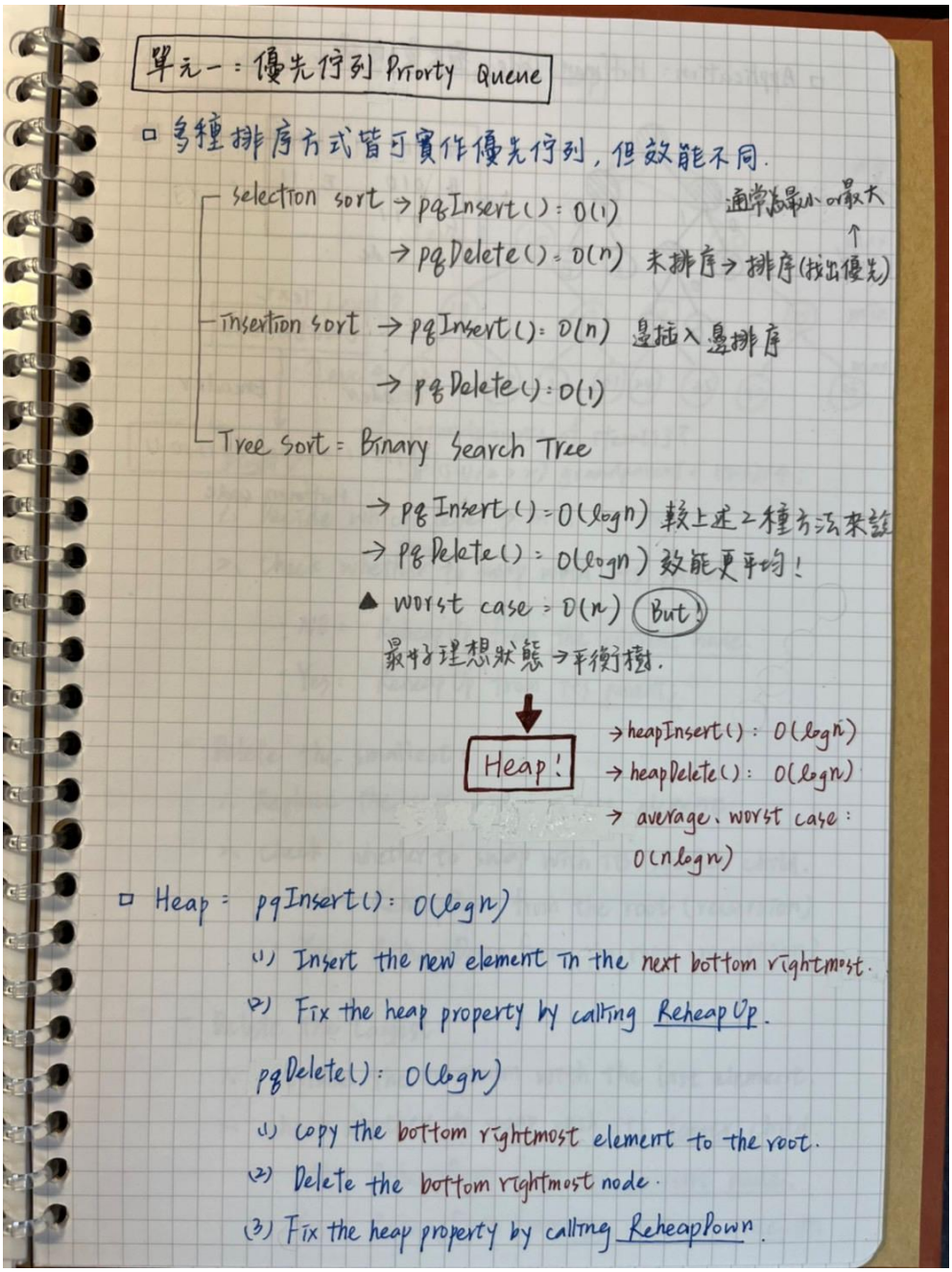
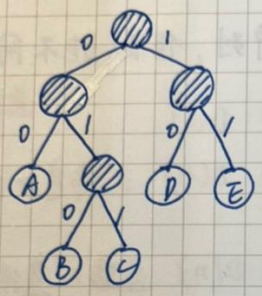
資節筆記 1-4

10927159 資訊二甲 林玟君

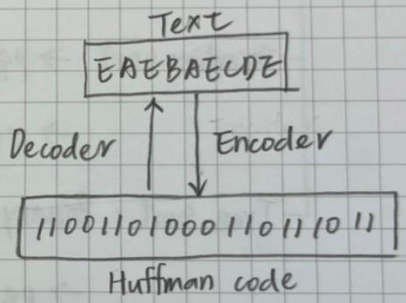


□ Application: Huffman code. 霍夫曼編碼



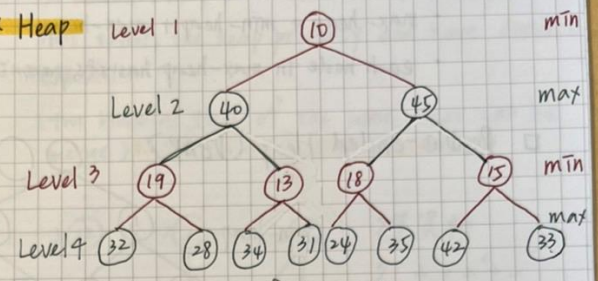
A: 00	D: 10
B: 010	E: 11
C: 011	

code



單元二：堆積變形 Variations of Heap

□ Min-max Heap



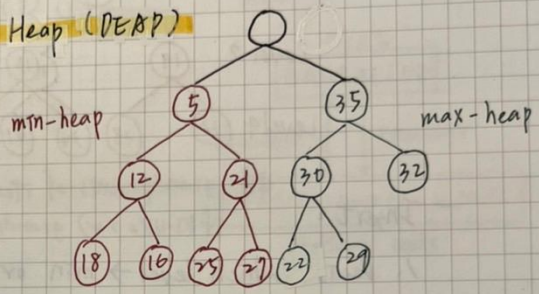
補：grandparents of item[i]?
if((i-1)/2 > 2) grandparent = (i-3)/4;

- Insert：
  1. Decide which level → min or max
  2. Check whether to swap with its parents.
     - NO：ReheapUp from the current node.
     - Yes：ReheapUp from its parents.
- Delete the smallest：
  1. Replace the root with the last element.
  2. Check whether to swap with its smaller child.
     - NO：ReheapDown from the root (recursion)
     - Yes：ReheapDown from the root (recursion)
- Delete the Largest：
  1. Replace the maximum with the last element.
  2. Check whether to swap with its larger child.
     - NO：ReheapDown from the current node.
     - Yes：ReheapDown from the current node.

□ Main Idea in Min-max Heap

- Three 4-way tree
  - max-heap + min-heap + max-heap
  - each node in max-heap has its parent in min-heap.

□ Double-ended Heap (DEAP)



- Insert:
  1. Examine the corresponding node: Left < Right
  2. ReheapUp if necessary (recursion)

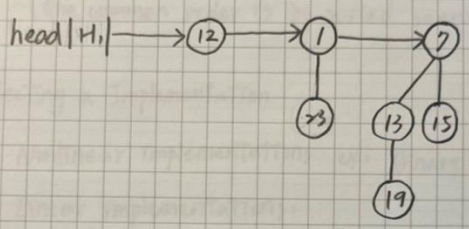
- Delete the smallest、largest:
  1. Replace the root of min/max-heap with the last element
  2. ReheapDown if necessary // 若走到樹葉，要檢查對應節點
  3. Examine the corresponding nodes: Left < Right

□ Main Idea in DEAP

- Two heaps
  - Pseudo root + min-heap + max-heap
  - Each node in max-heap corrosponds to one in min-heap

☐ Binomal Heap $O(\log n)$

- Definition: A binomial heap is a collection of binomial trees that satisfy the heap property and have distanct orders.
  (two binomial trees of the same order can be merge)



7筆資料
$7 = 2^0 + 2^1 + 2^2$

- Insert $O(\log n)$
  1. Insert into the linked list of the roots.
  2. call merge function.

- Delete $O(\log n)$
  1. Find the minimum from the linked list of the roots
  2. Delete the root having the minimum.
  3. Add its children into the linked list.
  4. Call merge function

☐ Binomial tree of order k (B_k)

- The root has k children
- Merged by two binomial trees of order k-1
- number of nodes = 2^k
- Tree height = k+1 → $O(\log n)$

單元三：由下而上成長的平衡樹

□ The ADT table

- Our table assumes distinct search keys.
  (Other tables could allow duplicate search keys.)
- The traverseTable operation visits table items in a specified order.
  (One common order is by sorted search key.)

□ Selecting a Implementation

- Nonlinear implementations ex: Binary search tree. ↓
  Insertion : O(logn)
  Deletion : O(logn)
  Retrieval : O(logn)
  Traversal : O(n)
- Linear implementations:

	array based	pointer based
Unsorted	Insertion : O(1) Deletion : O(n) Retrieval : O(n) Traversal : O(n)	Insertion : O(1) Deletion : O(n) Retrieval : O(n) Traversal : O(n)
Sorted	Insertion : O(n) Deletion : O(n) Retrieval : O(logn) Traversal : O(n)	Insertion : O(n) Deletion : O(n) Retrieval : O(n) Traversal : O(n)

- 比較
  - 線性：適用較少筆資料的情況.
  - 非線性：通常表現好的選擇 (ex: balanced binary search tree).

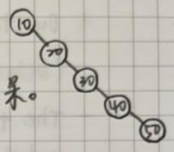
□ Binary Search Tree

- The efficiency tree is related to the tree height.

  Maximum : n

  Minimum : $\lceil \log_2(n+1) \rceil$

  原因：新增和刪除的順序會影響到結果。



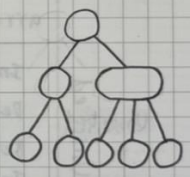
□ 2-3 Tree

- Are general trees, not binary trees.
- Are never taller than a minimum-height binary tree.

  (A 2-3 tree never has height greater than $\lceil \log_2(n+1) \rceil$)

  2-3樹最糟糕的情況，是二元樹最好的情況。

- Insert = 由下往上.



  1、先找到要插入資料的位置(樹葉)

  2、新增資料

  3、如果樹葉沒有滿，則直接加入

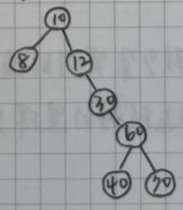
  4、如果樹葉有3筆資料，則分離成2個節點

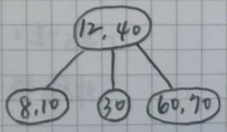
  case1 → 若內部節點，則繼續分離 (recursion)

  case2 → 若樹根，則分離成2節點(新樹根)，樹高增加

Practice: Answer (10, 12, 30, 8, 60, 40, 70)

Binary Search Tree v.s. 2-3 Tree





- Delete

一、先找到要刪除資料的位置(樹葉)

二、刪除資料

三、若此樹葉尚包含一筆資料，則完成.

四、若此樹葉內沒有資料了，則

(a). 重新分配：retain the tree structure

(b). 合併樹葉：its parent has one less child

二者取決於兄弟節點有幾個 key (幾筆資料)

Q: 遇到要刪除內部節點 → 和中序中的下一個樹葉節點交換
→ 要刪的節點換到樹葉

□ 2-3-4 Tree

- Are general trees, not binary trees.
- Are never taller than a 2-3 tree
- Insert：和2-3樹基本相同，搜尋中遇到某節點已含3筆資料，則先分離(用空間換時間→不用遞迴上去)
  - Splits occur only at the path from the root to a leaf.
  - No upward recursion is needed.
- Delete：和2-3樹基本相同，搜尋中遇到某節點只有1筆資料，則先合併。
  - Transformations occur only at the path from the root to a leaf.
  - No upward recursion is needed.

□ Search

- The efficiency depends on the tree height.
- All leaves are at the same level.

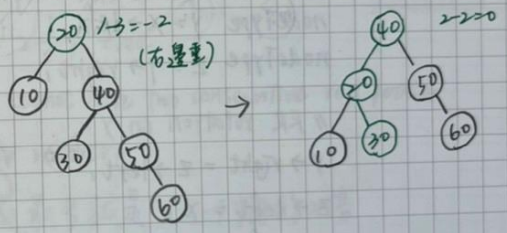
(2-3 tree、2-3-4 tree)

二元樹無法控制樹高.

單元四、由上而下成長的平衡樹

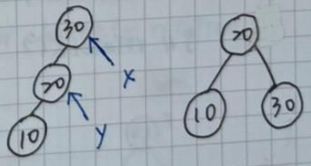
□ AVL tree

- A balanced search tree
- Matains the tree height close to the minimum.
- Main idea : After each insertion or deletion
  1. Check whether the tree is still balanced.
  2. If the tree is unbalanced, rotate to restore the balance
     - single rotation
     - double rotation



- LL Rotation : Pseudocode

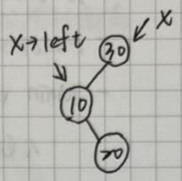
```
nodeType rotateLL (nodeType x) {
    nodeType y = x→left;
    x→left = y→right;
    y→right = x;
    return y;
} // rotateLL()
```



- LR Rotation : Pseudocode

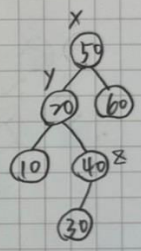
// first rotate left child with its right child : RR
// then, rotate node x with its new left child : LL

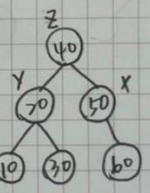
```
nodeType rotateLR( nodeType x ) {
    x→left = rotateRR ( x→ left ) ;
    return rotateLL (x) ;
} // rotateLR
```



→ Version I.

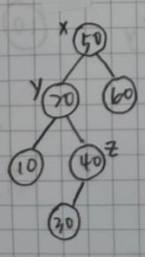
```
nodeType rotateLR( nodeType x) {
    nodeType y = x→ left ;
    nodeType z = y→ right ;

    // RR rotation on y
    y→ right = z→ left;
    z→ left = y ;
    x→ left = z ;

    // LL rotation on x
    x→ left = z→ right ;
    z→ right = x ;
    return z ;
} // rotateLR()
```

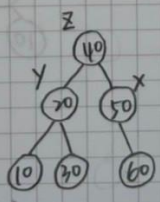




→ Version II.

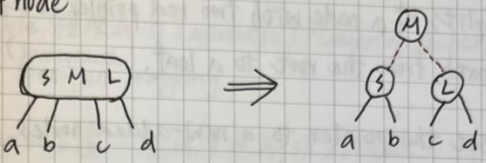
```
nodeType rotateLR2(nodeType x){
    nodeType y = x→ left ;
    nodeType z = y→ right ;
    y→ right = z→ left;
    x→ left = z→ right ;
    z→ right = x ;
    z→ left = y ;
    return z ;
} // rotateLR2()
```



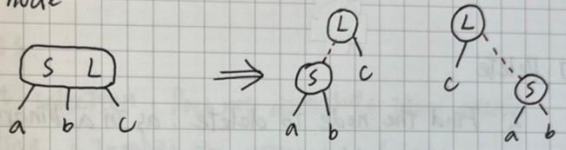


□ Red-black Tree

- 4 node



- 3 node



- Every external path has an equal number of black pointers.
  ⇒ Height of 2-3-4 tree
- External path cannot have two consecutive red pointer
  ⇒ Rotation to recover
- 紅黑樹樹高最多就是 2-3-4 樹的 2 倍
- 紅黑樹的搜尋有可能會比較慢，但新增刪除會比較快

□ Splitting a 4-node

1. parent is a 2-node : Only change the colors.
2. parent is a 3-node : Color change (+ rotation)

□ Insertion :

1. Splits of a node with two red pointer occur only on the path from the root to a leaf. (downward)
2. Set the pointer to a new-added nodes as red.
3. Rotate if there are two consecutive red pointers.

□ Delete

1. Find the node to delete, as in a binary search tree
   - two children → swap with the inorder successor
   - only one child → pointed to by a black pointer
   - Leaf → pointed by a red or black pointer.
2. Replace the node of only one child with its child
3. Delete the leaf if the pointer to it is red
4. Recolor or rotate → Leaf pointed by a black pointer
   black pointer → have a sibling

單元五、雜湊 Hashing

- Hash table 雜湊表 ≅ (automatic) Manger of lockers
  key → location mapping.
- Collision 碰撞 → 雜湊必須解決的問題
  - 不同的key對應到同一個location
- Functions.
  - Digit selection : Does not distribute item evenly.
  - Folding : Involves the entire search key.
  - Modulo arithmetic 取餘數: 用質數 prime 可避免太多 collision.
  - Converting character strings: 轉換成數字.
- Collision Resolution

  ∧ Open Address ☆hash table最好用質數!!!

  key independent
  - a、Linear probing: Search the first location and the next.
    - Promble: 櫥佔鵲巢, 且當資料黏在一起, 要搜尋很多筆數
    - Delete: 刪掉某筆資料, 下次搜尋過到空隔, 不代表不用繼續找
  - b、Quadratic probing: Search the first location and continue at the increments of $1^2, 2^2, 3^2$ (次方)
    - Problem: 空間問題 → overflow area (可能有空間, 但找不到)
    - 也有群聚問題, 但沒 Linear probing 嚴重

  key dependent
  - ∨、Double hashing: 用2個 hash function 來減少 collision.
    ($h_1 \neq h_2$ 且 $h_2(key) > 0$ → step size
    - table size 質數 v.s. step size 要互質, 才能用到每個空間

2、Restructuring the hash table：允許多個item存在同一格.

a. Buckets

- Each location in the hash table is array.
- 若還有碰撞，則前面3種方式擇一.

b. Separate chaining

- Each hash table location keep a linked list.
- The size of the hash table is dynamic.

c. Buckets + Separate chaining.

- Memory management（把空掉的空間拼起來）
- Worst case：O(n)

□ Inefficient Operations on Hashing. 不適合用雜湊

- Traversal：sorted order
- NN search：Find the smallest or largest item ex: Heap...
- Range Query：Find the item between two search key. ex: tree...

□ Application 1：Hash Join.

- Join is an important operation in database.
（在不同群的資料中，找查某筆特定資料）

□ Application 2：Count-Min Sketch

- 在資源有限的情況下，紀錄下某些資料.（有可能有一些錯誤）.
- 應用：Hot IPs / users / files / words（熱門→超過某些門檻）

Python 不專業筆記

一、字串 string　phrase = "Hello Mr.White"

- 相關函式
  - phrase.lower()　# 字串變小寫.
  - phrase.upper()　# 字串變大寫
  - phrase.Tslower()　# 判斷是否小寫
  - phrase.Tndex("H")　# H所在index
  - phrase.replace("H","h")　#交換
- 相關知識 - 可直接表示 phrase[0]　# 像陣列

二、讀入　name = Tnput("請輸入你的名字:")

輸出　print("哈囉"+name+"你今年"+age+"歲")

三、列表 list(陣列)

scores = [90, 70, 60, 50, 40]
friends = ["小黑", "小黃", "小綠"]
things = [90, "小黑", true]

- 相關函式
  - scores.extend(friends)　# scores + friends.
  - scores.append(30)　# scores 加一筆 30.
  - scores.Tnsert(2,50)　#在第二格插入50
  - scores.remove(90)　# 刪除 90.
  - scores.clear()　# 清空 scores
  - scores.pop　# 移除最後一位.
  - scores.sort()　# 排列.
  - scores.reverse()　# 反轉
  - scores.count(60)　# 有幾筆 60.

四、元組 tuple　scores = (90, 80, 70, 60, 50)

・相關知識：不可更改裡面的值，不可新增、刪除。

五、函式 function

```
def hello(name, age)
    print("hello" + name + "你今年" + str(age) + "歲")

hello("小白", 87)    #呼叫函式
```

六、if 判斷句

```
if score == 100:
    print("超棒")
elif score >= 60:
    print("普通")
else:
    print("不及格")
```

七、字典 dictionary

```
dic = {0:"apple", 1:"banana", 2:"cat", 3:"dog"}
print(dic[3])        // output: dog.
```

八、for 迴圈

```
for letter in "小白你好"
    print(letter)    // output: 小白你好
```

九、檔案讀取、寫入

```
open("檔案路徑", mode="開啟模式")
#絕對路徑
#相對路徑  以程式的位置做延伸

# mode="r"  讀取
# mode="w"  覆寫
# mode="a"  在原先的資料後寫東西.
file = open("123.txt", mode="r")
print(file.read())
file.close()
```

十、類別 class & 物件 object

```
class Phone:
    def __init__(self,os, number, waterproof):
        self.os = os
        self.number = number.
        self.waterproof = waterproof.

phone1 = phone("ios", 123, True)
```

十一、繼承 inheritance

```
from person import Person   #先引入模組
class student(Person):      #寫法.
    def __init__(self, name, age, school):
        :

    def print_school(self):
        :
```