Deap (Double-Ended Heap)

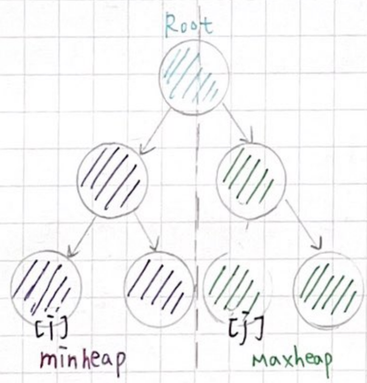
定義:

① 是一個 complete binary tree

② Root 為空(不存 Data)、Root 的左子樹為 minheap、Root 的右子樹為 maxheap

③ 令 i 為 minheap 的一個節點編號，j 為 maxheap 的一個節點編號

⇒ Deap[i] ≤ Deap[j]



Insert() : Insert x in a deap

① 將x放到最後節點的下一個位置

② 判斷x的位置在minheap堆 or Maxheap堆：

若x在minheap堆，

① 檢查x是否小於在Maxheap堆中對應點的父節點

Maxheap (cor.father)

→ 若小於，則將x和對應點的父節點的值交換，然後維護Maxheap堆。

→ 反之，維護minheap堆。(將x放到符合minheap特性的位置)

minheap (x)

若x在Maxheap堆，

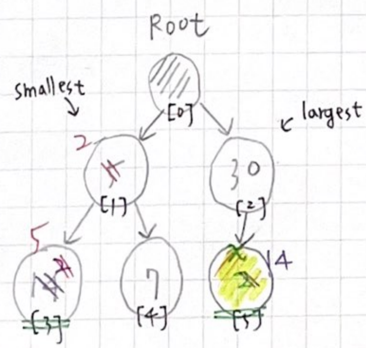
① 檢查x是否大於在minheap中的對應點

minheap (correspond)

→ 若小於，則將x和對應點的值交換，然後維護minheap堆。

→ 反之，維護Maxheap堆。(將x放到符合Maxheap特性的位置)

Maxheap (x)



step① ⇒ check x in minheap or maxhep → Maxheap!

step② ⇒ check (if) the correspond node of x in minheap $< x$

⇒ No! ⇒ swap!!

step③ ⇒ maintain minheap!

2 < 5 ⇒ swap!!

No.
Date: / /

min-Max heap

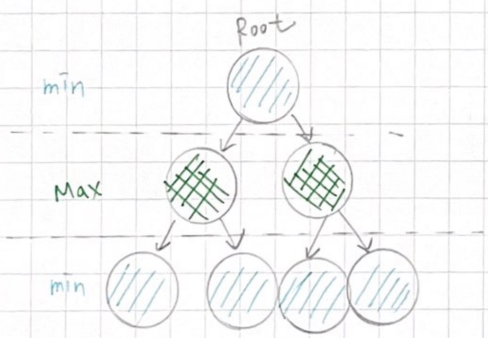
定義：

① 是一個 complete binary tree

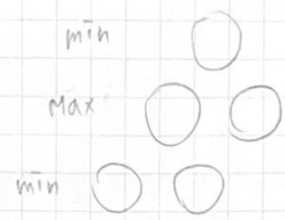
② 此樹的 level 是以 min-level 與 Max-level 交替出現，

Root 位於 min-level

△ 若 x 位於 min-level，則表示在以 x 為 root 的子樹中，x 是最小值
max-level 大



Insert() : Insert x in a min-max heap



① 將x放到最後節點的下一個位置

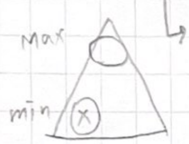
② 判斷x的位置在min-level or Max-level

若x在min-level，（上一層是Max-level）

① 檢查x是否小於其父節點

→ 若小於，維護minheap()；

→ 若大於，swap() & 維護Maxheap()；



若x在max-level，（上一層是min-level）

① 檢查x是否大於其父節點

→ 若大於，維護maxheap()；

→ 若小於，swap() & 維護minheap()；

AVL Tree

△ 平衡二元搜尋樹 (balanced binary search tree)

△ maintains the tree height close to the minimum
保持樹高接近最小值

Main idea:

1. After each insertion or deletion: 新增 or 刪除
   ① check whether the tree is still balanced. 檢查樹是否平衡
   ② If the tree is unbalanced, rotate to restore the balance.
   不平衡 → "旋轉" 使樹保持平衡

2. Balance Factor (BF) 平衡係數
   BF (a node) = h (left subtree) − h (right subtree)
   左子樹高 − 右子樹高

✩ The heights of the left and right subtrees of any node in a binary search tree differ by no more than 1.
何為平衡? ⇒ 任何節點的左、右子樹的樹高差 ≤ 1

✩ 每次新增或刪除節點只會影響某個子樹的樹高 +1 or −1
所以 BF 是 +2 or −2 時要進行 rotation

Date . .

實作：

1. After each insertion or deletion：

   ① check whether the tree is still balanced

   ② If the tree is unbalanced, rotate to restore the balance.

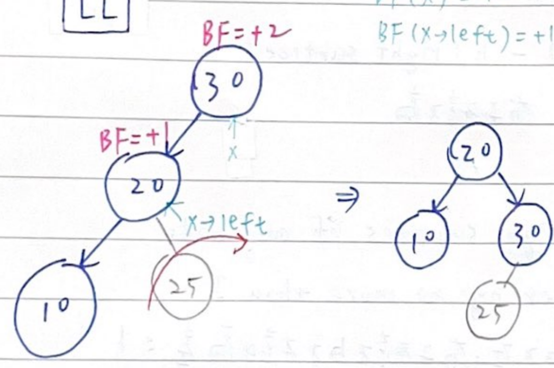
Rotation

- single rotation ⇒ RR, LL　　單一旋轉
- double rotation ⇒ RL (LL→RR), LR (RR→LL)　　複式旋轉

△ single rotation：

$\boxed{LL}$

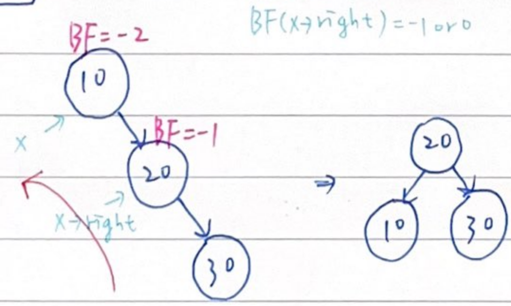
BF(x) = +2
BF(x→left) = +1 or 0




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

$\boxed{RR}$

BF(x) = -2
BF(x→right) = -1 or 0



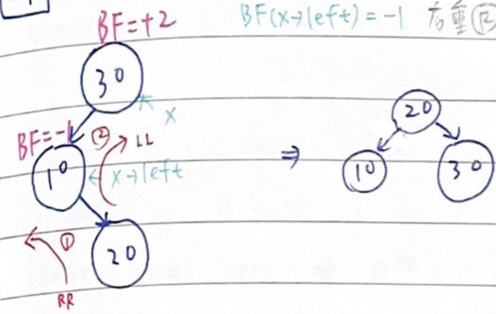

```
// rotate x with its right child
nodeType rotateRR (nodeType x) {
  nodeType y = x→right;
  x→right = y→left;
  y→left = x;
  return y;
} // rotateRR()
```

Date

△ double rotation

LR

BF(x) = +2 左重 (L)
BF(x→left) = −1 右重 (R)

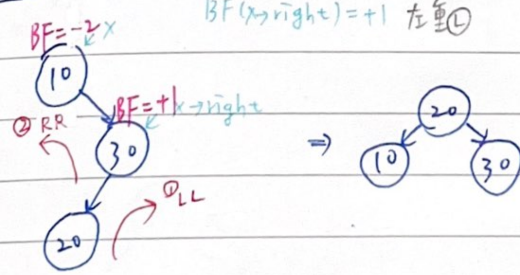


// first rotate left child with its right child ⇒ (RR)
// then, rotate node x with its new left child ⇒ (LL)

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

RL

BF(x) = −2 右重 (R)
BF(x→right) = +1 左重 (L)

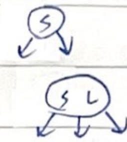


// first rotate right child with its left child ⇒ (LL)
// then, rotate node x with its new right child ⇒ (RR)

```
nodeType rotate RL (nodeType x) {
    x→right = rotateLL(x→right);
    return rotateRR(x);
}// rotateRL()
```

2-3 Tree

- 2-node ⇒ has one dataitem and two children
- 3-node ⇒ has two dataitem and three children

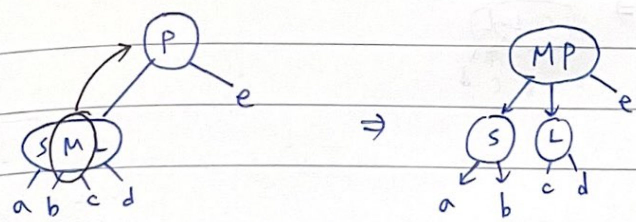


- △ traverse a 2-3 tree ⇒ inorder traversal
- △ searching a 2-3 tree is as efficient as searching the shortest binary search tree ⇒ $O(\log_2 n)$

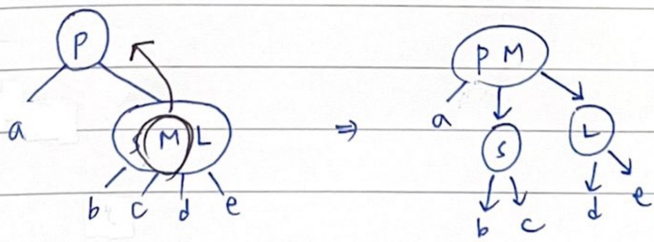
Insertion: 先新增再檢查要不要分裂

- Insertion into a 3-node causes it to divide. (split)

(a)



(b)



To insert an item I into a 2-3 tree:

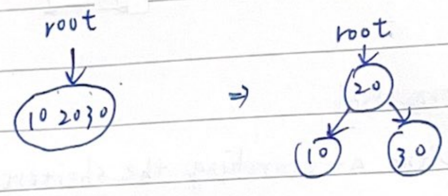
Step:
① Locate the leaf at which the search for I would terminate (結束,終止)
② Insert the new item I into the leaf.
③ If the leaf now contains only two items ⇒ end
④ If the leaf now contains three items ⇒ split

split the leaf into two nodes (n_1, n_2)

Date (split 3node)

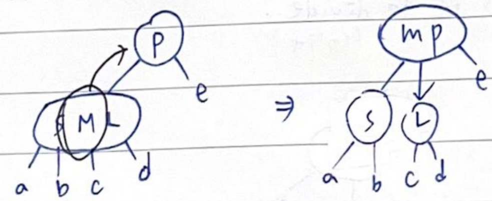
split: (recursion) 從新增的位置一路檢查回 root ⇒ upward

case①: 在根部分裂 → create a new node
& split the root into two nodes



case②: 其他地方分裂 (internal node)

Ex:



2-3-4 tree

△ 2-node ⇒ has one dataitem and two children

3-node ⇒ has two dataitem and three children

4-node ⇒ has three dataitem and four children

△ general tree, Not binary tree

△ never taller than a 2-3 tree

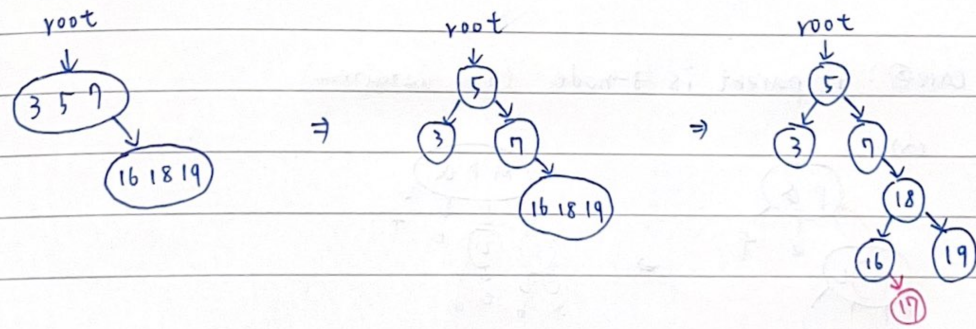
△ 2-3-4 tree is always balanced

△ 2-3-4 tree requires more storage than a binary search tree.

Insertion : 先檢查經過的路上是否有4node，再新增 ⇒ downward

(有的話就分裂) ⇒ 不用recursion

Ex: 目前要新增:17



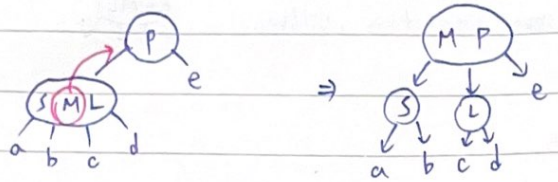
☆ Split occur only at the path from the root to a leaf. (downward)

Date . .

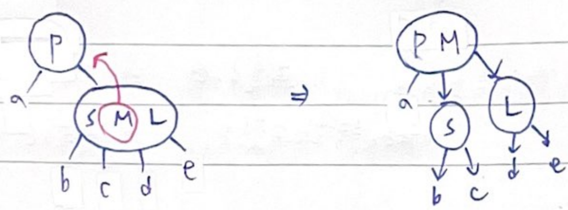
Split: (split 4node)

case①: if parent is 2-node (one dataitem)

(a)

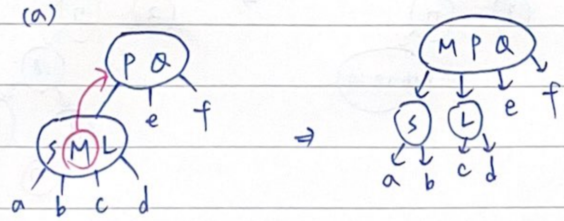


(b)

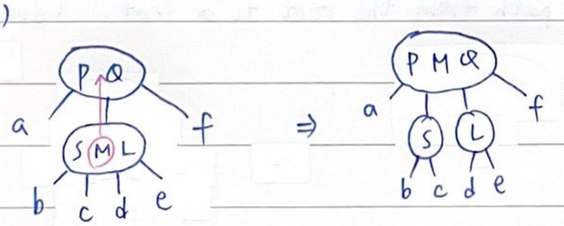


case②: if parent is 3-node (two dataitem)

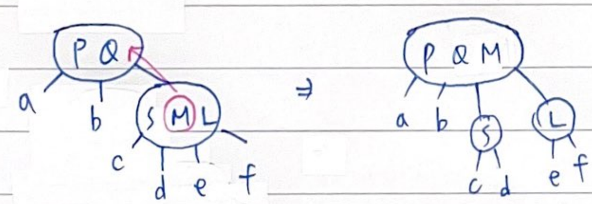
(a)



(b)



(c)



Date

Red-black Tree 紅黑樹

- △ 2-3-4 tree 的結構 + AVL 的旋轉
  以 binary search tree 呈現
- △ Represent each 3-node & 4-node in a 2-3-4 tree as an equivalent binary search tree
- △ a binary search tree represent a 2-3-4 tree maybe skewed,
  ⇒ rotations like AVL tree
- △ has the advantages of a 2-3-4 tree, without the storage overhead.
- △ easy to keep balanced and simple insertion / deletion

- △ class RBTreeNode {

```
class RBTreeNode {
    TreeItemType  item;
    RBTreeNode*   leftChildPtr, rightChildPtr;
    colorType     leftColor, rightColor;
};
```

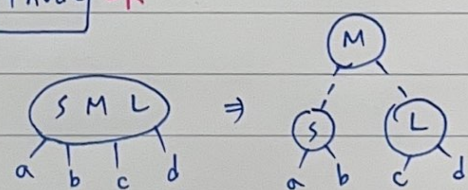
Main idea:

* 紅,黑標記在 pointer 上

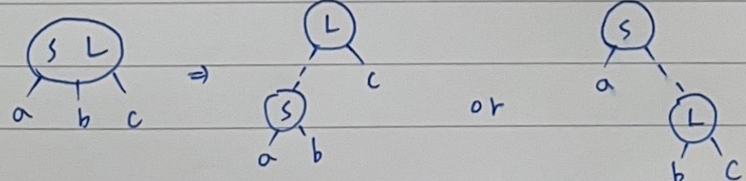
- 2 node → 2B
- 3 node → 1R1B
- 4 node → 2R

— Black
--- Red

$\boxed{\text{4node}}$ 2R



$\boxed{\text{3node}}$ 1R1B



Date

Split: (split 4-node)

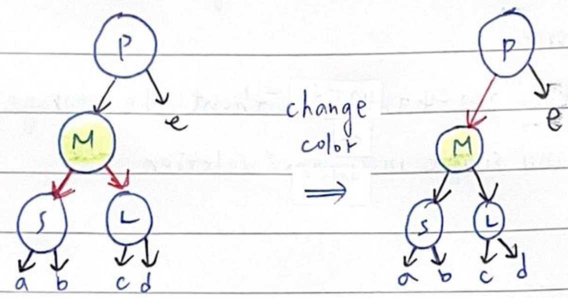
case ①: parent is 2-node

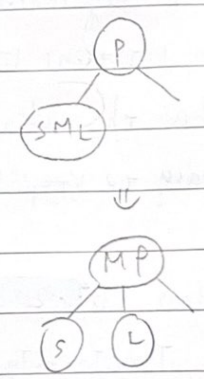
⇒ change color

{ to child: R→B
  from parent: B→R

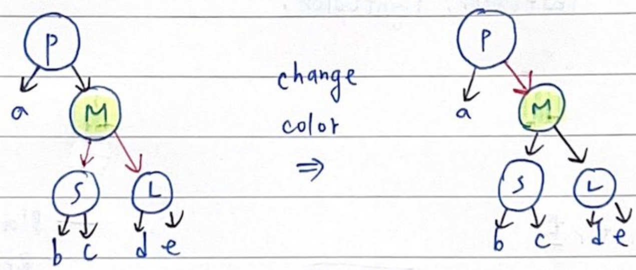
等同於 2-3-4 tree

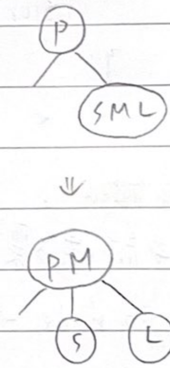
(a)





(b)





Date . .

case ②: parent is 3-node

① change color

{ to child : R → B
  from parent : B → R

② check if need rotation

⇒ LL, RR, RL, RL