
        while stack:
            current_node = stack.pop()
            i += 1
            print(f'Loop ',i)
            print(f"print node: {current_node}")
            if current_node not in process_set:
                process_set.add(current_node)
                neighbors = sorted(graph.get(current_node, []), key=lambda x: x[1], reverse=True)
                stack.extend(neighbor for neighbor, _ in neighbors if neighbor not in process_set)
            print(f"stack: {', '.join(stack)}")
            print(f"processSet: {', '.join(process_set)}\n")
    # 測試
    print("BFS 走訪結果：")
    bfs('S')
    print("\nDFS 走訪結果：")
```

dfs('S')以下是根據給定的有向圖和演算法進行 BFS 和 DFS 的走訪過程：

BFS 走訪結果：

Loop 1
print node: S
queue: B, A, C
processSet: S, B, C, A

Loop 2
print node: B
queue: A, C, X, Y
processSet: S, A, Y, X, B, C

Loop 3
print node: A
queue: C, X, Y
processSet: S, A, Y, X, B, C

Loop 4
print node: C
queue: X, Y, Z
processSet: S, A, Y, X, Z, B, C

Loop 5
print node: X
queue: Y, Z
processSet: S, A, Y, X, Z, B, C

Loop 6
print node: Y
queue: Z
processSet: S, A, Y, X, Z, B, C

Loop 7
print node: Z
queue:
processSet: S, A, Y, X, Z, B, C

DFS 走訪結果：

Loop 1
print node: S
stack: C, A, B

processSet: S

Loop　2
print node: B
stack: C, A, C, Y, X, A
processSet: S, B

Loop　3
print node: A
stack: C, A, C, Y, X
processSet: S, B, A

Loop　4
print node: X
stack: C, A, C, Y, Z, Y
processSet: S, B, X, A

Loop　5
print node: Y
stack: C, A, C, Y, Z, Z
processSet: S, A, Y, X, B

Loop　6
print node: Z
stack: C, A, C, Y, Z, C
processSet: S, A, Y, X, Z, B

Loop　7
print node: C
stack: C, A, C, Y, Z
processSet: S, A, Y, X, Z, B, C

Loop　8
print node: Z
stack: C, A, C, Y
processSet: S, A, Y, X, Z, B, C

Loop　9
print node: Y
stack: C, A, C
processSet: S, A, Y, X, Z, B, C

Loop　10
print node: C
stack: C, A
processSet: S, A, Y, X, Z, B, C

Loop　11
print node: A
stack: C
processSet: S, A, Y, X, Z, B, C

Loop　12
print node: C
stack:
processSet: S, A, Y, X, Z, B, C

五、請完成下列表格有關排序演算法的 time complexity（假設排序資料有 n 個，資料位數有 d 個）、是否為 In-Space 演算法、是否為 Stable 演算法及範例數列 50, 46, 37, 28, 19 進行降冪排列時所需的比較次數。（30 分）

排序演算法	Time Complexity		In-Space（Yes/No）	Stable（Yes/No）	降冪比較次數
	Best	Worst			50, 46, 37, 28, 19
Bubble					
Insertion					
Merge（奇數時，後半段多 1）					
Quick（第一個當 pivot）					
Radix（base 10）					
Selection					

【解題關鍵】
《考題難易》：★★
《破題關鍵》：本題為排序法的基本比較題，只要了解各種排序演算法即可作答。

【擬答】

以下是排序演算法的 time complexity、是否為 In-Space 演算法、是否為 Stable 演算法以及降冪排列所需的比較次數：

排序演算法	Time Complexity		In-Space（Yes/No）	Stable（Yes/No）	降冪比較次數
	Best	Worst			50, 46, 37, 28, 19
Bubble	O(n)	O(n^2)	Yes	Yes	10
Insertion	O(n)	O(n^2)	Yes	Yes	10
Merge（奇數時，後半段多 1）	O(n log n)	O(n log n)	No	Yes	7
Quick（第一個當 pivot）	O(n log n)	O(n^2)	Yes	No	10
Radix（base 10）	O(nk)	O(nk)	No	Yes	7
Selection	O(n^2)	O(n^2)	Yes	No	10