程序導向:照食譜一步一步做

物件導向:先把食材個別處理好，最後在照順序跑(變回程序導向)

物件導向

重複出現的程式可以變成模組

物件導向是將問題的資料屬性和資料本身的相關操作一起思考

例如:蓋房子，窗戶、木門…都變成模組。再蓋的時候，窗戶就可以堆疊就

好，不用每個都重作

讓物件之間互相配合(像是組合積木，但有順序)

物件之間的溝同需要「訊息」配合

程式=物件+訊息

物件導向程式設計可以視為一種組裝程式的過程

同樣的「物件」可以抽象化為「類別」

每一個類別有自己的屬性、方法

Ex:交通工具🡪可分為陸海空🡪空中交通工具🡪飛機🡪某種飛機(實體)

Ex:學生類別的屬性🡪年紀、年齡…(描述資料的內容)

Ex:學生類別的方法🡪選課資格(有什麼動作可以執行，操作或行為)

每一個物件都有獨一無二的「識別」，例如學生就是學號

在程式代表的就是，某物件在記憶體裡面的位置

類別之間有關連性

Part of 某類別是其他類別的零件

Wheel車輪和driving computer 行車電腦是car車類別的零件

抽象類別

不能代表物件的類別(無法實體化)，例如哺乳類不能產生實體

定義類別

Class Person(類別名稱):

Pass(類別內容)

Robert=Person() --> Robert(實體)是person的類別

當你建立一個物件，就會有一個識別(放在記憶體的某個位置)

Print(id(person))

Print(id(robert))

方法:function

Def function(a1,a2):

ex:

def hallo(name): name是引數

print(“hello”+name)

Hello(“john”) 🡪call出function

方法裡面要求的參數一定要對應

但也可以不寫例如

def hallo(name=John):

print(“hello”+name)

一開始就給預設值

假設:

Def add(a=1,b=1)

Print(a+b)

Add(a=2,b=1) 這樣就會把預設值改掉

Class person:

Name=none

Age=none

Def setData(self,name,age):

Self.name=name

Self.age=age

註記:self因為每個學生的名字不同而設立

如果要調特定資料，就可以用self.名字

Robert=person()

Robert.setData(Robert,48)

Person是類別

Robert是person底下的物件

Setdata設定了robert的屬性

一個class裡面可以有多個function

Class person:

Def \_init\_(self,name,age):

Self.name=name#實體變數

Self.age=age

Def say\_hello(self):#方法

Print(hello,I am+self.name)

P1=(amy,25)

P1.say\_hello()

函數yeild

每次call程式都會把結果留下來

Return是一次給所有結果

Instance method實體方法

必須要帶預設參數self當作實體

Class method 類別方法

必須要帶預設參數cls，當作類別

Static method靜態方法

不用帶任何參數，物件可以使用，本身也可以使用

大家都可以用

類別下面有許多物件(self)

靜態方式的方法，所有人都可以叫

Self只有物件可以使用

Cls類別跟物件都可以使用

物件導向有封裝性

可以將物件隱藏

物件的繼承

又稱一般化，父類別有的子類別也會有

新的class在括號內加上上一個class

Class eltriccar(Car):

用super(EletricCar,self).\_init\_(…) #父類別寫過的程式用super函數直接繼承

要加新的再寫

Self.brand=brand

若有多重繼承的問題(屬性撞到)，會以先寫得為主

繼承不一定要是實體，可以有抽象類別(簡稱abc套件)

多型

假設function名稱都一樣，可以互相覆蓋

但class要不同

Ex:根據資料不同類型，會自動改變處理方式。針對字串和list處理方式不

同

類似於覆蓋的功能

9/29/20

Static method

讓這個function能始終存在在記憶體內，就不用寫self。全部都能用。

動態連結(dynamic binding)

可以隨時修正

Ex:a=2 整數 會在記憶體挖個洞讓a指向2這個過程就叫linking

若要更新成 a=3 那麼就會讓a指向3 然後原本的2會在記憶體裡被回收

Console(視窗介面)的程式語言

1.c,c++(硬體)🡪可控制記憶體(沒有用的指向手動釋放)

2.Java(沒有用的指向會自動釋放)

3.python(沒有用的指向會自動釋放)

Web program

1.HTML,CSS,Java script

2.ASP.net,PHP,JSP(動態連結)

3.MySQL

A=6

B=6

A is B -->判斷物件的id(在記憶體中的位置)是否相同

A==B 🡪判斷物件的數值香不相同

List也是相同的狀況，只是裡面的屬性對指不同記憶體空間但是串在一起

變數的有效範圍

區域變數(在函式裡面寫的)

全域變數(全部都能用)

傳遞方式

1.Pass by value

2.Pass by reference

3.Pass object by reference

1.在呼叫函數的時候，把船進去的參數複製一份給記憶體讓他去連這個空間

但因為是複製進去的，所以函數裡面怎麼變，都不會影響原來的東西

EX: mylist=[x] mylist在記憶體裡面挖個洞叫x

有一個add(list)的func

add(mylist) 他會把函數的list指向複製的list值也就是x(會用到不同記憶體位置)

在add裡面怎麼改都不會動到原本的mylist

2.根據上述，但差別在於函數會指向同一個空間。所以函數跑完mylist會變

3. pass by sharing (python獨有)

Def foo(lst):

Lst.append(1)

Lst=[2]

M=[]

Print(foo(m)) 結果會是1

因為lst=[2]他會重新定位在記憶體的位置(會換新的位置)

所以在python只要用到 = 通常都會挖新的位置

總結:有做任何的修正，他都會開新的位置存

如果把 a=[] 然後用 a.extend 的話就會在原本的位置擴展

但如果用append就會改位置

例外處理

有錯誤或不合理🡪中斷

但如果你不想中斷🡪 try except

還可以加 else(有except以外的事件) finally(全部做完之後的動作)

Property

所有跟屬性有關的function都可以用property()包起來做

@property常用寫法

可以針對每個動作來做說明

資料結構與演算法

把資料定義得很清楚可以增加搜尋的速度

EX: array 有0~4 那要怎麼很快速地找到4?-->有效率的程式(演算法)

資料結構習慣用excel的格式(比較少用的圖像、影像、文字、網路)

演算法:解決問題的步驟

方法的好壞(記憶體消耗、執行速度)

演算法:能夠利用電腦解決問題的步驟

五個特性:

Input(輸入):例如營業額

Output(輸出):例如要讓整體營業額提升

Definiteness(明確):例如影響業績的各種面向，要拆開來一個一個看

Finiteness(有限性):例如影響業績的各種面向，會漏掉一些

Effectiveness(有效性):實踐後的結果

看到loop先看裡面，再看外面

For寫越多，越慢

演算法效能

1.big-o

2.Omega

3.Theata

速度:常數時間<對數時間<線性時間<次方時間<指數時間<皆乘時間

遞迴演算法

逐步化簡、性質相同的問題

要設定好終止條件

一般情況怎麼遞迴

例如:自己call自己

重複性的工作可以寫成函式，並呼叫自己

陣列array

Excel就是二維陣列，但缺點是很消耗空間，而且沒用的資料(0)很多

noSQL 只存有值的資料

或是用稀疏矩陣來記

遞迴VS迴圈例子

輾轉相除法(迴圈)

1.判斷ab誰大

2.a%b 餘數r

3.b%r 餘數r2🡪判斷是否為0

4.終止條件

If a<b: #先判斷大小

Swap(a,b)🡪a,b=b,a

While r !=0:

r=a%b

a=b

b=r

return a

#迴圈的優點，你可以知道每個答案，知道幾次會有結果。缺點程式碼長

#遞迴的缺點，就是沒辦法控制結束條件，你不知道第幾層會有答案。但精簡。

Link list VS array

Array+link list是最重要的

資料結構用array或link list可以解決

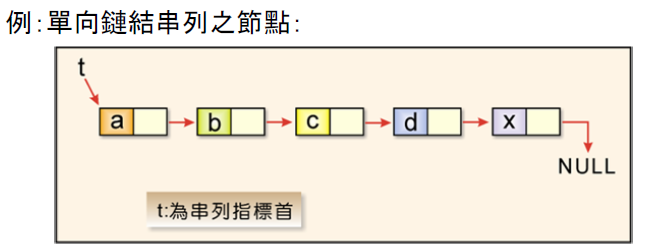
Array的缺點是搬移資料很麻煩，但好處是可以用index去查某個

也可以用for從第一個跑到最後一個

Link list 像火車一樣，可以隨時卸載、鏈結

把不同數值連載一起的稱為鏈結(linking or pointer)

某個list t=[a,b,c,d,e] null就是空的(功能像機器的接地線)



Link list不需要占用連續的記憶體空間，array要

Link list 會隨便站記憶體的位子，之後再link起來

各Node 不一定要佔用連續的 Memory 空間

Node 之型態 (Data Type) 不一定要相同

僅支援 Sequential Access

Insert/Delete Node 容易

只要把接點重接就好

(假設有a,c要新增b，先把b的接點指向c然後把a的接點指向b)

Circularly Linked Lists(環狀)

將Single link list中，最後一個node的指標指回第一個node

Doubly Linked List (雙向鏈結串列)

Link list的缺點是沒辦法回頭看，要繞回去才可以

所以雙向的話(一個前，一個後)就可以達成回頭查找

但是insert/delete會變得比較複雜，但一樣尊崇「先連好新的再回收」

Link list concat/merge

Concat 合併(單純合併)

Merge 資料表有key才能用(依照某個欄位)

Concat

假設有AB的link list我要用C把兩個合併

設定C的開頭，把A先跑完然後把最後一個接到B

把B跑完最後的決定要接地或接回頭

Link list的圖表

和array直接挖一個大空間塞圖表不同

他只會存有連線的，沒連線的不存，而且記憶體分開放

Link list文字

假設要用瀏覽器搜尋資料，如何找到最符合的?

標題、摘要、連續出現

1.把摘要切成一個個詞

例如: I am a boy、I am a girl、I am happy

然後array就會是一個表格，分別代表出現哪些字的時候會有怎樣結果

例如輸入I 就會有三本書，表格呈現1,1,1以此類推

以上是array的作法，缺點是無用資訊很多，就會跑很久

用link list的做法就是一樣切成字

但是只有有出現的才會連結，出現最多次的排在最前面(優先考慮)

這樣搜尋就會變快

Stack堆疊

常用在算術運算

第一個進來的會最後一個走FILO(first in last out)

插入用push

彈出用pop

所有動作都只能從第一個(top)做

後進先出、先進後出

Top是指標，動作都會對top做

Top(S)

isEmpty(S)

isFull(S)

stack功能

運算式有中序、後序、前序

中序🡪一般人看的，要用括號來看誰先做

後序、前序不用

運算子有優先順序差別

請看ppt「DS」 35頁

順序法:在stack裡面遇到同等級就彈出，括號完成將括號內彈出。

+,-不能高於\*,/

10/13/20 複習

練習:

S=’happy’

S[::-1]

🡪yppah

迴圈

For I in range:

🡪yppah 從最後面一個一個拉出來

遞迴

Def revs(s):

If s==” ”

Return s

Else

Return revs(s[1:])+s[0]

Tail recursive

由於遞迴每次到最後才知道答案

為了改良，可以將每次的答案或是特定次數的答案存取起來

#Tail recursive

def tailfib(n,tmp=0,result=1):

if n==1:

return result

else:

return tailfib(n-1,result,tmp+result)

讓tmp存上一次的東西，result把現有的就加起來

Python有個東西可以讓tail recursive的速度更快 @classmethod

link list做stack

設訂一個頭(top)指向某一個車廂

堆疊只有一個top拿掉一個，下一個就變成top

Queue佇列

跟stack不一樣，可以看到頭跟尾。Stack橫放的感覺

如果頭跟尾相同，代表array是0

Top-rear !=0才是有東西

比較常做成環狀，可以任意從頭或尾增加刪除

若我要在尾部增加新的資料，把原本指向接地的指向新資料，再把尾指向

新資料(請見DS PPT 40頁)

只有圓弧狀的queue會用到mod找出餘數

為的是能夠繞一圈後繼續對應位置

Tree樹

能表達 if else 以各種條件走不同分支的路徑

決策樹如何以array表達?

但儲存方法很多取決於你如何定義(例子請見DS PPT 43)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | E | F | C | D | G | H | I |

這個方式是一層一層儲存(廣度搜尋)

也可以一條條分支存(深度搜尋)若為空也會保留位置

要取決於自己搜尋的目標、樹的結構來決定用哪個

通常都要嘗試看哪個好

兄弟🡪同一個父節點的所有子節點

祖先🡪樹根到該節點路徑中經過的所有節點

Leaf葉子🡪分支為0的節點

Level階度🡪從樹根到該節點的距離，樹根為1

Height高度🡪一棵樹level值的最大值

用link list表現樹(請見DS PPT 49)

但是很浪費空間，而且很難改進🡪因此binary tree出現(只有兩個分支)

限制節點數，左右子樹有次序之分，又稱order tree

樹具有遞迴關係，每個分支點都可以當作一個root來看，再分左右根

Huffman algo.

計算每個字出現的頻率

把小的數字放左邊，然後找合起來最小的數字組合變成新的數字

建構出樹狀，節點最多只接兩個

節點左邊連結的是0右邊是1

例如:2是1,0,0,0

 最常出現的，用較短數字表達

補充:

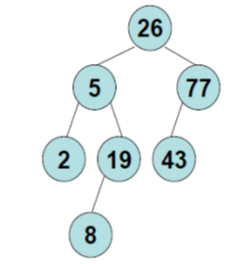
拿到資料後🡪可以label encoding 或是 one hot encoding

把類別變項的資料轉換為數字處理

10/20/20

Binary search tree(BST)

首先，這是一種Binary tree 但是左分支一定小於root右分支一定大於root

範例

減少從頭開始搜尋的不方便

依照inorder來搜尋，從左邊走完(小)再走中(根)再走右(大)。會自動排序。

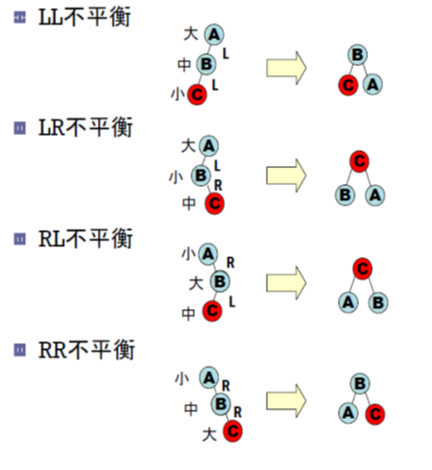
AVL tree 平衡樹

BTS可能會因為root選擇不當而變得不平衡ex不小心挑到最大或最小的當

root

繼承於BST和BT必須兩者都符合才是AVL樹。

兩邊的層數只能差一



解決方法就是把中間的拉起來

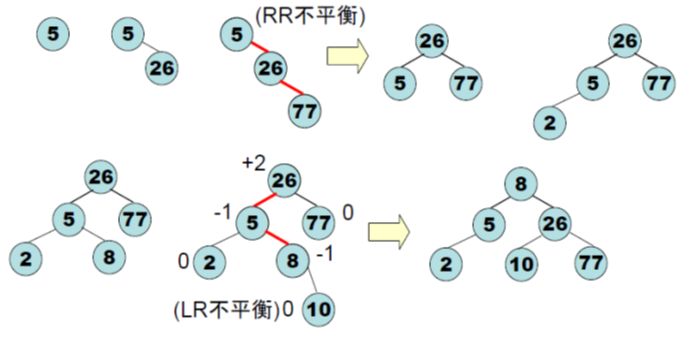
例題:5,26,77,2,8,10,19,12

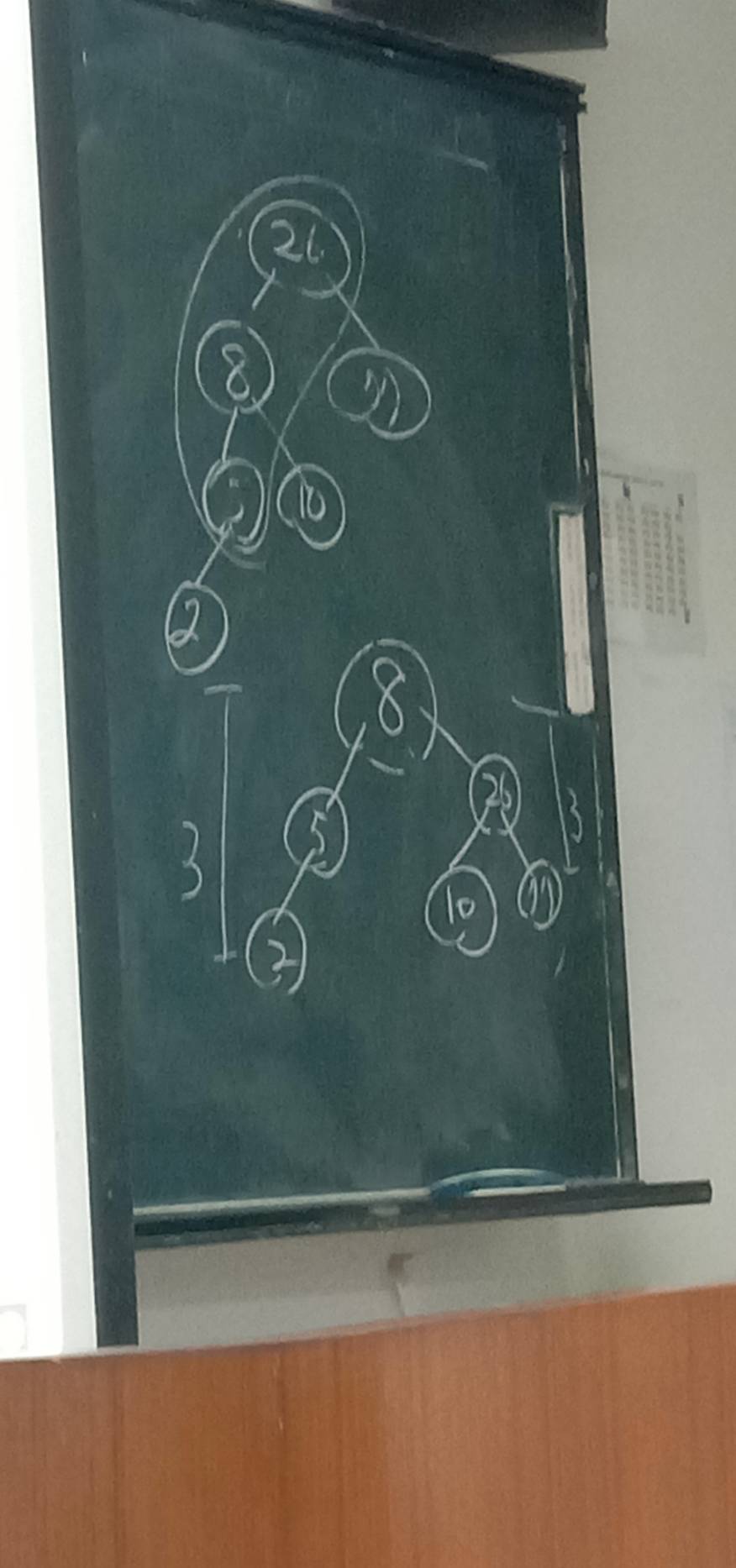
程式會從頭開始拜訪，一邊建構

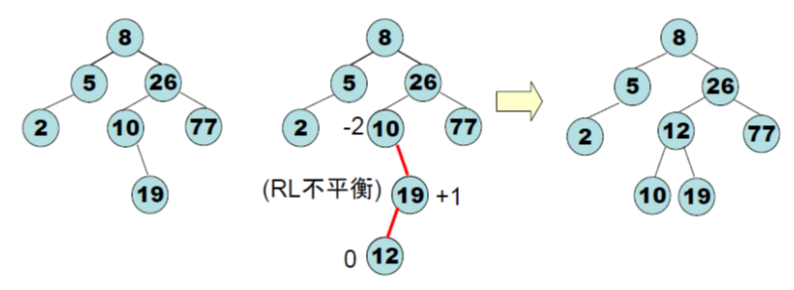
BST會整個右偏層數(2:6) 高度越深找越久

AVL可以整理成3層

每多一層可以容納2n-1的點(理想狀態，不偏)







heap tree 堆積樹

根小於分支點

只要先做好binary tree即可

若要找最小值，這種最好用(最快)

max heap tree

根要大於分支點

若分支點比較大就和根交換

若要找最大值，這種最好用(最快)

若有刪除根結點，把最後項(最右下)排到root的位置，再進行交換

適合用於找top幾的時候

Min-max heaps tree

例如下棋，最大化自身利益，最小化對手利益

對抗競賽時使用

min層之間淺層的min要小於深層的min

max層之間淺層的max要大於深層的max

若插入新的值，要跟全部的數值比較看屬於哪個位置，交換後再調整

刪除某值後，用最後一個(分支的最下最右的值)來補，之後繼續交換調整

先調整min層再調整max層

Deaps雙向堆積樹

樹根不包含任何數(空的)

左子樹要比右子樹小

左邊子樹是min heap 右邊子樹是max heap

若新增數值，要看兩邊相對應的位置比大小，如有需要就要交換

交換後要調整(左邊min小的往上交換，右邊max大的往上交換)

若刪除值，則是用最尾端的值補上後再進行調整

Tree的問題

最常希望的平衡是「矮胖型」🡪但一個節點要存的值就不只兩個

可以限定範圍。Tree degree=m

增加tree degree 就能降低search tree的高度(搜尋速度加快)

同時也能降低disk I/O的速度提升external seatch的效益

一個節點超過兩個分支🡪又稱為m-way tree

補充:search tree主要用於外部搜尋，因為資料量大不可能全塞在memory

B-tree

目前最常用是m=2,3,4的tree

左邊最小，往右越大

根一定要有兩個節點

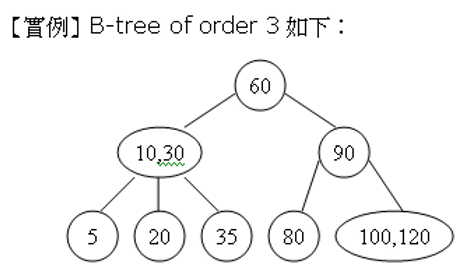
每一個節點數值最多可以存m-1個(一個node可以存複數個值)

除了樹根和葉節點外，其餘節點最多能分之m/2(取uper整數)和m之間。

例如m=3一個節點就只能取2~3個分支

要保證葉節點都要在同一層

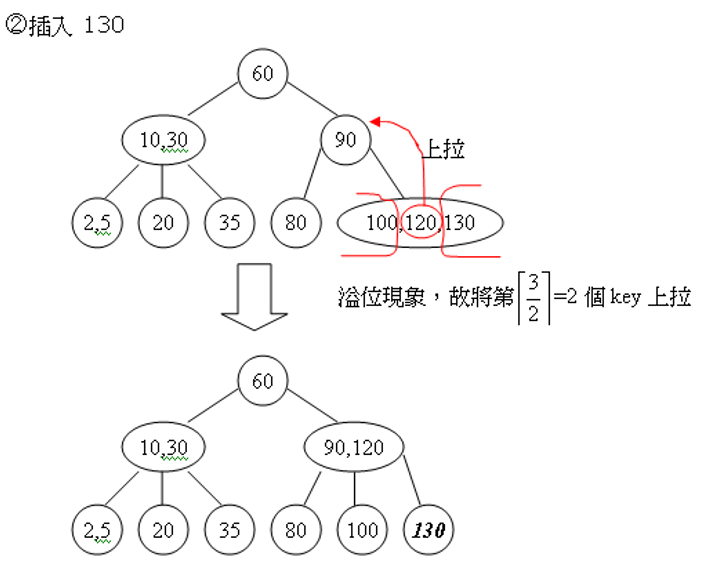
Search很容易，只要看要查的數值介於哪兩個值之間就好



插入新的數值，根據其所在的區間去新增，可以存在同一個節點

要讓葉子保持在同一個階層

溢位現象(節點能存的值超過時)的處理🡪把中間的值往上拉



刪除數值時可能會導致某一個葉節點空值稱為underflow

Underflow，這時需要用rotation從別的節點借值，一樣是中間值往上拉其

餘往下放

如果刪除root，拿左子樹的最右邊，或是右子樹的最左邊來補(中間值)

移動交換的次數比較少

B-tree的調整結果可能不同，但只要符合條件就可以。

例如:調整2-3樹，一個node可以存兩個值，並可以有三個分支。同時符

合左邊小右邊大。葉節點同一個水平。

B+ tree

用於資料庫的搜尋上，找任何一筆資料都很快

所有的資料都在葉子節點

上面的node全部都是比較用的索引值(index)

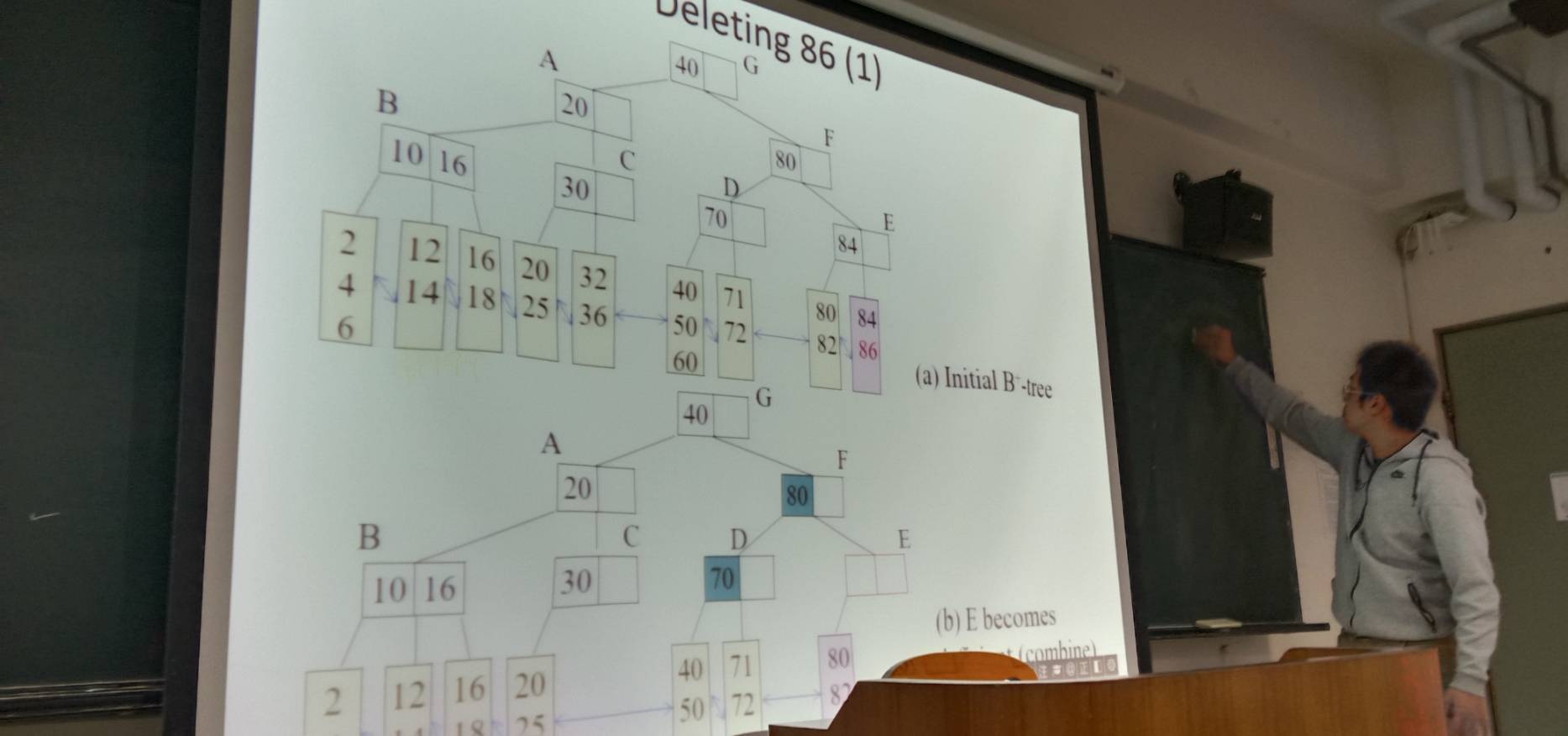
樹狀搜尋，為了更方便查找，葉節點會額外做double linked list

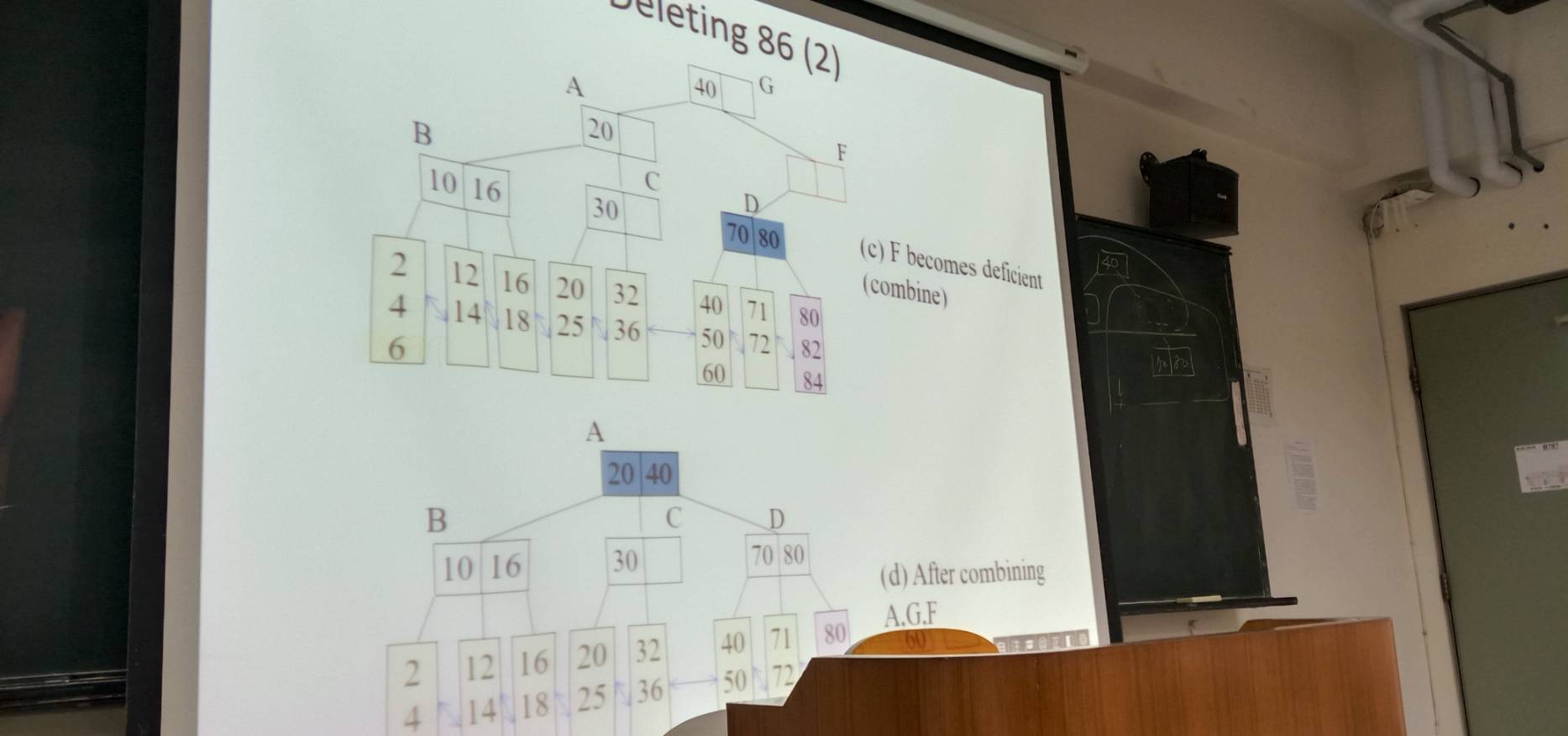
讓葉節點可以互通搜尋，不用每次都從root重新開始

如果要存新資料，全部都要放在葉節點。

並且盡量讓葉節點存的數量平衡。(找得快)

進行平衡的方法和Btree一樣，但是資料不往上抬，只有值往上抬。

假設要刪掉86



Graph

分成有向圖和無向圖。

通常點之間會相互連通。

有向圖會呈現點的方向。無向圖則沒有。

Complete graph是一個擁有最多非重複邊線的圖

例如:社群中的小團體(聯絡密切的一群人)

推薦系統也會使用。

Subgraph子圖，從原本的graph切出一小塊。

Graph在意路徑長度，通常會選擇簡單路徑

simple path(除了起點和終點可以相同外，其餘點都不相同)

Cycle:簡單路徑，起點與終點相同

Connected:連通性:無向圖中每個點都有路徑相連

有向圖中還可以分成強連通或弱連通

強連通:代表每個頂點都有路徑連到對方

弱連通:有兩個點或以上沒有路徑連到對方

Degree:分支度(無向圖中有連到某個頂點的邊的個數)

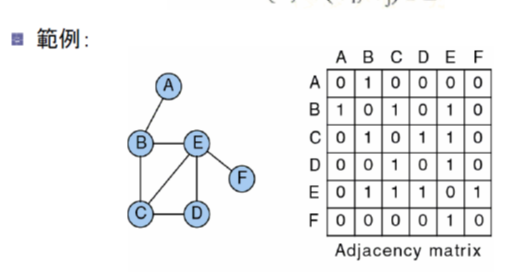
Outdegree:離開某個頂點的邊的個數(有向圖中往外連幾次)

Indegree:進入某個頂點的邊的個數(有向圖中連到自己幾次)

看degree可以找到哪些人是散布中心或聚集中心。

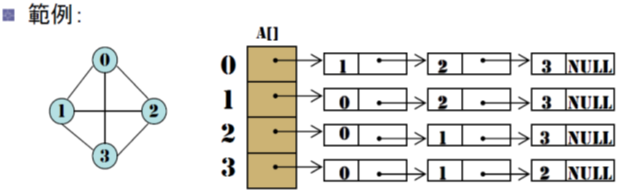
Graph可以用linked list和array來存。存法不太一樣。

array



有連結就是1，沒連結為0。

Linked list



以一維矩陣，將每個點的連結都列出來。

補充:特徵工程

假設我有一個資料表紀錄每個人的匯錢紀錄的金額與時間

時間對電腦而言只是字串，但對人而言是有意義的(早中晚)

可以劃出graph藉由degree來看誰是收錢大戶或出錢大戶

歐拉路徑

Complete graph

算處個頂點的degree

若全部都奇數🡪畫不出來

但如果是奇數開頭，最後也會是奇數結尾

偶數的話會回到原本的點結束

Hamilton

每個點都去過最後剛好回到起點(周遊世界問題)