01-01

Ch 1 : Priority Queue

Outline : Basic of Priority Queue

Heap ;

Heap Sort ;

Min - Max Heap ;

Basic of Priority Queue :

Ex : PQ (P1 , 5 , 23:55)
(P2 , 5 , 00:05)
(P3 , 3 , 00:10)
(P4 , 4 , 00:30)
↑ 病人 Level

PQ (5), (5), (3), (4)
↳ PQ (5), (5), (4)
↳ PQ (5), (5)
↳ PQ (5)
一個時間取一資料.

01-02

* unsorted List {Sorting Algorithm}

{ pqInsert (dataItem , priority)
pqDelete() , pall()

Selection Sort (找最小) < O(1) : 維護 unsorted 資料
O(n) : 插入時直接插最後

* sorted List

Insertion < O(n)
O(1)
↑
資料已排序
最大最小抓一個

* BST

Tree Sort < O(nlogn)
O(klogn)

Heap

01-04

1. Complete binary tree

Input

↓ * pqInsert : O(log n)

min - heap

max - heap

↓ * pqDelete : O(log n) . 刪除以後要找 root

Output

2. The value sorted at a node is greater (smaller) or equal to the values stored at the childderen.

01-05

```
struct HeapType {
    void ReheapDown (int , int ) ;
    void ReheapUp (int , int ) ;
    ItemType *elements ;
    int numElements ;
}
```

1. Insert :

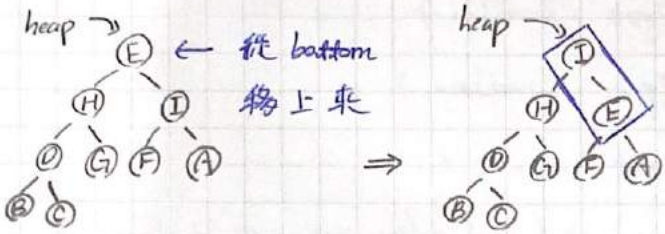
① Insert a new element in the next bottom rightmost place.

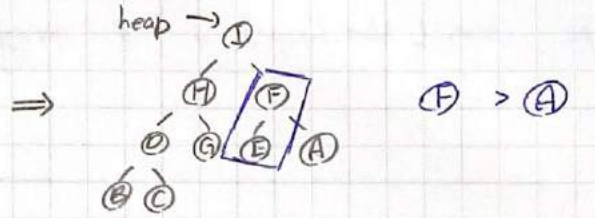
② Fix the Heap property by calling ReheapUp.

ReheapDown function

01-06

Assumption : heap property is violated at the root of the tree.





Delete

① Copy the bottom rightmost element to the root

② Delete the bottom rightmost node

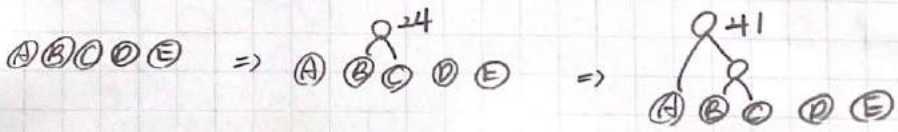
③ Calling ReheapDown to Fix Heap.

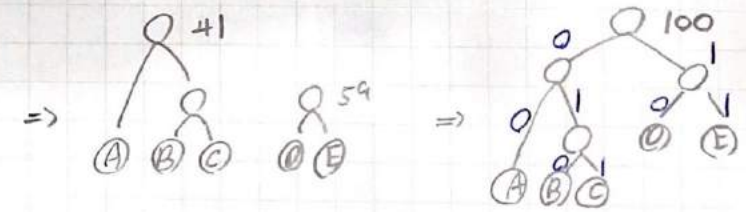
Huffman Coding

01-07

① 從樹葉往上建，樹最最小

ex. Ⓐ 17 , Ⓑ 12 , Ⓒ 12 , Ⓓ 27 , Ⓔ 32





二元樹

A : 00

B : 010

C : 011

D : 10

E : 11

Heap Operation

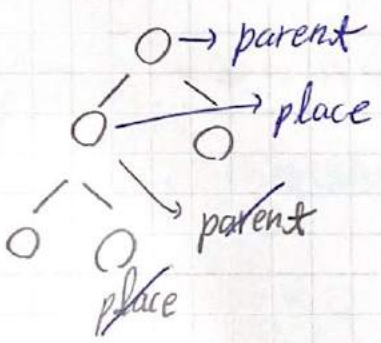
01-08

```
bool isEmpty () ;
void heapInsert ( heapItemType &newItem ) ;
void heapDelete ( heapItemType &rootItem ) ;
void heapRebuild ( int root ) ;
    // Converts the semi-heap rooted at index root into a heap.
HeapItemType items [Max_heap] ; // array.
int size.
```

```
void heapInsert ( heapItemType &newItem ) {
{ items [size] = newItem ; // put new Item at the end
  int place = size ;
  int parent = (place - 1) / 2 ; // trickle new item up.
  while ((parent >= 0) && (items[place] > items[parent])) {
      swap (items[place], items[parent]) ;
      place = parent ;
      parent = (place - 1) / 2 ;
  } // end while

  size ++ ;
} // end Insert
```

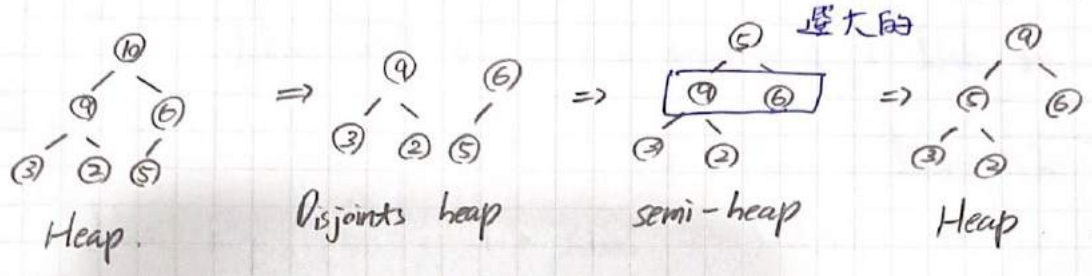
Step 1 Insert

Step 2 trickle up



Delete

01-09



```
void HeapDelete ( heapItemType & rootItem ) {

    if ( ! heapIsEmpty() ) {
        swap ( items[0] , items[size-1] ) ;
        delete items[size-1] ;

        heapRebuild (0) ;   // trickle it down to its
    } // end if               // proper position
} // end Delete
```

⇓

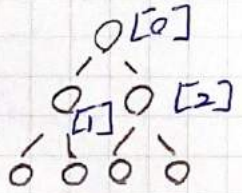
01-09

```
void heapRebuild ( int root ) {

    int child = 2 * root + 1 ;  // left child

    if ( child < size ) {
        int rightchild = child + 1 ;  // right child.

        if ((rightchild < size) && (items[rightchild] > items[child]))
                                                    max-heap.
            child = rightchild ;  // choose the largest child.
                                  max-heap.
        if ( items[root] < items[child] ) {

            swap ( items[root], items[child] ) ;
            heapRebuild [ child ) ;
        } // end if
    } // end outter if
} // end Rebuild.
```

Make Heap

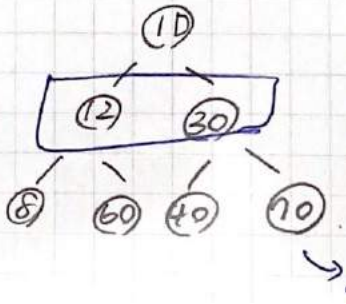
01-10

Call heapRebuild (i) , i = 2, 1, 0



question : Is it correct, if i = 0, 1, 2.

(Wrong)



如果 i 從 0 start.

下一層沒有最大的

就會錯.

HeapSort

01-12

* $O(n \log n)$

  { n = 取 bottom 補在 root
  { log n ; 每次呼叫 Reheap 就是 O(log n)

```
void HeapSort ( int size ) {
    int index ;
    for ( index = ( size -1 ) / 2 ; index >= 0 ; index -- )
        ReheapDown ( index , size ) ;
    // Convert array into heap

Sort:
    for ( index = size -1 ; index >= 1 ; index -- ) {
        swap ( items[0] , items [ index ] ) ; // root 和右下角
        ReheapDown ( 0 , index ) ;            // 互換.
    } // end for
} // end Sort
```

Ch 2 : Variations of Heap

02-01

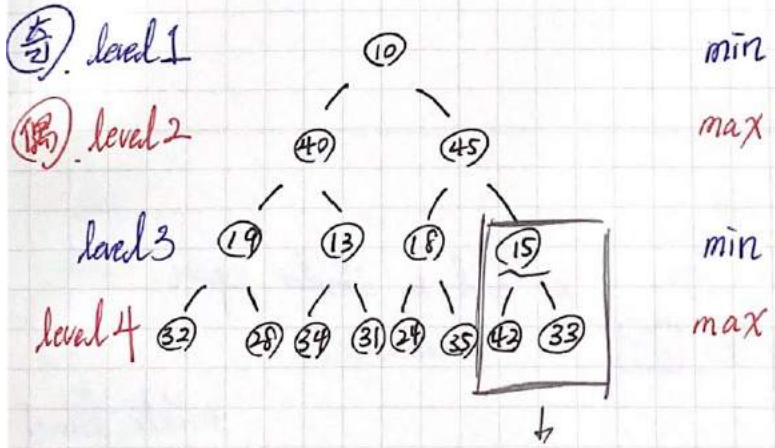
Outline

① Double-ended Priority Queues (DEPQ)
   (i) Min-Max Heap
   (ii) Double-ended Heap (DEAP)

② Forest (union) of Heaps
   (i) Binomial Heap
   (ii) Fibonacci Heap

Min-Max Heap

* Insert one key
* Delete the largest / smallest key



root 15 must be the smallest.

* 可以拆成 2 個 heap.

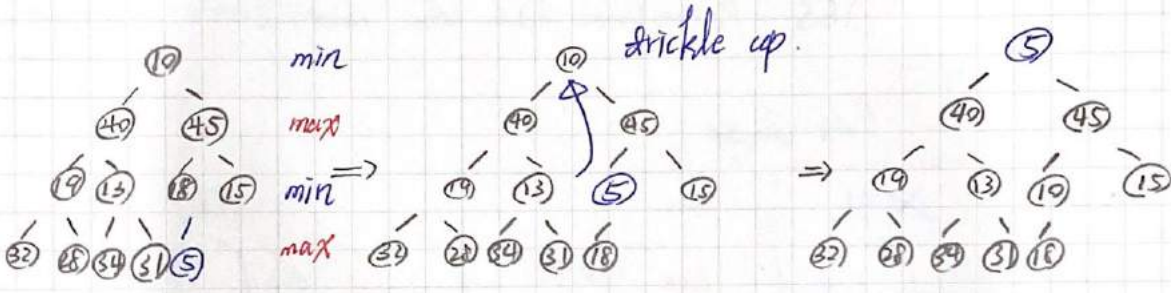
Insert

* Step 1 : Decide which level ⇒ min or max

Step 2 : Check whether to swap with its parent

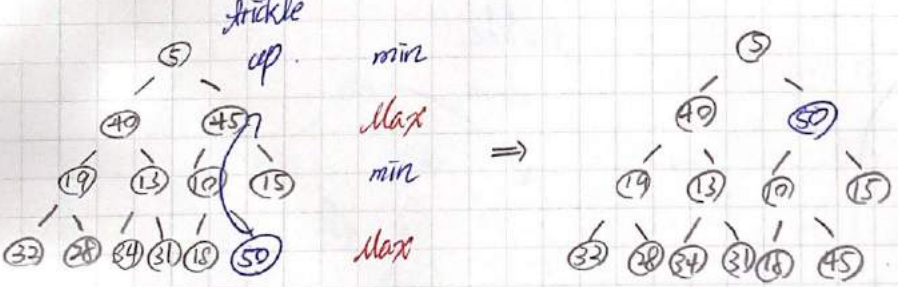
{ NO : Reheap from the current node
{ Yes : Reheap from its parent

Ex : YES



* 父節點是 "min" ⇒ swap ⑤, ⑱

Ex : NO



* 跟 parent [換 / 不換]，再跟同層交換

Delete

02-04

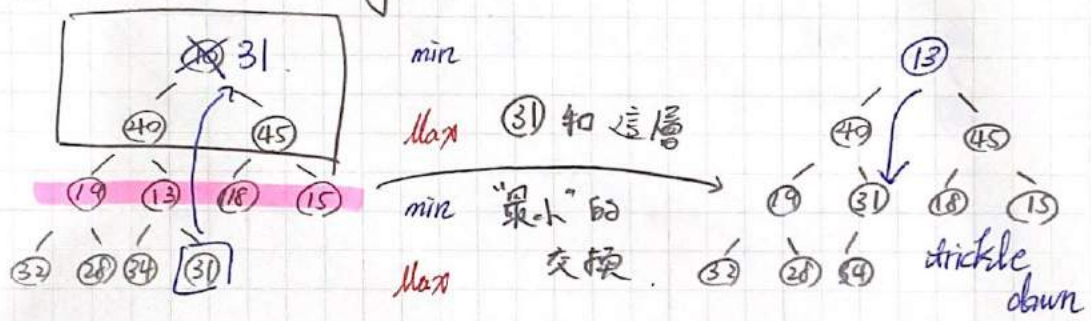
* Step 1 : Replace the root with the last element.

Step 2 . Check whether to swap with its smaller child.

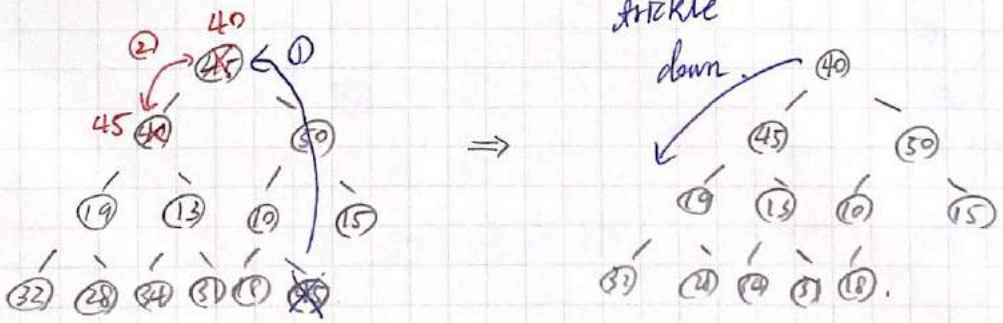
(第二層已經最大，但有可能搬上來的數 > 40，< 45，所以會跟第二層比較)

{ NO = ReheapDown from the root (recursion)
YES = ReheapDown from the root (recursion)

Ex 1 = Don't Change.



Ex 2 =



Decide which Level ?

02-05

- Level = (int) floor (log2(i+1)) % 2 ? Max ; Min ;

  ex 10 => log2(11) = 3

  3 % 2 = 1 => 奇

- grandparent = (i - 3) / 4 ;

  // if (i > 3) , 0,1,2 沒有 grandparent

- grandchildren = items [i * 4 + j] for j = 3, 4, 5, 6 .

Double - ended Heap (DEAP)

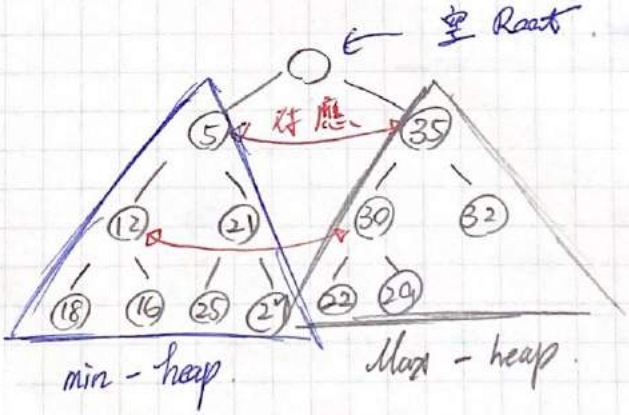
02-07

* Doubled - ended Priority Queue (DEPQ)

  Insert any key.

  Delete the smallest key

  Delete the largest key



DEAP · Insert

Step 1 · Examine the corresponding nodes · Left < right

Step 2 · ReheapUp if necessary (recursion)

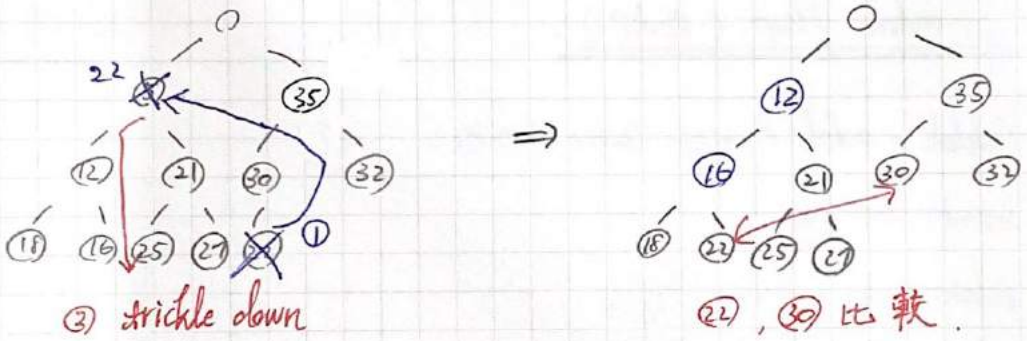
DEAP · Delete Smallest

02-08

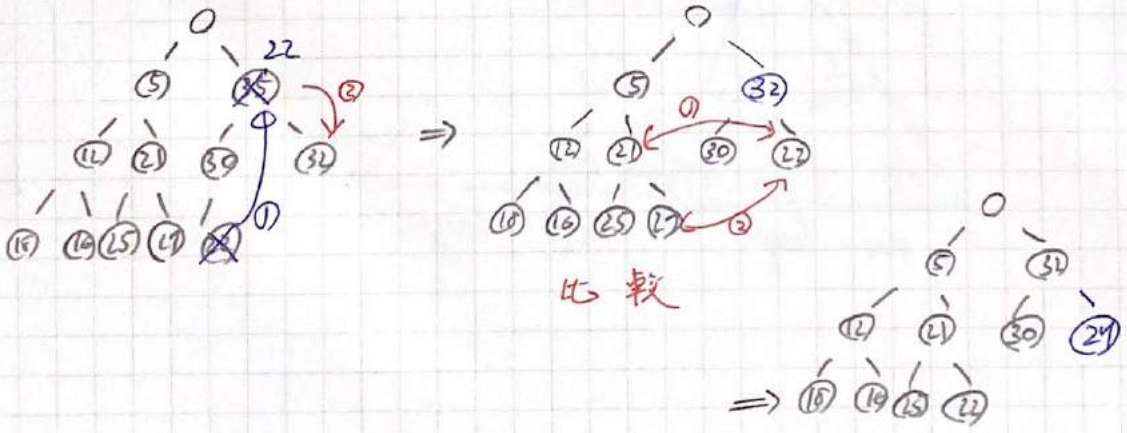
Step : 1. Replace the root of min-heap with the last element

2. ReheapDown if necesarry

3. Examine the corresponding nodes · Left < Right



DEAP · Delete largest



Application of DEPQ.

02-11

① External Sort

② quick sort + heapsort

③ large amount of data.

Application : Mergeable Priority Queues

① two cooks → one cook

② multiple servers (load balance) 負載平衡.

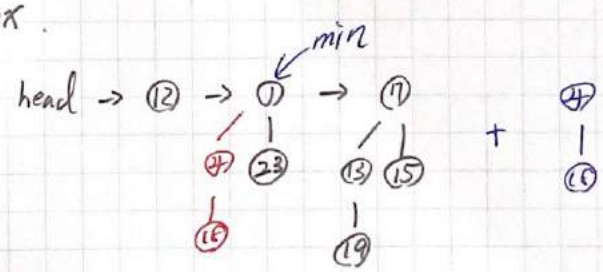
Binomial Heap.

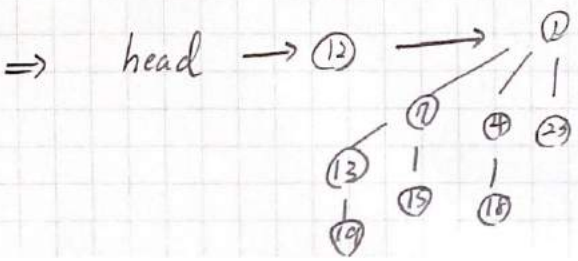
02-12

* Defination . A binomial heap is a collection of binomial trees that satisfy the heap property and have distinct orders.

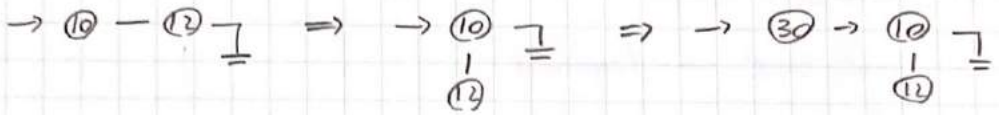
* 2 binomial trees of the same order can be merged.

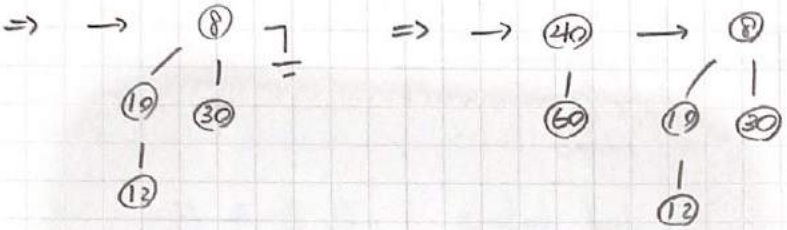
ex.

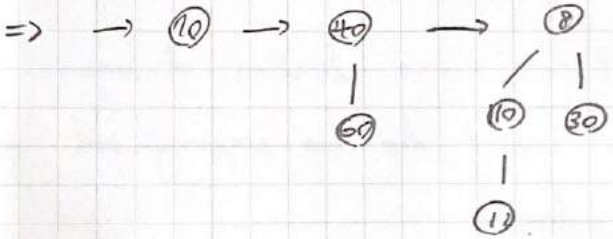




Ex. Input order 10, 12, 30, 8, 60, 40, 70







Ch3 · Binary Search Tree

The ADT table

03-01

- uses a search key to identify its items
- Its items are records that contain several pieces of data.

Selecting an Implementation

03-03

- Linear
  - Unsorted
  - Sorted.
- Non Linear

	Array - Based	Pointer - Based
Unsorted	Insertion : O(1) Deletion : O(n) (requires shifting data) Retrieval : O(n)	* No data shift Insertion : O(1) Deletion : O(n) Retrieval : O(n)
Sorted.	(shifting data) Insertion : O(n) Deletion : O(n) Retrival (Binary Tree) : O(log n)	* No data shift Insertion : Deletion : O(n). Retrival :

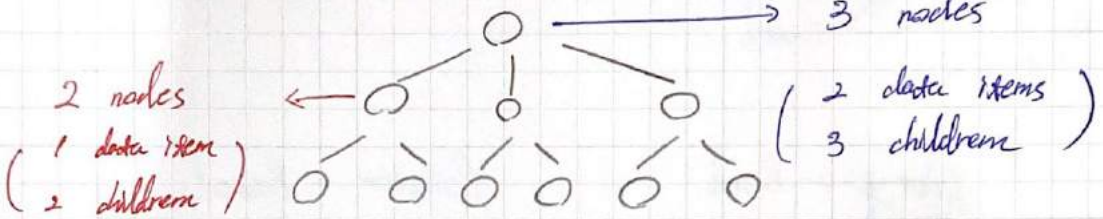
Balanced BST

03-04

- 2-3 tree, 2-3-4 tree
- AVL tree, Red-black tree (Always balanced).

2-3 tree

* Have 2 nodes and 3 nodes.



* general tree, not binary tree.
* Never taller than a minimum-height binary tree.

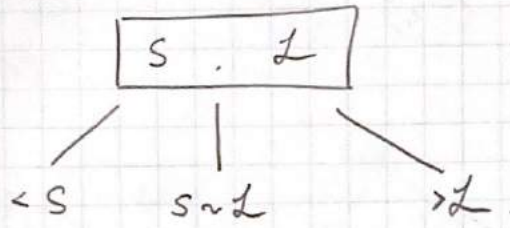
  height $\leq \log_2(n+1)$.

* BST : 由上往下長
  2-3 tree : 由下往上長.

Placing Data item in a 2-3 tree

03-05

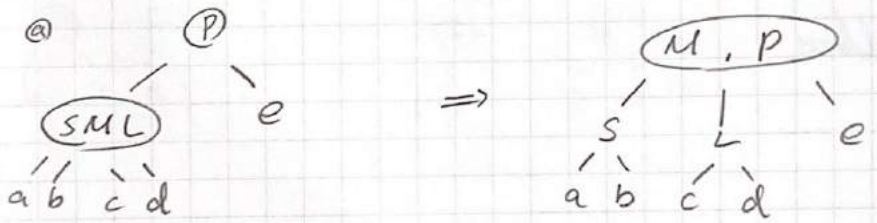
3-nodes (2 keys)



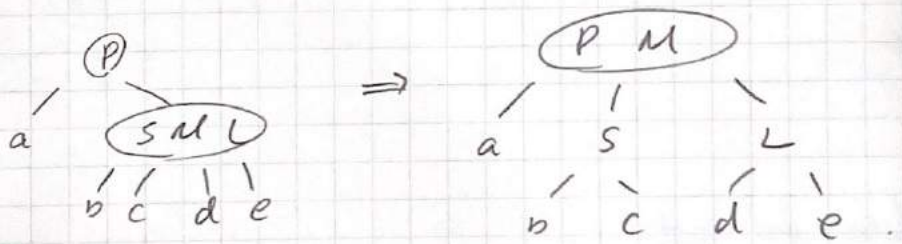
Insert O(log n)

03-06

ⓐ

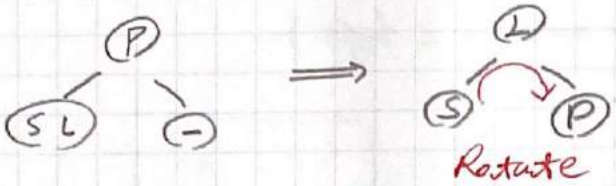


ⓑ

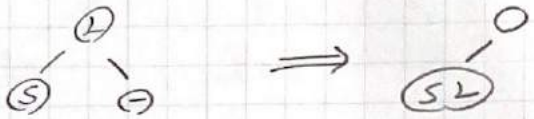


Deleting

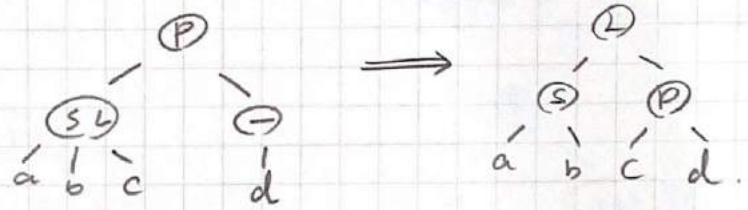
① Redistribute values



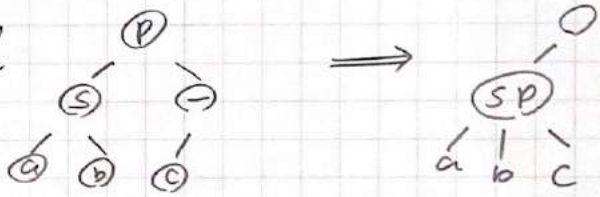
② Merge into Leaf



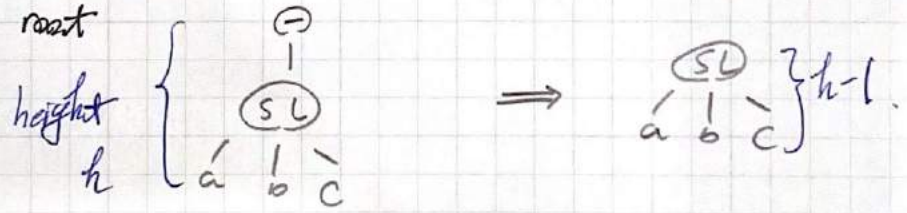
③ Redistribute values and children



④ Merge into an internal node.



⑤ delete the root

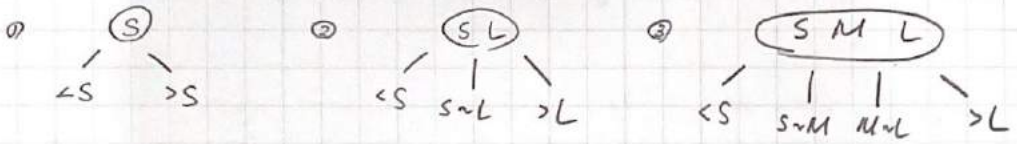


---

* 2-3 tree 總結
  - Searching · 不一定比 BST 快. 1 node 有 2 資料
  - Height
  - Maintaing the balance

2 - 3 - 4 tree

* 3 types of node

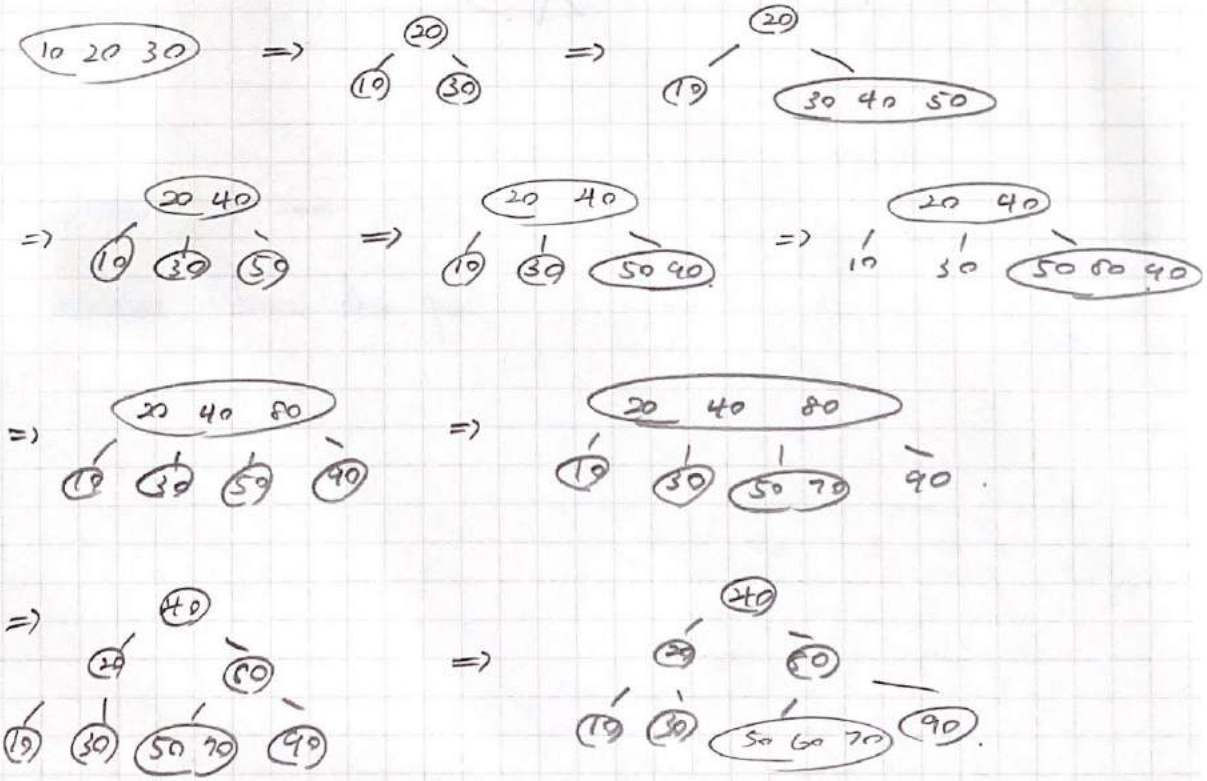


- Are general tree, not binary tree
- Are never taller than 2-3 tree.

Insert

* Splits occur only at the path from the root to a leaf (downward)!
* No upward recursion is needed.

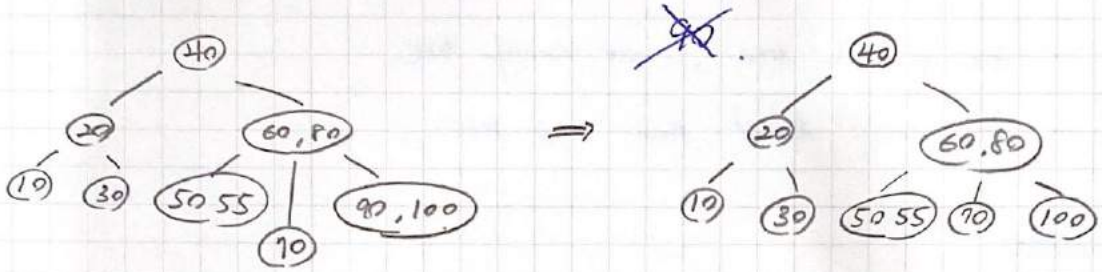
ex. Input = 10, 20, 30, 40, 50, 90, 80, 70, 60, 100

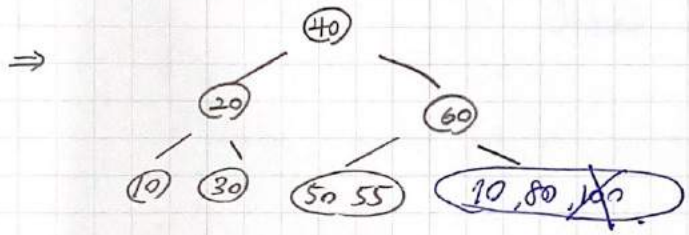


Delete

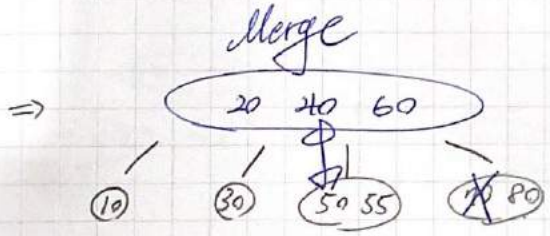
* Ensure that theItem does not occur in a 2 node. Transform each 2 node encountered into a 3 node or 4 node.

ex. Delete 90, 100, 70, 40

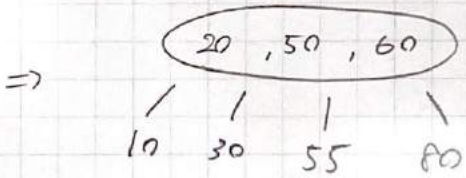




Delete from a leaf = Merge (2 nodes on the path)



Delete from a nonleaf = swap with inorder successor.



總結

* Insertion, deletion, algorithms - 234 tree requires fewer steps than 23 tree.

* 234 tree is always balanced

* 234 tree requires more storage than 23 tree

AVL tree

04-01

* A balanced BST

* Can be searched almost as efficiently as a minimum-height BST.

* Main idea:

  After each insertion or deletion. Check whether the tree is balanced.

  ⇒ Balanced Factor (BF)

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

  unbalanced → rotate to restore the balance.

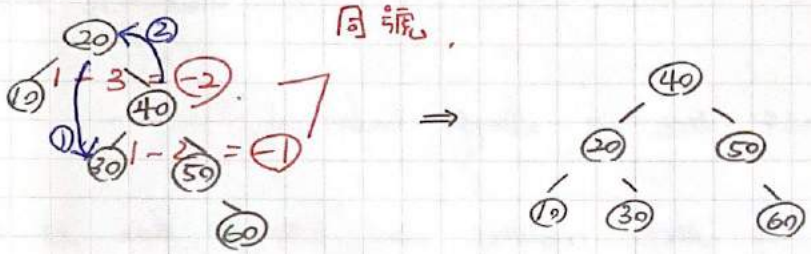
  ⇒ the left and right subtree's height differ by no more than 1.

* 把 Height 記在節點.

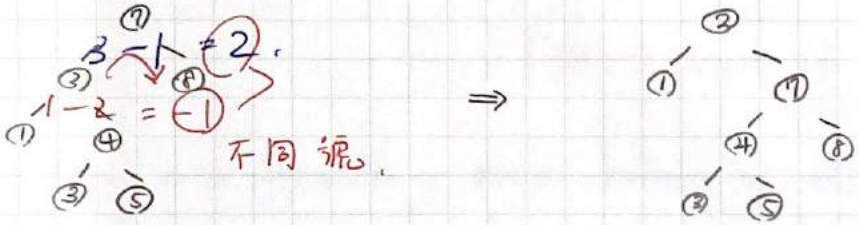
04-02

AVL Tree · Actions

① Single Rotation :



② Double Rotation :



Insert C)

04-03

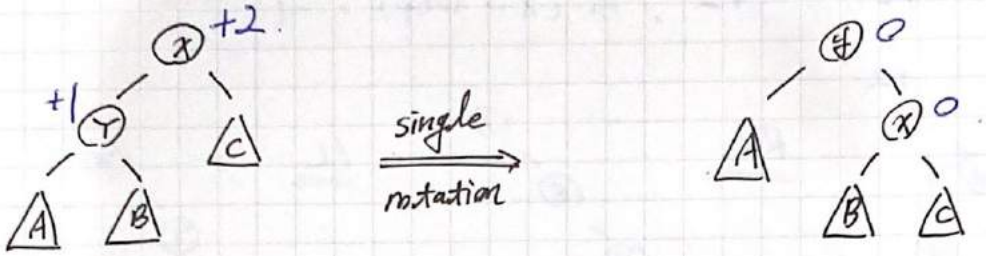
Step 1. Insert the new ~~key~~ as a ~~new leaf~~ just as in a bst.

2. Trace the path from the new leaf towards the root. Each node, check if the heights of left and right differ by at most 1.

If NOT => single / double rotation.

Single Rotation

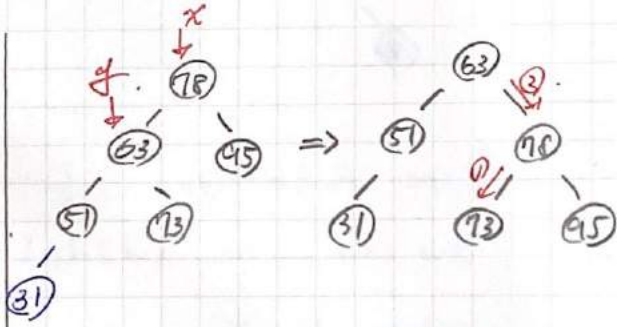
04-04



$$\begin{cases} BF(x) = +2, & BF(x \to left) = +1 \text{ or } 0 \Rightarrow LL \\ BF(x) = -2, & BF(x \to right) = -1 \text{ or } 0 \Rightarrow RR \end{cases}$$

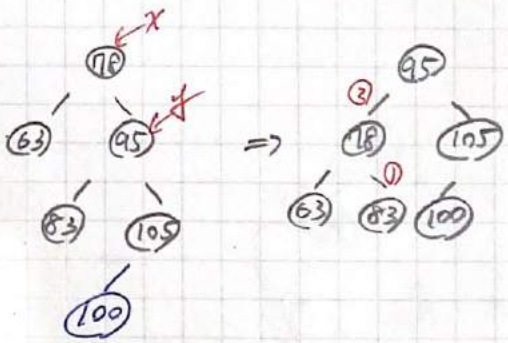
LL : // rotate x with its left child.

```
nodeType LL (nodeType x) {
    nodeType y = x → left ;
① x → left = y → right ;
② y → right = x
    return y ;
}
```



RR : // rotate x with its right child.

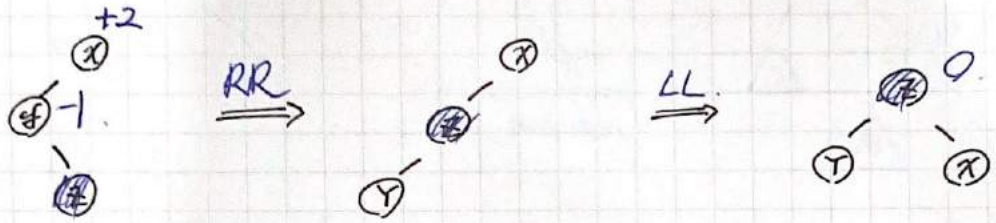
```
nodeType RR (nodeType x) {
    nodeType y = x → right ;
① x → right = y → left ;
② y → left = x ;
    return y ;
}
```



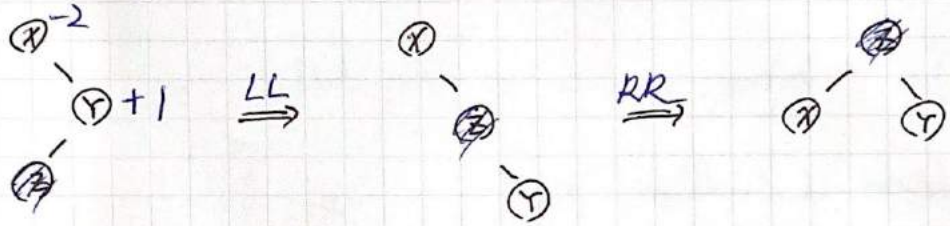
Double Rotation

04-05
04-06

LR : $BF(x) = +2$, $BF(x \rightarrow left) = -1$.



RL : $BF(x) = -2$, $BF(x \rightarrow right) = +1$.




```
nodeType LR (nodeType x) {
    x → left = RR (x → left) ;
    return LL (x) ;
}
```

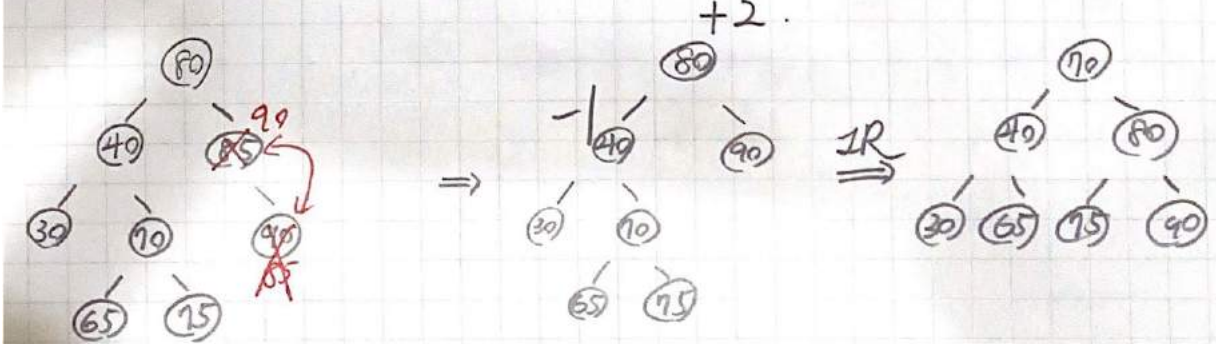
```
nodeType RL (nodeType x) {
    x → right = LL (x → right) ;
    return RR (x) ;
}
```

Delete

① 和 in_order successor swap .

② Check BF ⇒ rotation .

04-07



Red - Black Tree

04-08

* Represent each 3-node and 4-node in a 234 tree as an equivalent BST.

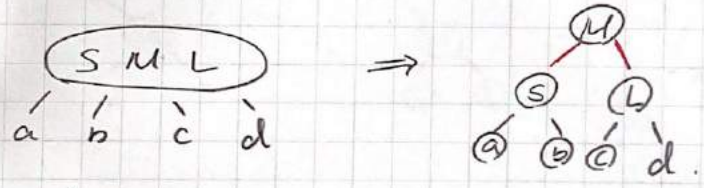
* A BST to represent a 234 tree.
  Maybe skewed ← rotations like AVL Tree.

* Has the advantage of a 234 tree, without the storage overhead. Easy to keep balanced and simple insertion / deletion.

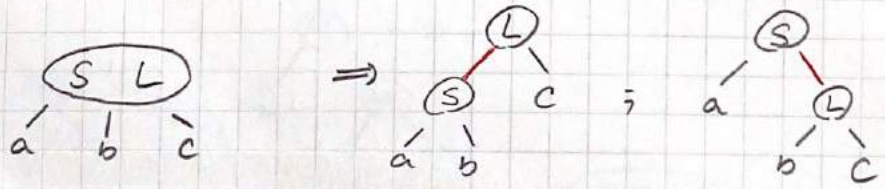
234 Tree → Red-black Tree

04-09

Type 1 : 4-node



Type 2 : 3-node



* Classify nodes by the colors of child pointers.

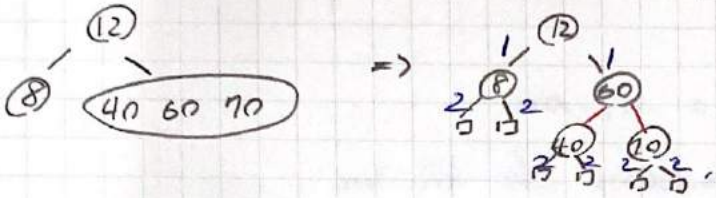
2 R	⇒ 4-node
1 B 1 R	⇒ 3-node
2 B	⇒ 2-node

Red-black vs. 234

04-10

* Every external path has an equal number of black pointers

  ⇒ Height of 234 Tree

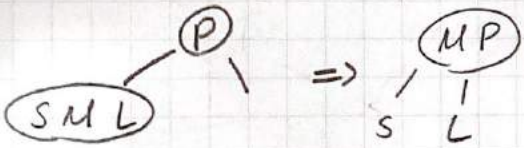


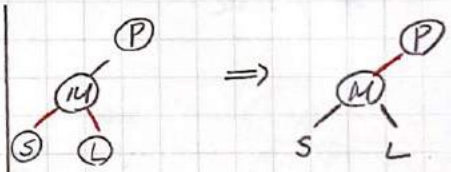
* External path cannot have 2 consecutive red pointers

  ⇒ Rotations to recover.

Splitting

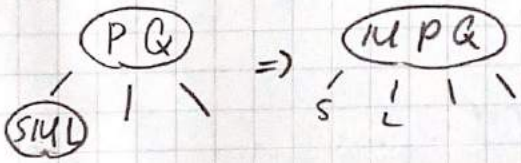
Case 1 : Parent is a 2-node

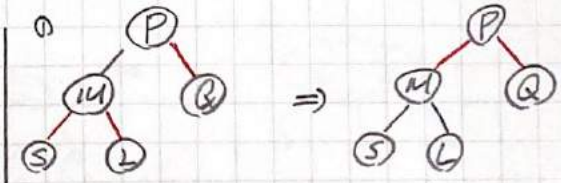


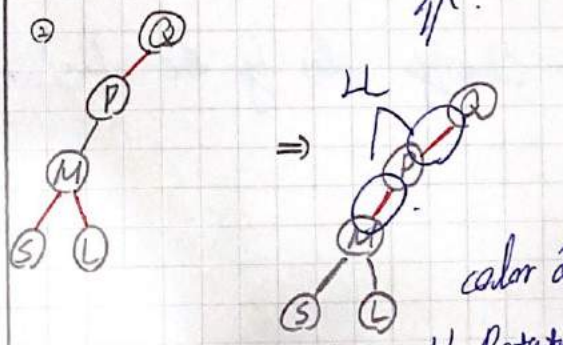


Case 2 : Parent is a 3-node

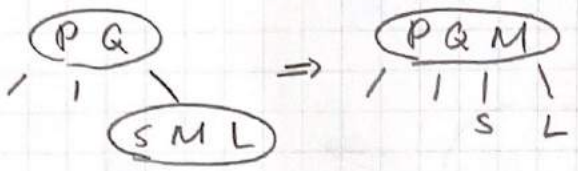
2-1

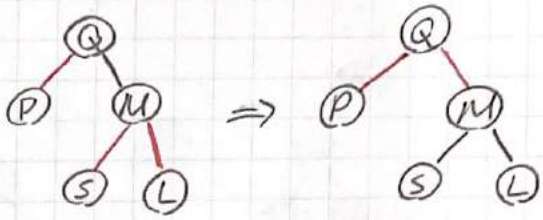




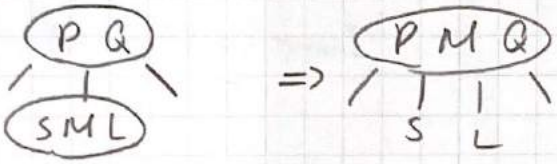


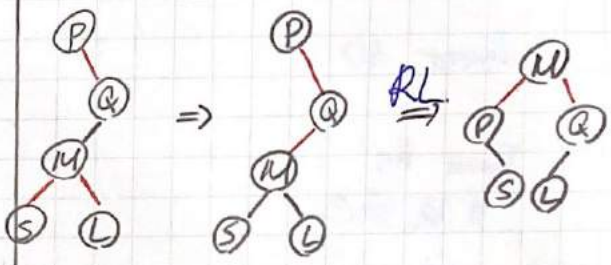
2-2





2-3



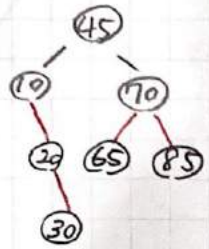


Insertion

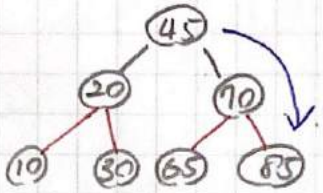
04-13

1. Splits of a node with 2 red pointers (like 4-node in 234 Tree) occur only on the path from the root to a leaf (downward)
2. Set the pointer to a new-added node as red.
3. Rotate if there are 2 consecutive red pointers.

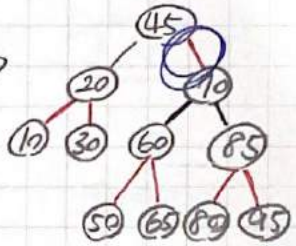
Ex. Insert 30, 50, 65, 80, 95, 40, 5, 55.



RR ⇒



⇒

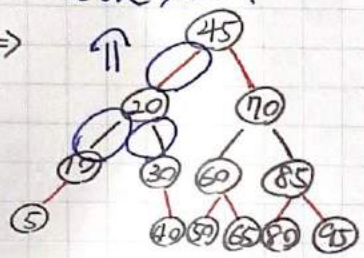


Insert 30

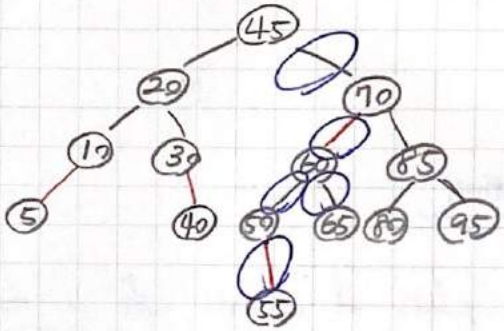
Insert 50, 65, 80, 95

Insert 前
先換顏色.

⇒



⇒



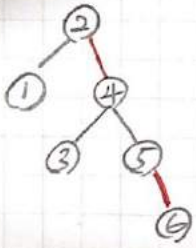
Insert 40, 5.

04-14

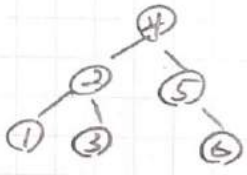
Red-black vs. AVL.

* 有些情況 樹高 AVL < Red-black.

Ex. Insert 1, 2, 3, 4, 5, 6.



vs



↑

比較平衡.

(search 較好)

Red - black Tree · Deletion

Main Idea :

Case I · two children → swap with the in-order successor

Case II · only 1 children → pointed to by a black pointer

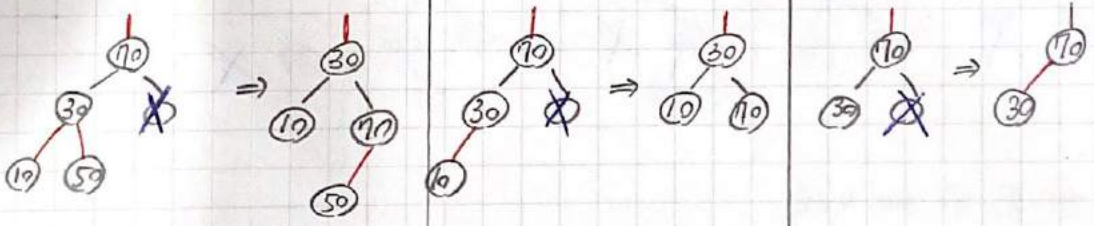
Replace the node of only 1 child with its child.

Case III · Leaf → pointed to by a red or black pointer.

Fixed number of black pointers on an external path.

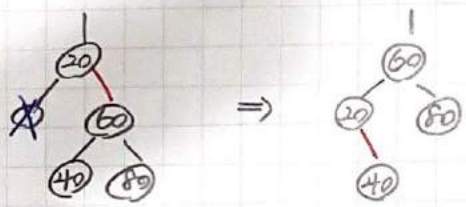
black pointer → have a sibling.

① Pointed to its parent is red 父節點 → red.



④ 2 children ④ 1 children (Rotate) ④ No children (改顏色就ok)

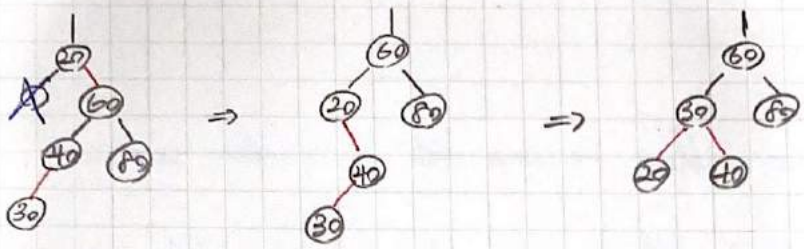
② Pointed to its sibling is red



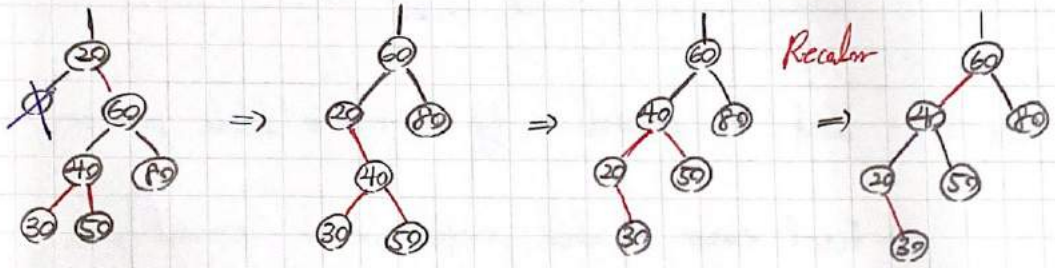
RR rotation + recolor

60 一定有 2 個 black child

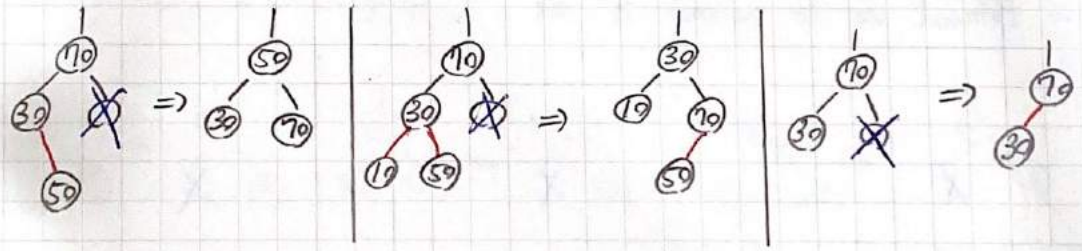
如果 40 有小孩，再 rotate 一次



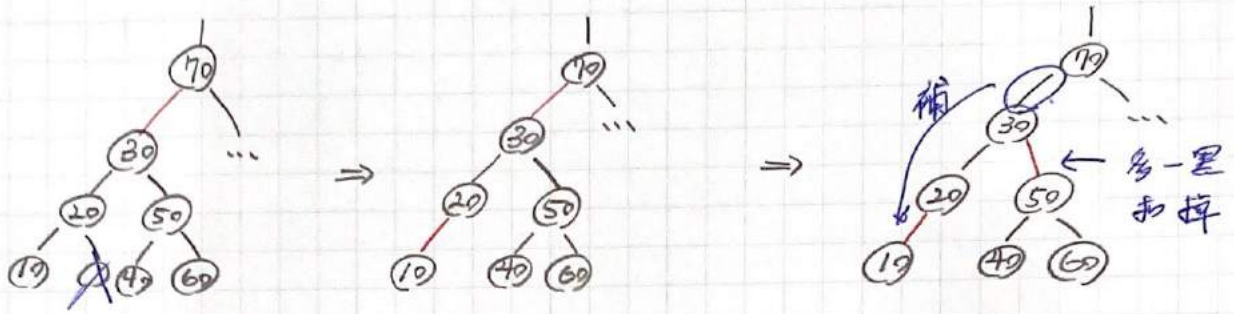
如果 40 有 2 個小孩 ⇒ 連續 2 個 Red.

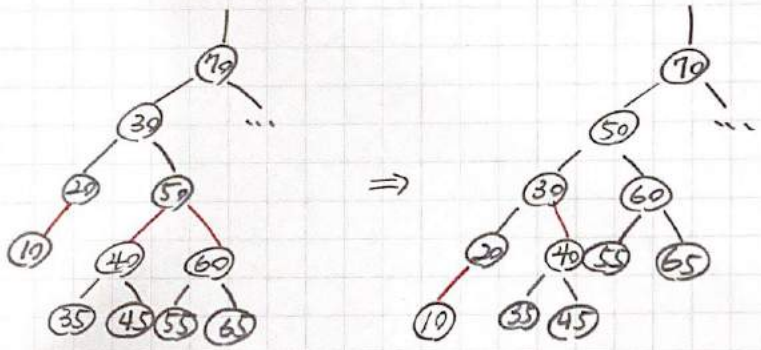


③ Pointers to its parent and sibling are black



利用 39 的小孩





No child = recolor + upward recursion
(unless the root) . consider its parents .

① pointer to its parent is red ⇒ recolor

② Pointer to and child of its sibling is red
⇒ rotation + recolor .