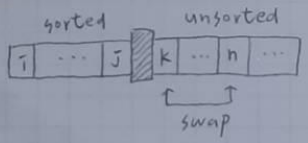
Priority Queue (FIFO)!

* Sorting algorithm

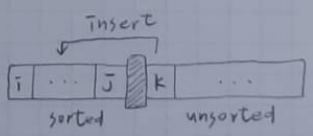
	Worst	Average	Best
Binary Search Sort	$O(n)$	$O(\log n)$	—
Bubble	$O(n^2)$	$O(n^2)$	$O(n)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n^2)$	$O(n^2)$	$O(n)$
Quick	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix	$O(n)$	$O(n)$	$O(n)$

* Selection sort



insert : $O(1)$
delete : $O(n)$

* Insertion sort



insert : $O(n)$
delete : $O(1)$

* Huffman coding (heap)

ex.

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

↳ 任何一個 code 都不會是另一個 code 的 prefix

↳ 用於壓縮, 編碼

* Max-heap (insert : pseudo code)

```
items[size] = newItem; // put new item at the end of heap array
int place = size;
int parent = (place-1)/2;
while (parent >= 0 && items[place] > items[parent]){
    HeapItemType temp = items[parent];
    items[parent] = items[place];
    items[place] = Temp; // swap items[place] with items[parent]
}
size++;
```

* Red-black Tree (insert)

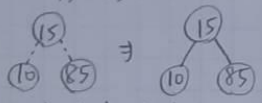
↳ 新增節點時，預設指標為 red

↳ 做法：

1. 路徑上換顏色，再加入節點
2. 旋轉

↳ special case：root

ex. 15、85、10



直接換顏色就好！

* 比較紅黑樹 & AVL 樹

1. 樹高：AVL較矮
2. 新增刪除：紅黑樹較方便
3. 搜尋：AVL較快

* Red-black Tree (delete)

↳ cur has 2 children

1. swap with in-order successor
2. delete

↳ cur has 1 child

1. This child must be red
2. swap with its child
3. delete

↳ cur is a leaf

A. cur is red : delete

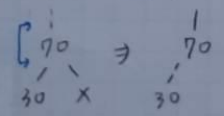
B. cur is black

* cur must have sibling

a. pre is red

↳ sibling has no children:

recolor + delete



↳ else (sibling has children)

* children must be red

(red-black tree : delete

→ cur is a leaf

→ cur is black

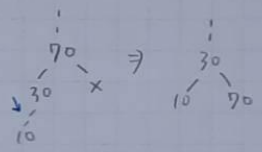
→ pre is red

→ cur's sibling has children 續)

* 以 cur 是 pre 的 right child 舉例

↳ sibling has left child only:

LL + recolor + delete

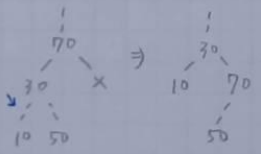


↳ sibling has right child only:

LR + recolor + delete

↳ sibling has 2 children:

LL + recolor + delete



Search！(由上而下的平衡搜尋樹)

✳ AVL Tree

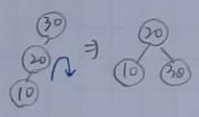
↳ Balance Factor (H表height)

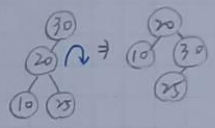
$BF(x) = H(x\to left) - H(x\to right)$

↳ Single Rotation

↳ 左邊太重(LL)，做右旋

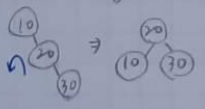
1. $BF(x) = 2$
2. $BF(x\to left) = 1$ or 0

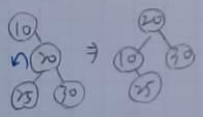




↳ 右邊太重(RR)，做左旋

1. $BF(x) = -2$
2. $BF(x\to right) = -1$ or 0

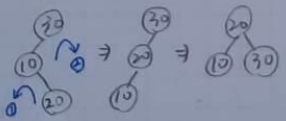




↳ Double Rotation

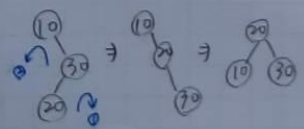
↳ LR型(先RR，再LL)

1. $BF(x) = 2$
2. $BF(x\to left) = -1$



↳ RL型(先LL，再RR)

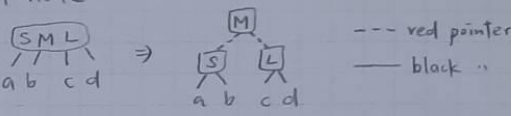
1. $BF(x) = -2$
2. $BF(x\to right) = 1$



✳ Red-black Tree (node)

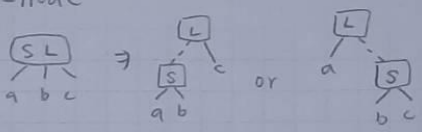
↳ 結合2-3-4樹 & AVL樹，創造一個節點上只有一個key的平衡二元樹

↳ 4-node



節點下有2 red ∋ 本來是4-node

↳ 3-node



節點下有1 red 1 black ∋ 本來是3-node

↳ 2-node

節點下有2 black

↳ 到不同子葉節點的路徑會有相同數量的 black pointers

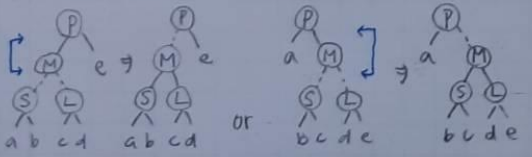
↳ 不允許連續出現2個 red pointers，出現時要旋轉

↳ 紅黑樹的樹高最高就是2-3-4樹的2倍

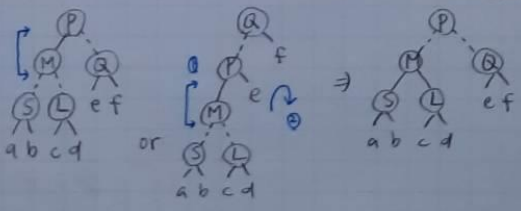
(∵ red pointers ≤ black pointers)

✳ Red-black Tree (spilt)

↳ pre is a 2-node：換顏色就好

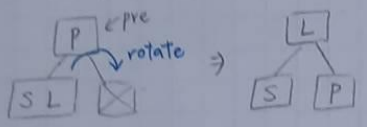


↳ pre is a 3-node：換顏色(再做旋轉)

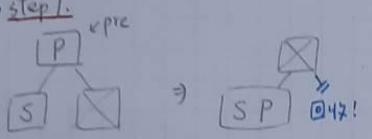


* 2-3 tree (delete - merge)

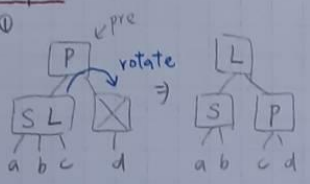
* current 節點被刪空，須重分配
* 看 sibling 有幾個 key
  - 2個：rotate，1個：merge
- 要 delete 的 value 在子葉
  1. sibling 有 2個 key

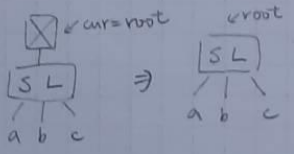


  2. sibling 只有 1個 key
     - step 1.



     - step 2.

① 

② 

- 要 delete 的 value 在 internal node
  1. Swap with the in-order successor on a leaf.
     (右邊節點的最左邊小孩)
  2. Delete the value from the leaf.
  3. 要 delete 的 value 在子葉了

* Order - review

pre : 1 2 4 5 3 6 7
in : 4 2 5 1 6 3 7
post : 4 5 2 6 7 3 1

* 2-3-4 tree

- node : 2-node, 3-node, 4-node
- insert - split
  在新增的路上，看到 3-node 就先分裂，乘下的和 2-3 樹一樣
- delete - merge
  在刪除的路上，看到節點只有 1個 key 就先合併，乘下的和 2-3 樹一樣
- summary
  1. insert/delete for 2-3-4 tree require fewer steps than those for 2-3 tree (only 1 pass from root to leaf).
  2. 2-3-4 tree requires more storage than a binary search tree (用空間換時間).
  3. 2-3-4 tree is always balanced.
  4. 2-3-4 tree is more efficient than a binary search tree.

Search!（由下而上的平衡搜尋樹）

* Linear implementation
  - sorted、unsorted
  - array、pointer
* Non-linear implementation
  - ex. binary search tree
* ~~Efficiency (linear)~~

	array-based	pointer
sorted	In: O(n) De: O(n) Re: O(log n)	In: O(n) De: O(n) Re: O(n)
unsorted	In: O(1) De: O(n) Re: O(n)	In: O(1) De: O(n) Re: O(n)

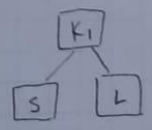
In: insert
De: delete
Re: retrieval

- 效率跟資料量相關

* Balanced binary search tree
  - 2-3 tree、2-3-4 tree
  - AVL tree、Red-black tree

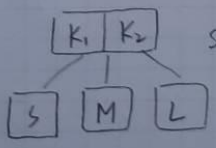
* 2-3 tree (node)

① 2-node (1 key, 2 children)



S < K1 < L

② 3-node (2 key, 3 children)



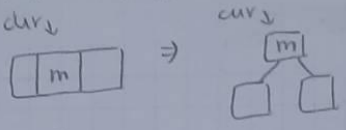
S < K1 < M < K2 < L

* ~~2-3 tree (insert - spilt)~~

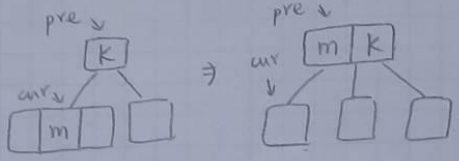
x current節點太飽滿，須要分裂

- ~~cur是子葉(在底部)~~

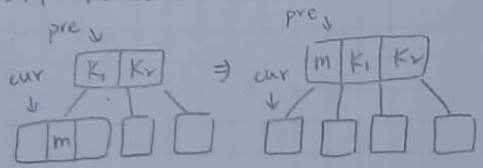
1. 只有一個節點



2. pre是一個2-node

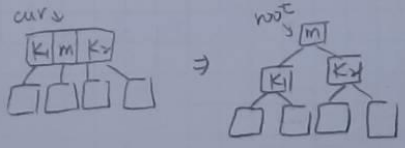


3. pre是一個3-node

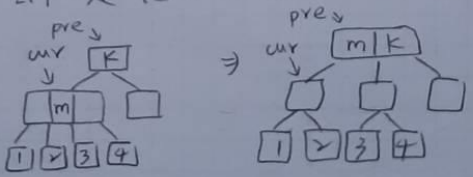


- else (cur不是子葉)

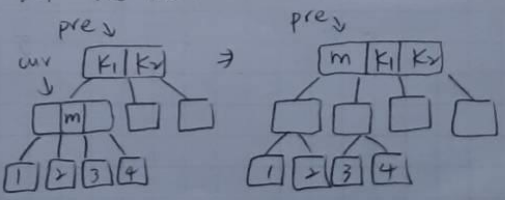
1. special case：要建新樹根



2. pre是一個2-node

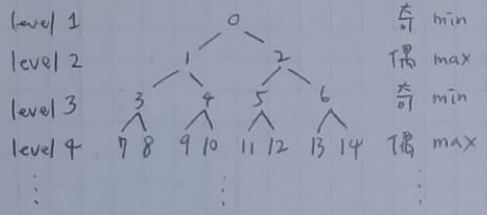


3. pre是一個3-node



Variations of Heap 堆積變形

* Double-ended Priority Queues (DEPQ)
  - ↳ min-max heap
  - ↳ double-ended heap (deap)
* Forest (union) of Heaps
  - ↳ binomial heap
  - ↳ fibonacci heap
* Min-max Heap (insert)



↳ 父節點位置 parent

parent = (i-3)/4

↳ insert one by one (pos: current position)

↳ min level

pos < parent : ① parent = (pos-3)/4 ② reMin

pos ≥ parent : ① swap ② pos = parent, parent = (pos-3)/4 ③ reMax

↳ max level

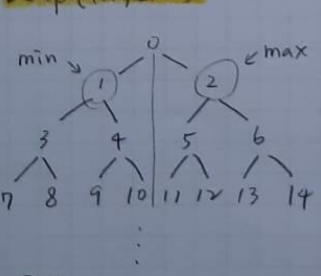
pos > parent : ① parent = (pos-3)/4 ② reMax

pos ≤ parent : ① swap ② pos = parent, parent = (pos-3)/4 ③ reMin

↳ reMin、reMax

負則由下往上檢查並維持 min/max heap 性質

* Deap (insert)



level
1
2 + 2^1
3 + 2^2
4 + 2^3
⋮

↳ Deap (insert one by one)

↳ min side

* deap[pos] > deap[corr]: ① swap ② pos = corr, parent = (pos-1)/2 ③ reMax
* deap[pos] ≤ deap[corr]: ① parent = (pos-1)/2 ② reMin

↳ max side

* deap[pos] < deap[corr]: ① swap ② pos = corr, parent = (pos-1)/2 ③ reMin
* deap[pos] ≥ deap[corr]: ① parent = (pos-1)/2 ② reMax

↳ 層數 level = (int)log2(size) + 1

↳ 對應節點位置 corr = i ± 2^(level-2)