單元一 優先佇列

- Basics of Priority Queues

⇒ Priority Queue = Search key + priority value

① Use Selection Sort : (刪除所需時間較長)

pq Insert() : O(1) - 直接放在最後面

↓

Selection Sort = Unsorted List

↓

pq Delete() : O(n) - 每次都找最小的時間需較長.

② Use Insertion Sort : (插入陣列所需時間較長)

pq Insert() : O(n) - 需尋找最佳位子插入

↓

Insertion Sort = Sorted List

↓

pq Delete() : O(1) - 資料已排序好優先度最高在第一個

③ Use Binary Search Tree : (插入&刪除時間都不太差)

pq Insert() : O(logn) (最差情況 O(n))

↓

Tree Sort

↓

pq Delete() : O(logn) (最差情況需 O(n))

- Application

鄰近點搜尋 (不需排序)

① 先分區, 運用 priority queue 尋找最近區

② 在最近區內再運用 priority queue 尋找最近點

• Heap – 不排序，只找出最大 or 最小 ⇒ 效率較好

① min-heap

pq.Insert(): O(logn)

↓

min-heap

↓

pq.Delete(): O(1) – 最小值在樹根

② max-heap

pq.Insert(): O(logn)

↓

max-heap

↓

pq.Delete(): O(1) – 最大值在樹根

③ 定義

1. 是一棵完整樹

2. 從樹葉往上走的每一條路都可以看成一個 sorted list.

④ 新增資料：加在 bottom ⇒ ReheapUp

⇒ 新增於 bottom 的下一個，由下往上檢查 (ReheapUp)

⑤ 刪除資料：刪 root ⇒ ReheapDown

⇒ 把 bottom copy 放在 root，刪除 bottom，由上往下檢查 (ReheapDown)

跟子節點中較大 (小) 的交換

⑥ 應用於霍夫曼編碼

⇒ 用 min-heap 做 2 次 Delete 產生兩個最小的

△ 任一個 code 不會是另一個 code 的字首 (ex. 有 01 就不會有 010 or 011...)

```
→ int size;
→ HeapItemType items[MAX-HEAP];

→ heapInsert(){
    if(size >= MAX-HEAP) throw HeapException("Heap full");
    items[size] = newItem; // 將資料放在最後
    int place = size;
    int parent = (place-1)/2; // 他爸
    while((parent >= 0) && (items[place] > items[parent])){
    // reheapUp!
    HeapItemType temp = items[parent]; // swap.
    items[parent] = items[place];
    items[place] = temp;
    place = parent;
    parent = (place-1)/2;
    } // while

    ++size;
  } // heapInsert()

→ heapDelete(){
    items[0] = items[size-1]; // 把 bottom copy 到 root
    --size; // 刪除
    heapRebuild(0);
  }
```

→ heapRebuild (int root) { // 把 semi-heap 變成 heap.
↳ 只有root是錯的

```
int child = 2 * root + 1 ; // 左小孩
if ( child < size ) {
    Tnt  rightChild = child + 1 ; // 右小孩
    if ( rightChild < size && items[rightChild] > items[child])
        child = rightChild // 比較大的小孩
    if ( items [ root] < items[child] ) {
        HeapItemType  temp = items [root] ; // swap
        items [ root ] = items [ child ] ;
        items [child] = temp ;
        heapRebuild ( child) ; // 繼續往下
    } // end if
} // end if
} // heapRebuild ()
```

- 把完整樹直接轉成 max-heap

⇒ 重複 call heapRebuild(i), i = n (↱最後一個節點) ... 2, 1, 0

⇒ why 從下往上呼叫?

先把子樹都變成 heap ⇒ 變一個 semi-heap.

⇒ why not 呼叫 0, 1, 2 ... ?

若最大值不在 root 的小孩中就會錯

- Heap Sort

① 建 heap

② 印出樹根, delete 樹根

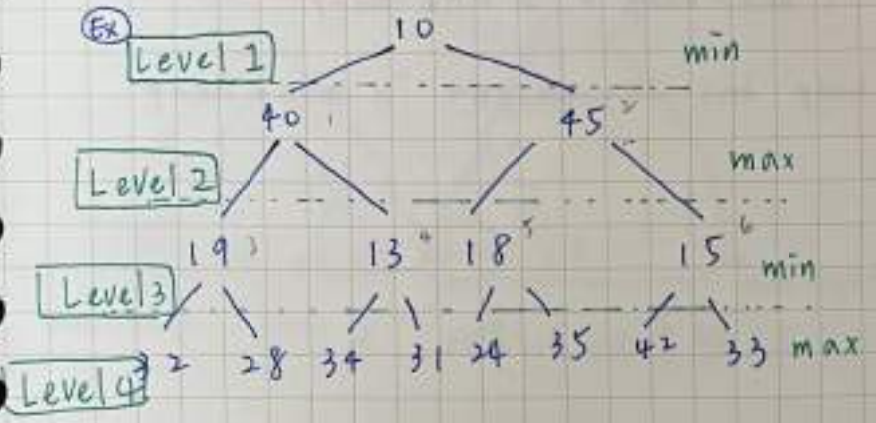
③ heapRebuild, 重新 step ①

④

單元二 堆積變形

• Min-Max Heap

Ex



• 奇數層為min，偶為max.

1. Insert

① 計算插入的節點位於第幾層

② 檢查是否要和父節點交換值

- 否：ReheapUp from the current node
- 是：ReheapUp from its parent (跟祖父節點)

2. Delete

ex 刪最小

① 把最尾端資料放到root

② 確認要不要跟"較小"的子節點交換 (root在min層→要比小孩小)

- No：ReheapDown from the root (看所有的min層)
- Yes：ReheapDown from the root (看min層！)
  - ↳ 如果交換後的值是min層的最下層(樹葉)要再多檢查是否比下面的max層小！！

ex 刪最大

① 將最大值和尾端節點交換並刪除最大

② 檢查是否要和"較大"的子節點交換(最大值在max層→比小孩大)

- No：ReheapDown from the current node (max層)
- Yes：ReheapDown from the current node (max)

3. level = $\log_2(size+1)$ →再取除2的餘數判斷 min or max

4. grandparent = $(i-3)/4$ parent = $(i-1)/2$ ⇒ grandparent = $\left[\frac{(i-1)}{2}-1\right]/2$ $= (i-3)/4$

grandchildren = $(i\times4+j)$，$j=3,4,5,6$

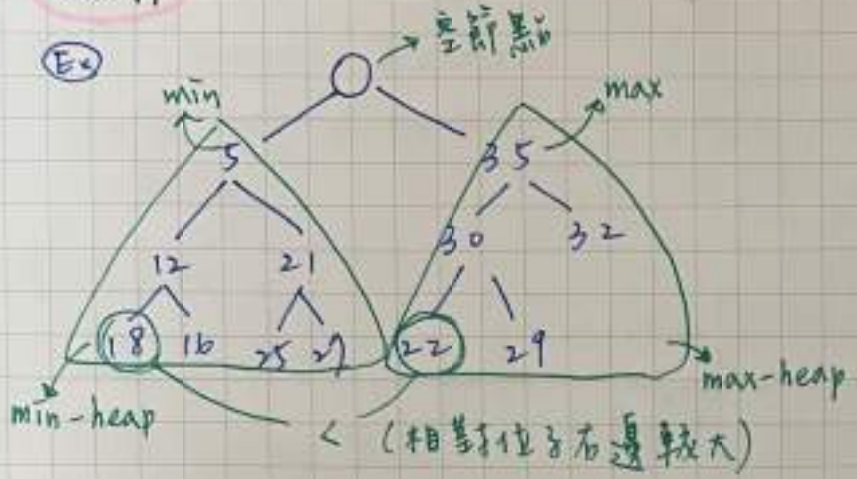
5. 總結

① Min-Max Heap = 1 min-heap + 2 max-heap.

② each node in max-heap has its parent in min-heap.

· DEAP

Ex



1. Insert

① 放入陣列尾端，比較相對位置（Left < Right）※若相對位子沒東西 ⇒看父節點.

② ReheapUp if 交換位子

2. Delete

ex 刪最小

① 將最小堆積的根與底部節點交換 ←左樹

② ReheapDown

③ 看需不需要和另一棵樹的對應節點交換（Left < Right）

ex 刪最大

① 根 ⇄ 底部節點

② ReheapDown

③ 校查左右關係 及其子節點們※

3. 總結

① DEAP = 空的root + min-heap + max-heap

② 左右對應關係（左 < 右）

單元三 由下而上成長的平衡二元樹

- 搜尋 (Search、Retrive)

  search key + Record
  (關鍵字)

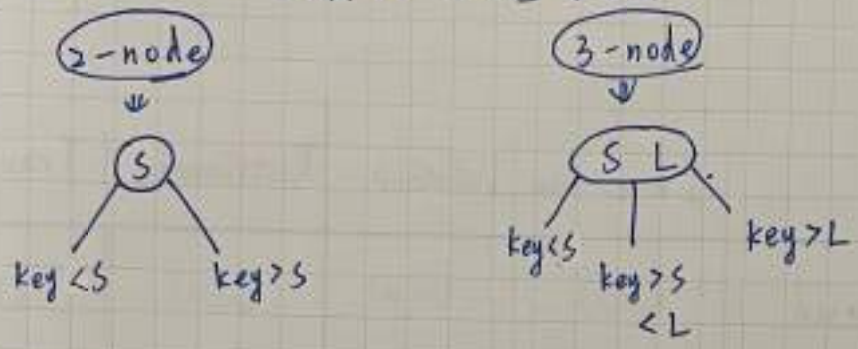
- 實作方式

	Insertion	Deletion	Retrieval	Traversal
Unsorted array based	O(1)	O(n)	O(n)	O(n)
Unsorted pointer based	O(1)	O(n)	O(n)	O(n)
Sorted array based	O(n)	O(n)	O(logn)	O(n)
Sorted ptr based	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(logn)	O(logn)	O(logn)	O(n)

- 平衡二元樹
  - → 2-3 Tree
  - → 2-3-4 Tree
  - → AVL Tree
  - → Red-Black Tree

• 2-3 Tree (優：樹高矮，平衡)

{ 2 nodes ⇒ 1筆資料(key) 2個小孩
{ 3 nodes ⇒ 2筆資料 3個小孩



1. Insert

① 找到存放位置的樹葉節點

② 若資料筆數=3 就要分裂 ⇒ (1 2 3) ⇒ 

③ 若要分裂的節點是 root ⇒ 樹高+1

2. Delete

① 找到要刪的資料刪除

② 如果刪除的節點還有一個資料就結束

③ 若刪除的變成空節點(leaf):

case 1: 若最近的兄弟是3node (有2個資料)

⇒ 把兄弟的資料拆一個過來

case 2: 最近的 sibling 是 2-node

⇒ 做合併 (把爸爸拆給兄弟)

再呼叫刪除父節點 (recursion)

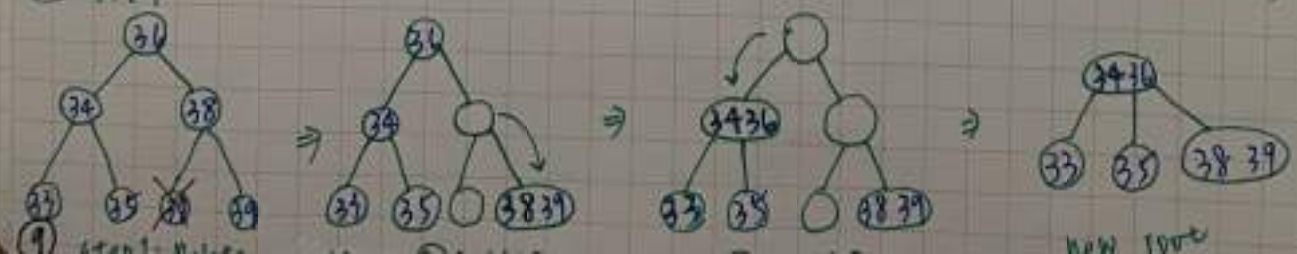
case 3: 若刪的是 root

⇒ 指向新 root

★④ 若刪除的是中間節點

⇒ 先和中序的後繼者交換位子 (右子樹的最左小孩)

ex 刪37.



3. Delete 演算法

```
deleteItem( in ttTree , in theKey)
    x = the tree node whose search key equals theKey
    if( x is not a leaf)
        y = Successor (x); ⇒ 中序的後繼者
        swapkey ( x , y);
        x = y;
    Delete theKey from x;
    if( x now has no item)
        fix(x);


fix( in x)
    if (x == root)
        remove the root ;
    else
        p = parent of x ;                    3node
        if( the nearest sibling of x has two items)
            Redistribute items among the sibling , p and x ;
                                              (rotate)
            if( x is not a leaf)
                Move appropriate child from sibling to x; (最大者的
        else // merge
            s = the nearest sibling of x ;
            Move appropriate item down from p to s ;
            if (x is not a leaf) Move x's child to s ;
            remove x;
            if( p now has no item) fix(p);
```

• 2-3-4 Tree (比2-3 Tree更精)

- 2-node : one data two child
- 3-node : two data three child
- 4-node : three data four child

1. Insert (和2-3 tree不同的地方):

① Insert前先檢查該節點是否為4-node

② 先split再Insert (路上遇到的4-node都要split)

2. Delete

① 在路上遇到的2-node都做合併(∵可能造成空節點)

3. 缺點:

用空間換時間 (用到比BST更多空間)

單元四 由上而下成長的平衡樹

- 定義：平衡，高度最矮的二元樹
  左右樹高度相差 <= 1

- AVL Tree：

1. Insert：

① 直接 Insert（同 BST 的 Insert）

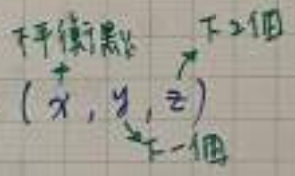
② 確認是否平衡：

Balance Factor (BF) - 平衡係數

= h(left subtree) - h(right subtree)

③ 不平衡才做 rotation

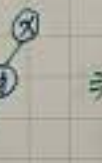
從不平衡點往新增的節點找下兩個（x, y, z）



type 1：

⇒ RR

type 2：



⇒ LL

} single rotation

type 3：

⇒ RL

type 4：

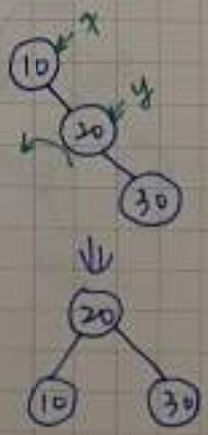
⇒ LR

} double rotation

2. single rotation

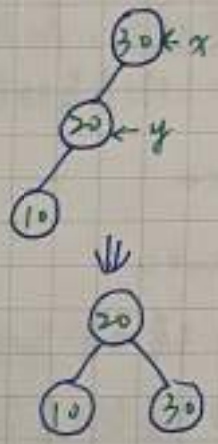
① RR

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



② LL

```
nodeType LL(nodeType x){
    nodeType y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}
```



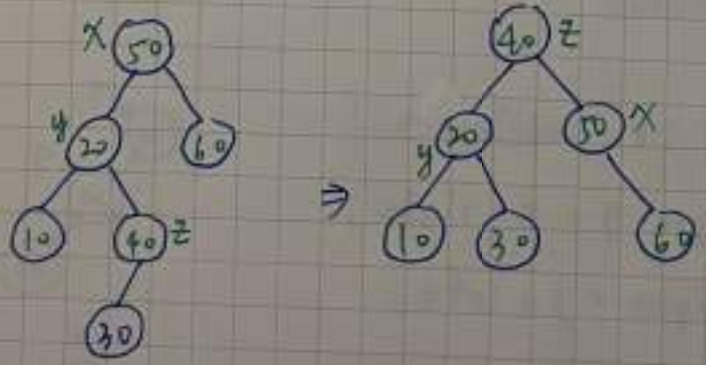
3. double rotation

① LR

```
nodeType LR(nodeType x){
    nodeType y = x->left;
    nodeType z = y->right;
    y->right = z->left;
    z->left = y;
    x->left = z;
    x->left = z->right;
    z->right = x;
    return z;
}
```

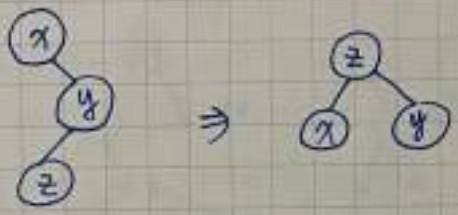
y 기준 RR

x 기준 LL



② RL

```
nodeType RL ( nodeType x) {
    nodeType y = x→right;
    nodeType z = y→left;
    y→left = z→right;        } 等效 LL
    z→right = y;
    x→right = z;
    x→right = z→left;        } x 做 RR
    z→left = x;
    return z;
}
```



4. Delete

① 若不是 leaf ⇒ 跟中序的後繼者交換

② 刪除

③ 檢查是否平衡 (同 Insert)

{ 是 ⇒ 結束
  否 ⇒ rotate.

5. 用 3 比 2-3 Tree 少空間 (一個節點只有一個東西)

· 紅黑樹 Red-black tree

1. 定義：

① 節點：2-3-4 tree

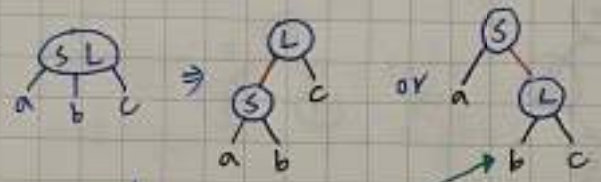
② AVL tree 的旋轉功能 ⇒ 平衡

③ 優：有 2-3-4 tree 的優點，用較少空間

④ 2 種顏色

(1) 4-node ⇒ 2 red

(2) 3-node ⇒ 1 red, 1 black



(3) 2-node ⇒ 2 black (root)

⑤ 從樹根走到任一個 leaf 經過的 black 數量一樣

⑥ 紅色不能連續出現 ⇒ rotate

⑦ 樹高最高是 2-3-4 Tree 的 2 倍.

⇒ 搜尋較慢

2. Split 4 node

① Parent is a 2-node ⇒ 改顏色

1. to child : red → black
2. from parent : black → red

② Parent is a 3-node ⇒ 改顏色

1. to child : red → black
2. from parent : black → red
3. 若 parent → grandparent 也是 red (3-node) ⇒ rotate (不能重複)

3. Insert

① 在路上遇到 2 red pointers (4-node) 就 split (同2-3-4 Tree)

② 新節點的 pointer is red

③ 若有連續的 red ⇒ 旋轉

4. Delete

① 找到刪除的節點

- two child ⇒ 和 in-order successor 交換
- one child ⇒ 把 child 接上來
- leaf ⇒
  - pointed to by red ⇒ 直接刪
  - pointed to by black ⇒ 一定有兄弟

② grandparend → parent is red & pointed to sibling is black

(a) 兄弟是 leaf ⇒ 把 black → red

(b) 只有 left child ⇒ (刪的是右小孩)
- LL + red → black (point to child)
- RR (If delete left child)

(c) 只有 right child ⇒
- LR + red → black (pointed to child)
- RL (If delete left child)

(d) 2 child ⇒ LL + red → black (pointed to left child)
- RR (If delete left child)

③ pointer to sibling is red

case 1: 刪的是右小孩 (sibling 一定有 2 個黑小孩)

(a) LL rotation + recolor

(b) sibling 的右小孩
- no child ⇒ 結束
- left child only ⇒ LL
- right child only ⇒ LR
- two child ⇒ LL + recolor

case 2: 刪左小孩

(和 case 1 相反) (L → R, R → L)

④ pointers to parents and sibling are black

case 1：刪右小孩（sibling 的小孩一定是紅色）

(a) no child ⇒ 把指向 sibling 改成紅色

⇒ 往上 recursion

(b) right child only：LR + recolor

(c) other：LL + recolor

case 2：刪左小孩

（和 case1 相反）（L→R，R→L）