DS2 CH1-4

Priority Queues

Sort algorithm

	worst case	average case
Selection	n^2	n^2
Bubble	n^2	n^2
Insertion	n^2	n^2
Merge	nlogn	nlogn
Quick	n^2	nlogn
Radix	n^2	n

selection : pqInsert O(1) pqDelete O(n)

insertion : pqInsert O (n) pqDelete O(1)

Heap

doubled-ended priority queues (DEPQ)

- min-max
- doubled-ended (DEAP)

Forestunion

- binomial
- fibonacci

Min-max Heap

Insert :

1. check level if min or max

2. check if need to swap with its parent
    a. True : ReheapUp (compare with its grandparent) from its parents
    b. False : ReheapUp (compare with its grandparent) from current

Delete the smallest

1. replace the root with last element
2. check if need to swap with its smaller child (then ReheapDown)

Delete the largest

1. replace maximun with last element
2. check if need to swap with its larger child (then ReheapDown)

Deap

Insert :

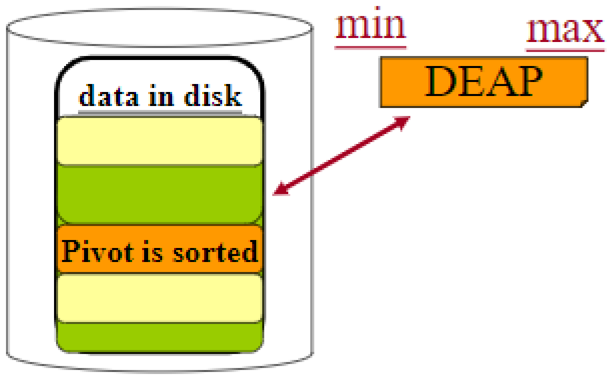
1. examin the cor node is less / bigger than current
    a. Left : swap if bigger than cor
    b. Right : swap if less than cor
2. ReheapUp (cur / cor)

Delete the smallest

1. replace the root of left (min-heap) with last element
2. ReheapDown
3. examine the cor nodes (like insert)

Application :

External Sort →Large amount of Data on secondary storage



Balanced Search Tree

ADT table

- use search key to identify its items (assume distinct keys)

Comparing Linear Implementations

Unsorted array :

- Insert O(1)
- Deletion requires shifting Data : O (n)
- Retrieval requires a sequential search : O (n)

Sorted array :

- insert / delete require shifting data : O (n)
- retrieval (efficient) : O(logn)

Unsorted pointer :

- insert (efficient) : O (1)
- deletion : O (n)

- retrieval : O(n)

Sorted pointer :

- No data shift
- insert / deletion / retrievals : O (n)

Linear

- for small table / unsorted table with few deletions

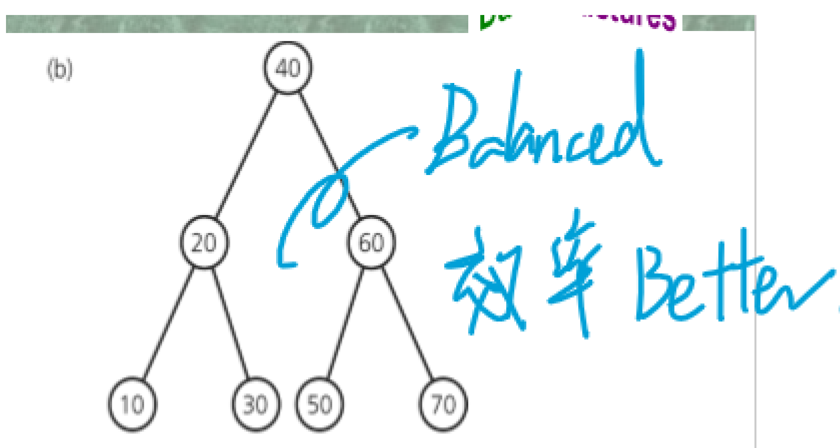
Nonlinear

- usually a better choice than further
- a balanced binary search tree

Binary search tree

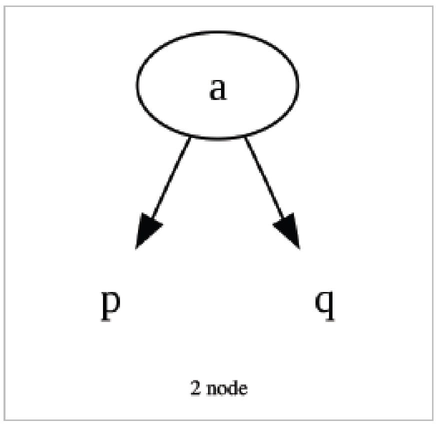
Height :

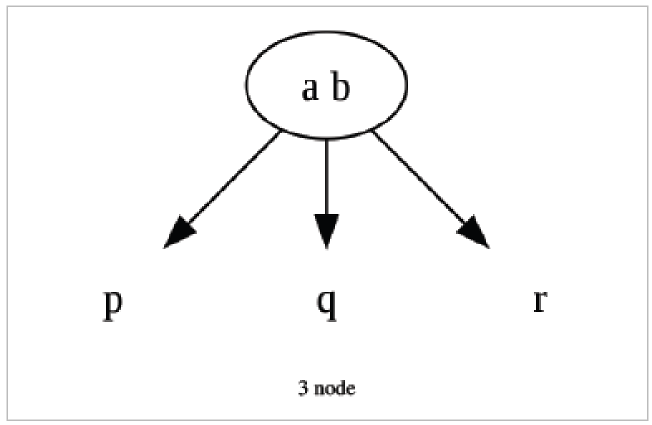
- max → n
- min → log(n+1)
- relate to the order of insertion / deletion



2-3 Tree

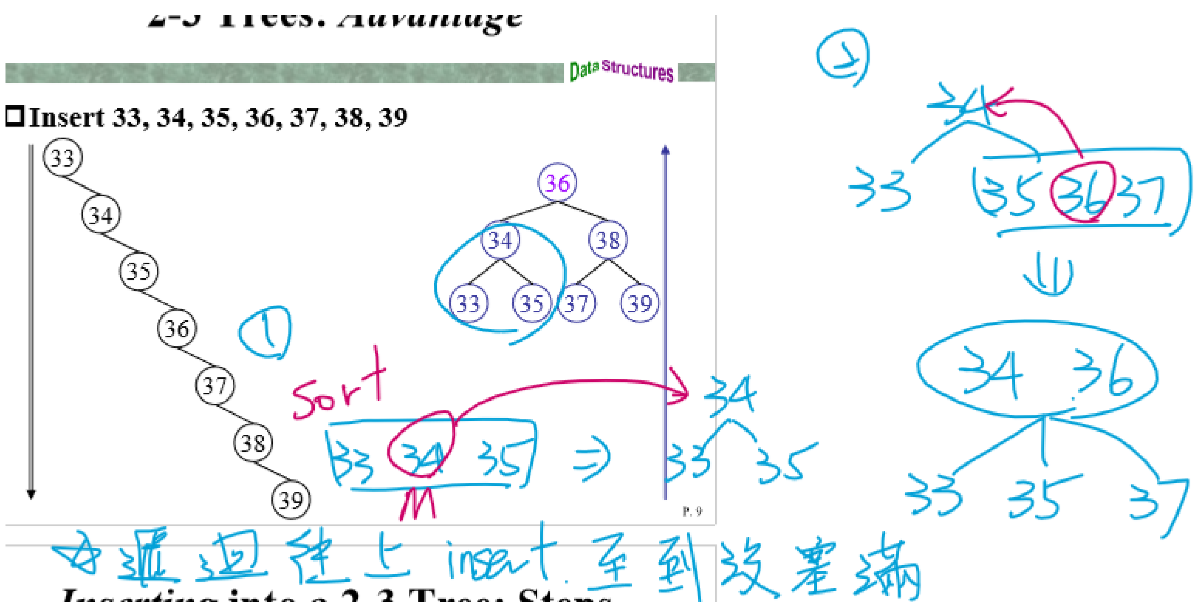
- are general tree, but not binary trees
- never taller than a minimum height binary tree (efficient search)
- Node (num of child)
  - 2-node (just like normal)
  - 3-node





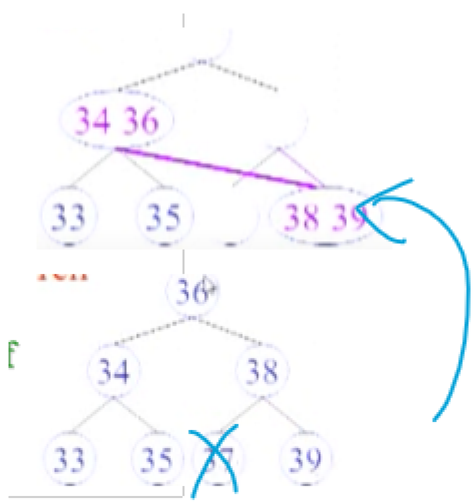
Insert

1. find the leaf which new item would terminate
2. insert the new item into leaf
3. check the leaf if contain two or three items
   a. if contains three items, splits (upward recursion) leaf



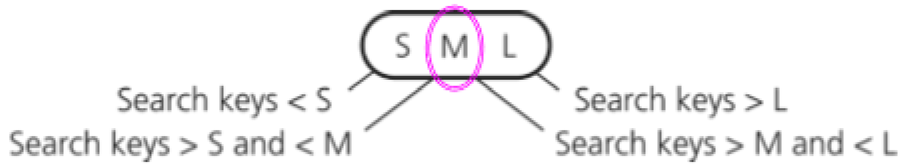
delete

1. find target item then delete it
2. if target item is not leaf than swap with in-order successor
3. if the leaf now contains no item :
    a. redistribute the values (3-node)
    b. merge into a leaf (2-node)

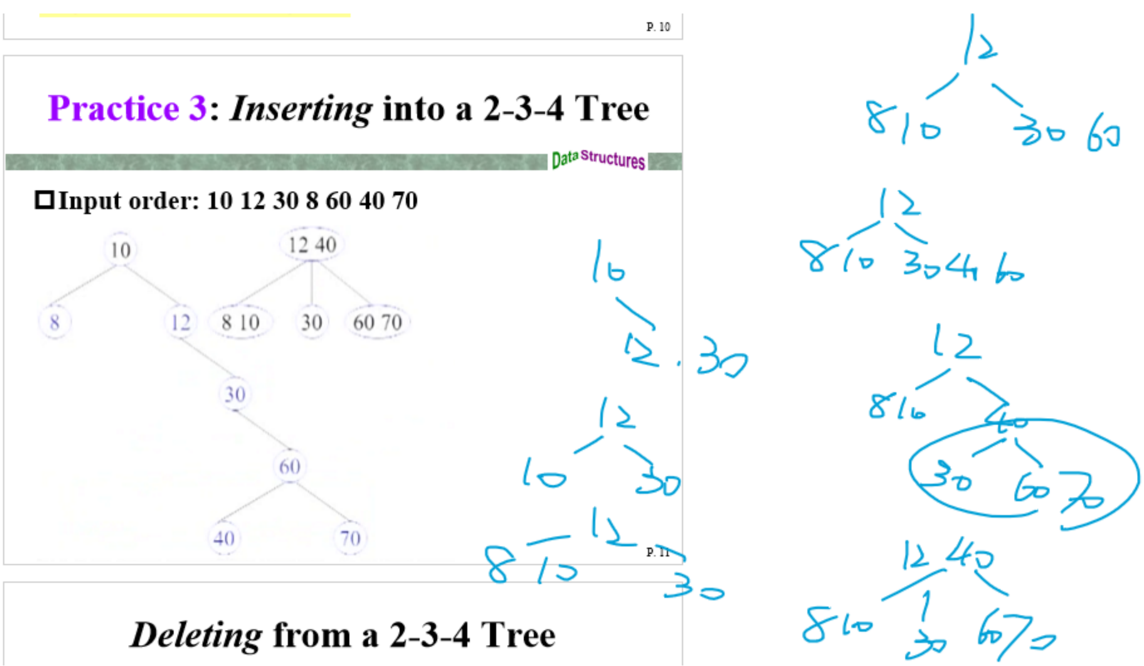


2-3-4 Tree

- never taller than 2-3 tree
- Node
    - 2 / 3 node : just like 2-3
    - 4 node



- split only occur at the path from root to leaf
- insetion and deletion are more efficient than 2-3 tree
- require more stroage than a BST



split :

1. move middle item up to parent node
2. when 4-node split , its parent can not be a 4-node
3. downward recursion

delete (always occur at a leaf) :

1. find target node at which node
2. swap with its inorder successor
3. if leaf is 3-node or 4-node then remove it
4. 先合併再刪除

AVL Tree

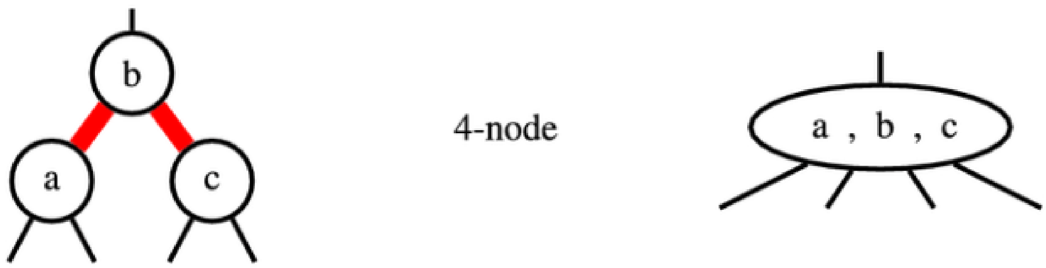
- a balanced BST
- search as efficiently as a minimum BST (due to height close to it)
- balanced factor = height (left subtree) - height (right subtree)
  - balanced factor should not differ by no more than 1
- Rotation
  - factor < -1 ⇒ left rotate
  - factor > 1 ⇒ right rotate
  - if subtree positive/ negative is diff then do double rotate

Insetion

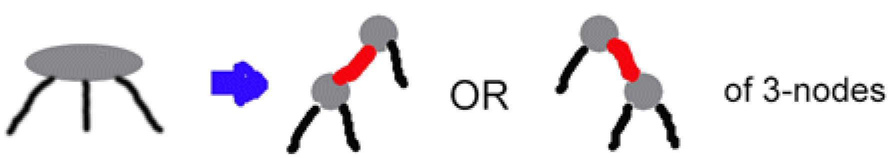
1. insert new item just like BST
2. trace the path from new leaf to the root
   a. check balanced for each node. If unbalanced then do single or double rotate

Red-Black Tree

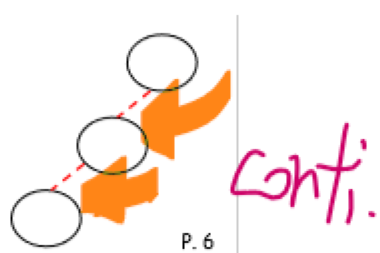
- represent each **3-node** and **4-node** in a 2-3-4 tree as an equivalent BST
    - rotation like AVL Tree
- has advantage of 2-3-4 tree without storage overhead
- an equal number of black pointer
- split only occur on the path from the root to leaf
- Node (classify node by the colors of child pointer)
    - 4-node ⇒ 2 red



- 3-node ⇒ one black pointer & one red pointer

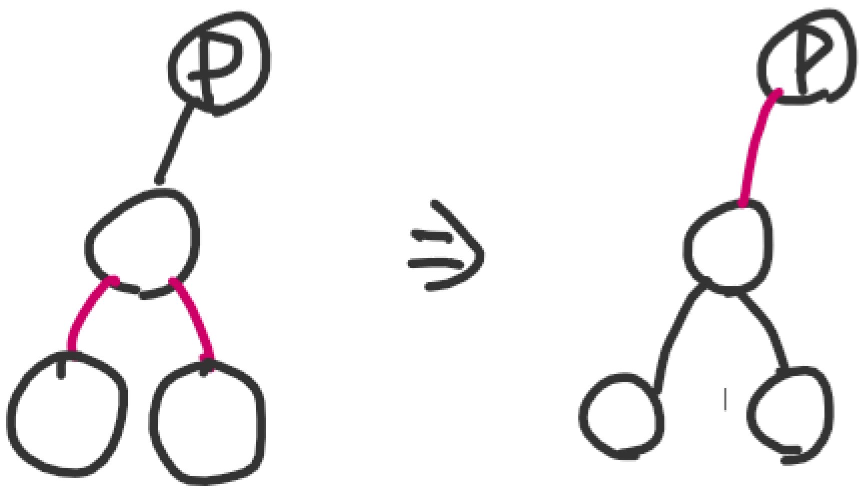


- 2-node ⇒ 2 black pointer
- red pointer cant be conti

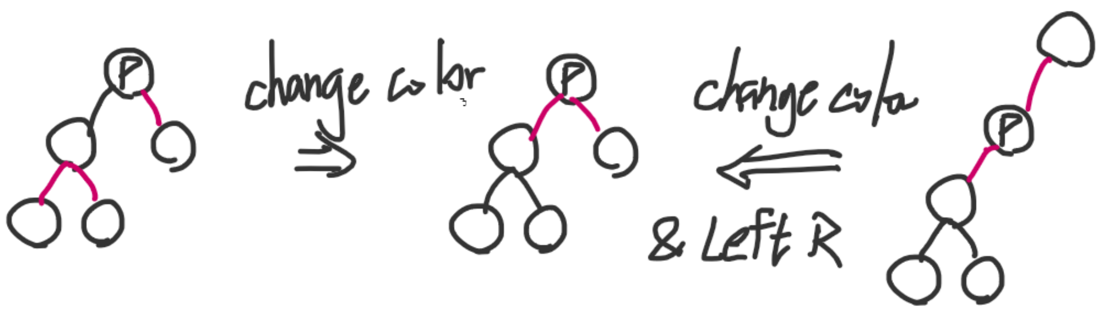


Split (depends on the num of parent's node)

1. 2-node ⇒ change color



2. 3-node ⇒ change color & rotate (conti red)
    a. grandparent is 3-node and sibling is not 3-node ⇒ left rotate



    b. grandparent is not 3-node and sibling is 3-node ⇒ right rotate
    c. just like AVL do double rotate if current node and its child doesn't on same side

Insert

1. set the pointer of new leaf as red
2. rotate if red conti