Priority Queues = Priority + Queue

- Basic of Priority Queues.

(P1, 5, 23:55), (P2, 5, 00:05), (P3, 3, 00:10), (P4, 4, 00:15)

↓ InsertPQ(dataItem, priority).

PQ — Q: Which sorting algorithm fit?

↓ DeletePQ() or called Pull()

(P3, 3, 00:10), (P4, 4, 00:15), (P1, 5, 23:55), (P2, 5, 00:05)

Sorting algorithm

A:

	worst case	Average case
selection sort	n^2	n^2
bubble sort	n^2	n^2
insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
quicksort	n^2	$n * \log n$
Radixsort	n	n

- Application of Priority Queues

Q which is the Nearest Neighbor ? (NN).

PQ C, D, A, B

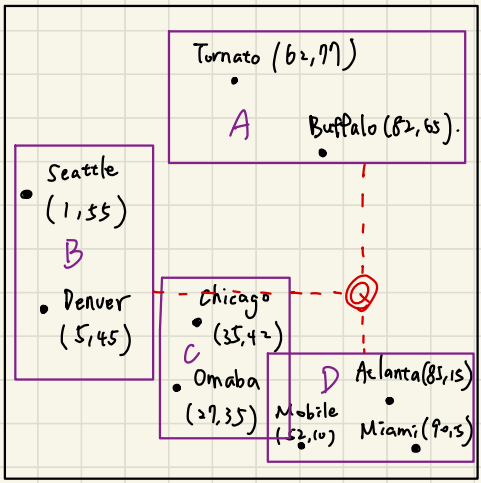
↓

PQ D, A, Chicago, B

↓

PQ (Atlanta), A, Chicago, B

Nearest Neighbor !



- Heap.

(P1, 5, 23:55), (P2, 4, 00:05), (P3, 3, 00:10), (P4, 2, 00:15)

↓

min-heap.

↓

(P4, 2, 00:15)

(P3, 3, 00:10) (P2, 4, 00:05)

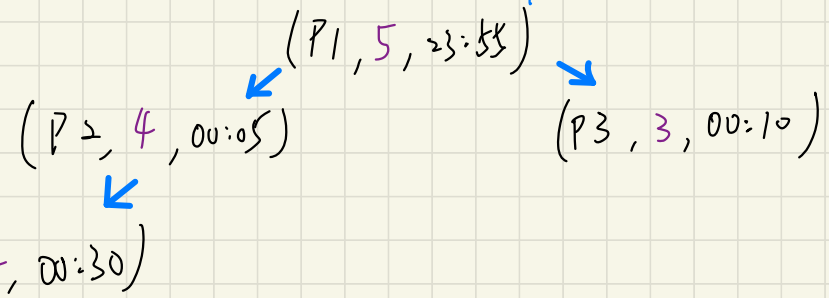
(P1, 5, 23:55)

(P1, 5, 23:55), (P2, 4, 00:05), (P3, 3, 00:10), (P4, 2, 00:15)

pqInsert(): O(log n)

max-heap

pqDelete(): O(?)

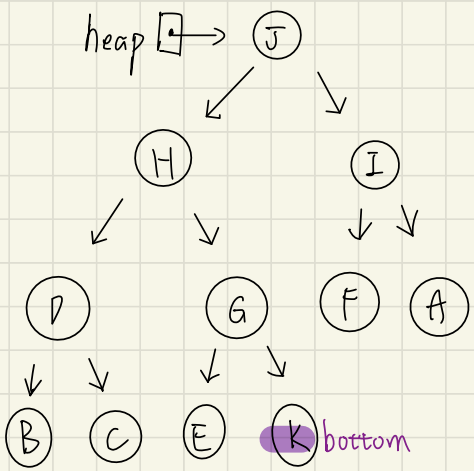


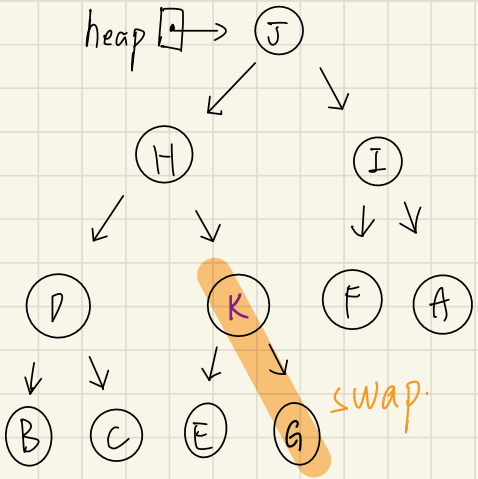
Q What is a heap?

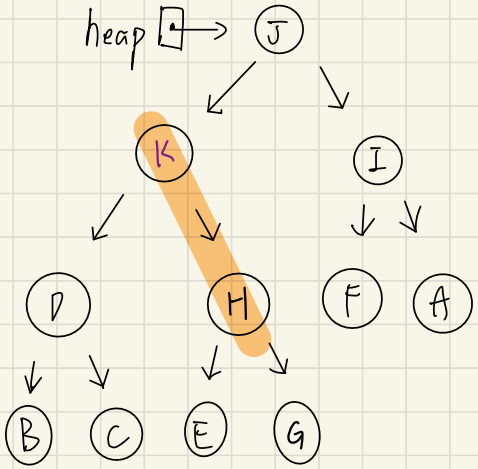
① It is a complete tree.

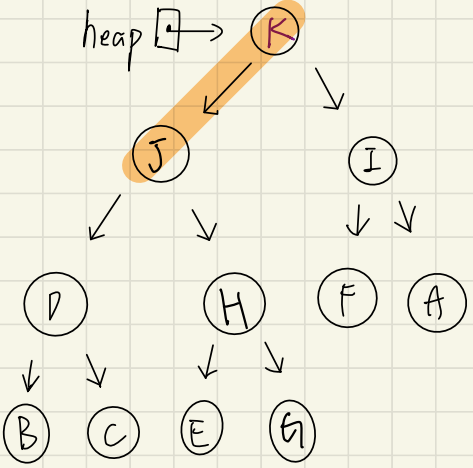
② the value stored at a node is greater (smaller) or equal to the values stored at the children (heap property).

- ReheapUp()



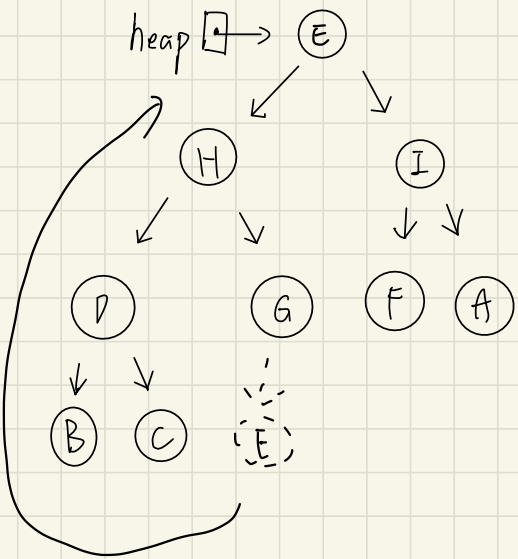


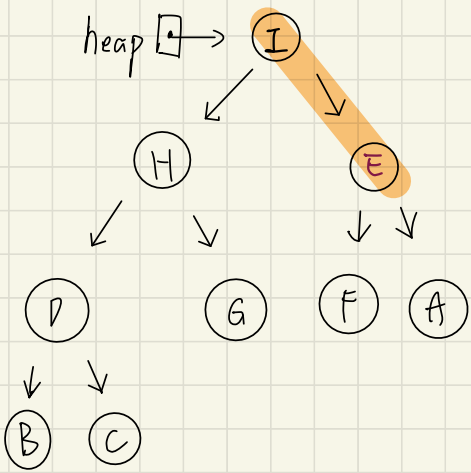


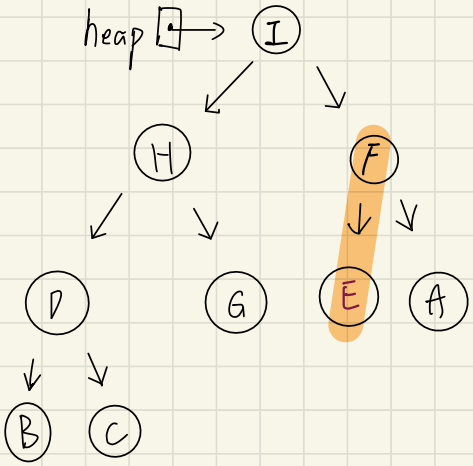


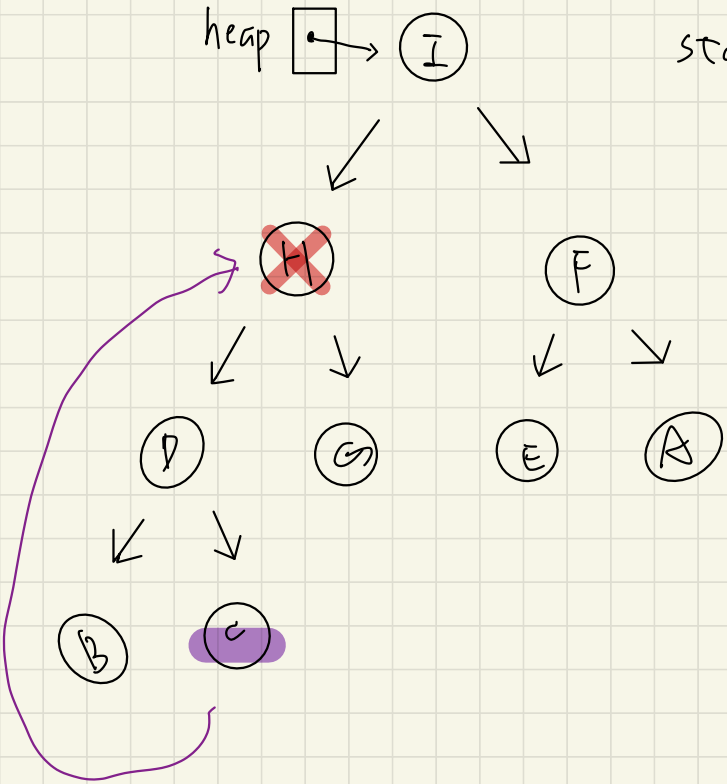
pqInsert(): O(log n)

- ReheapDown()



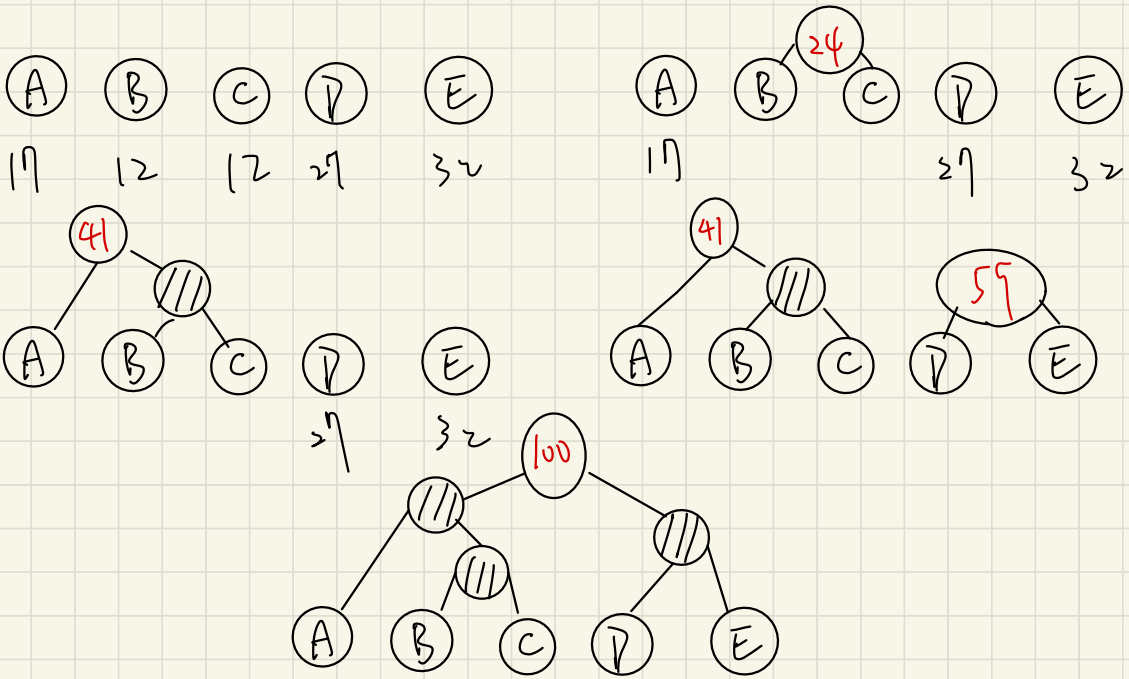




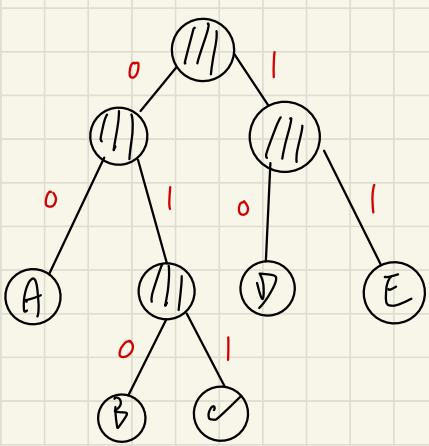


stay the tree still.

When to use a heap?



Application: Huffman Coding.



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

text

EAEBAECDEA

encoder

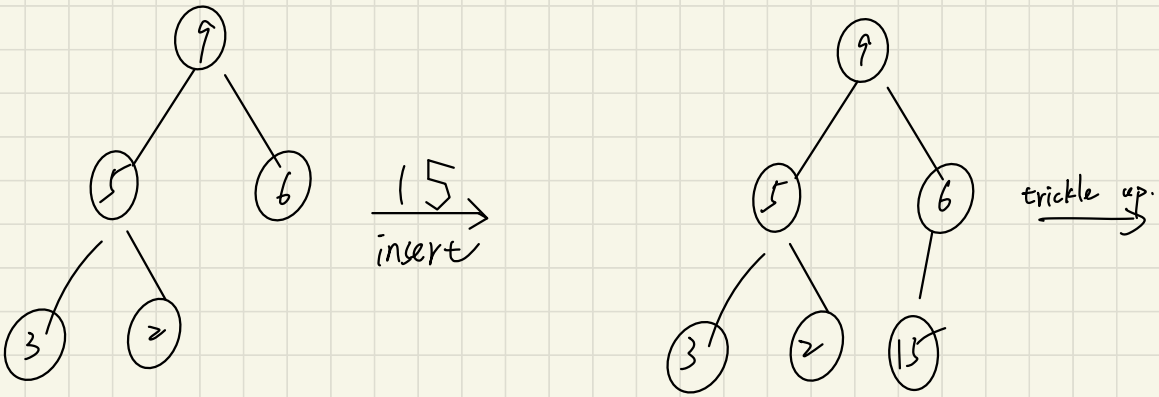
1100110001011101100

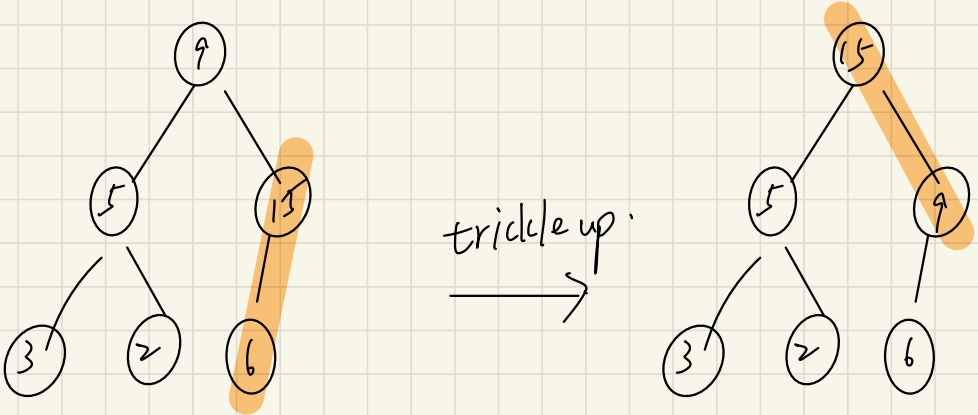
Huffman code.

decoder

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

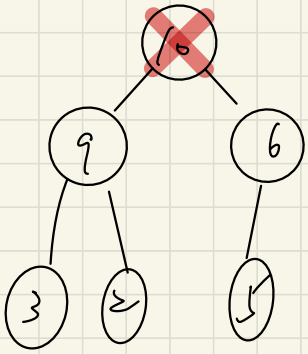
— heapInsert( ): Strategy.





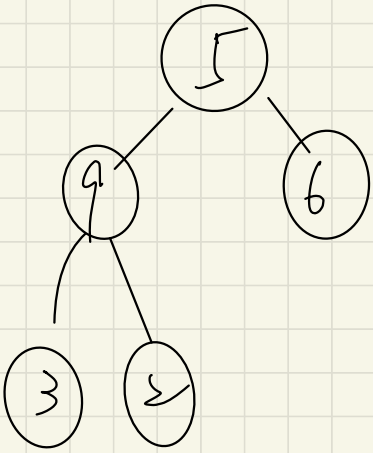
- efficiency: $O(\log n)$.

heapDelete(): Strategy.



copy the bottom node into the root

and remove the last node: --size;



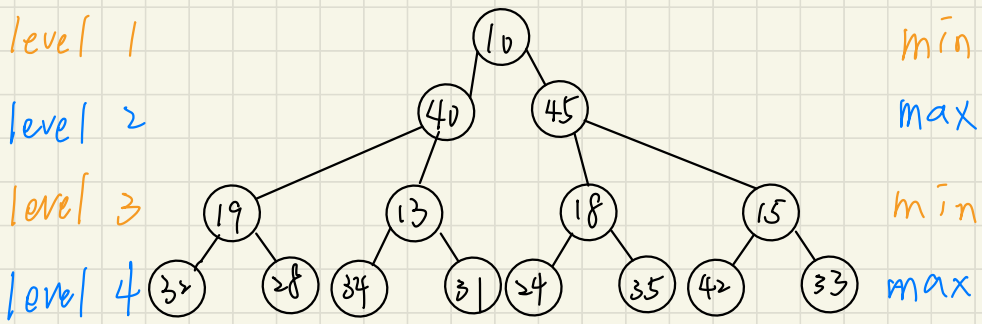
transform the semi-heap back into the heap

(use the recursive algorithm heapRebuild).

Variation of Heap.

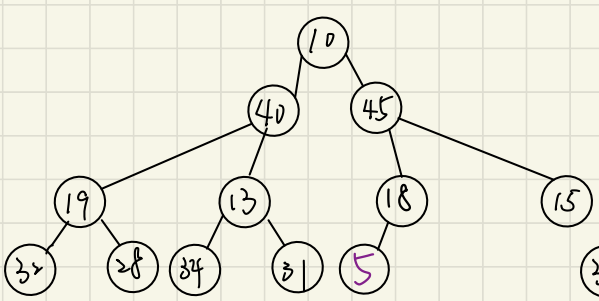
- Double-ended Priority Queues

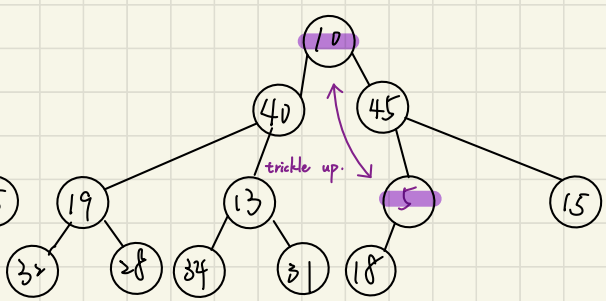
  - Min-Max Heap.



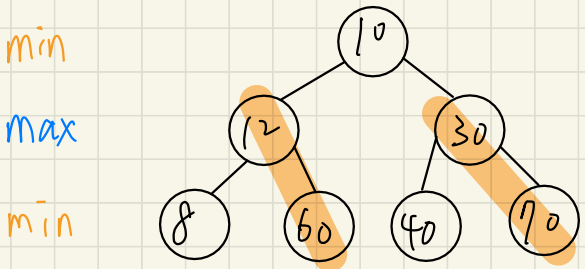
Min-Max heap : insert.

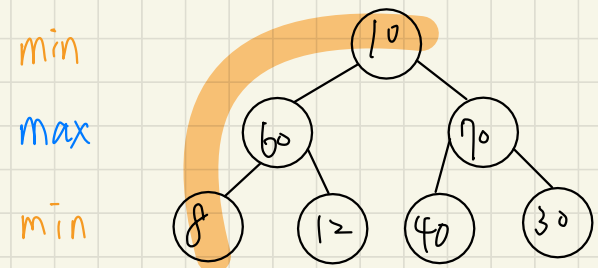
1. Decide which level -> min or max. $\left(level = ((int)floor(\log_2(i+1)) \% 2\right)$
2. check whether to swap with its parent.

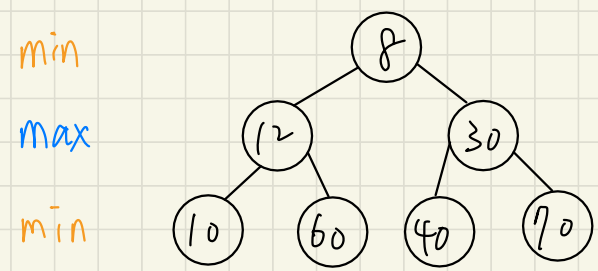




Ex. Input order : 10 12 30 8 60 40 70.







- Where is the grandparent of item[i]?

  if((i-1/2)>2) // 0, 1, 2 無子孫關係).

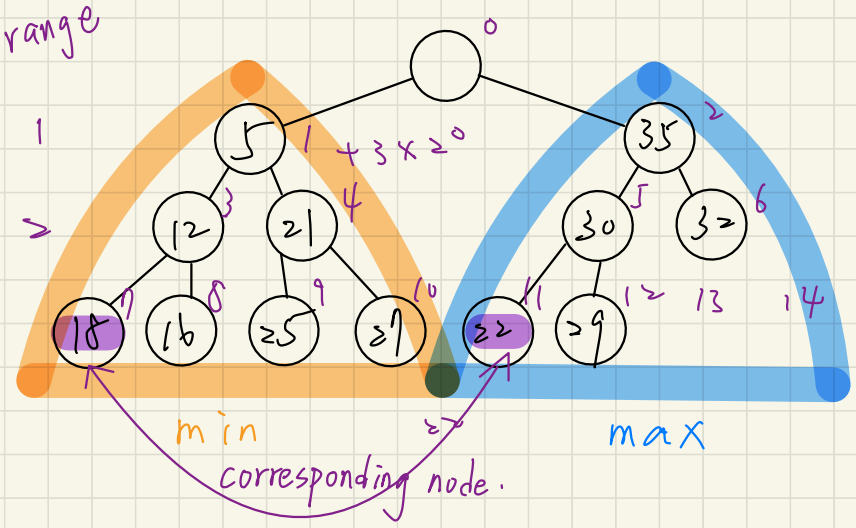
  grandparent = ((i-3)/4);

- Where are the grandchildren of items[i]?

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

— Double-ended Heap. (DEAP).

1. Examine the corresponding nodes : left < right.
2. Rehcap neccessary.



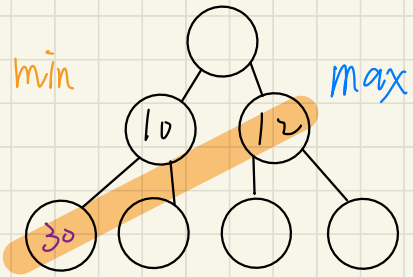
level 0 (2^0)

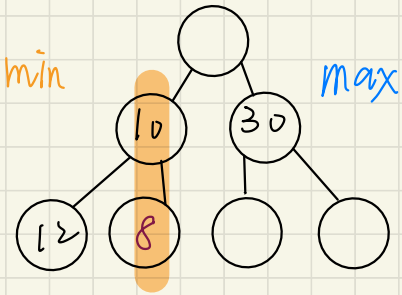
level 1 (2^1)

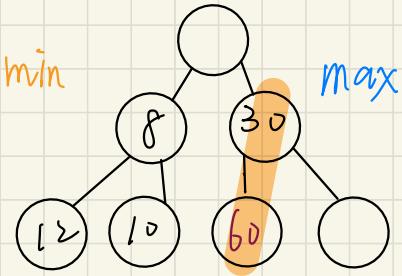
level 2 (2^2)

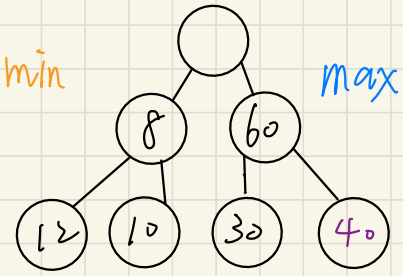
if i = 4 => level 3 => range = 2

ex. Input order : 10, 12, 30, 8, 60, 40









DEAP: delete the smallest.

1. Replace the root of min-heap with the last element.
2. ReheapDown if necessary.
3. Examine the corresponding nodes: left < right

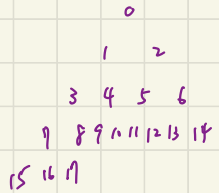
DEAP: delete the largest.

1. Replace the root of max-heap with the last element.
2. ReheapDown if necessary.
3. Examine the corresponding nodes: left < right

How?

$2^{i-1} < n < 2^i \quad (i = \lceil \log_2 (n+1) \rceil + 1)$.

$right = n + \lfloor (2^i - 2^{i-1}) / 2 \rfloor$.



Unit 3.

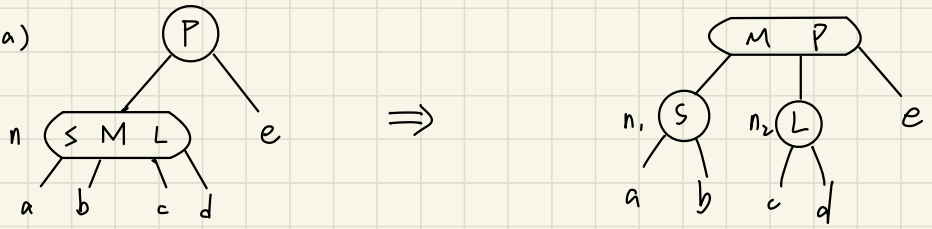
- 2 - node
  - S > the left child's search key(s)
  - S < the right child's search key(s)

search key<S S< search keys

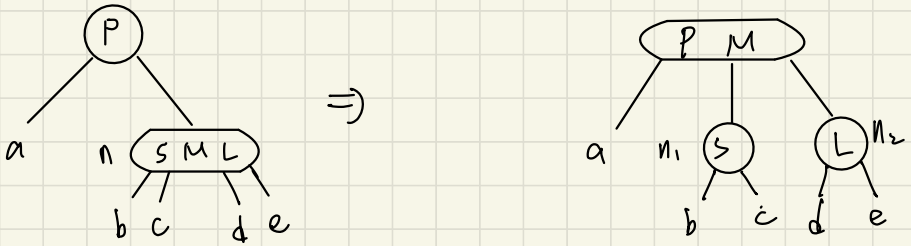
- 3- node
  - S > the left child's search key(s).
  - S < the middle child's search key(s).
  - L > the middle child's search key(s)
  - L < the right child's search key(s).

△ Inserting into a 2-3 Tree.

(a)

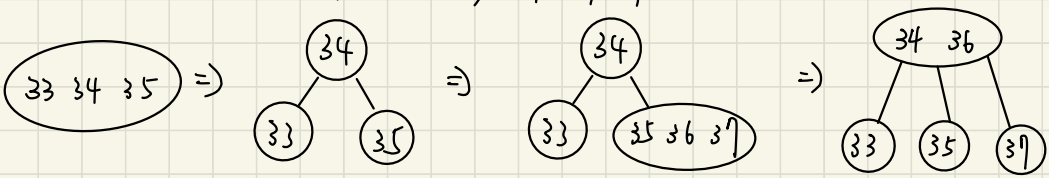


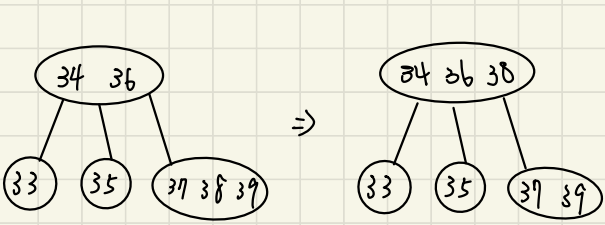
(b)



※ a container can only contain 2 data at most

ex: Insert 33, 34, 35, 36, 37, 38, 39.



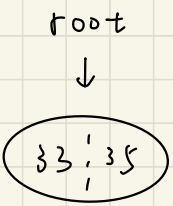


△ Insert 33, 35, 37, 39, 34, 36, 38

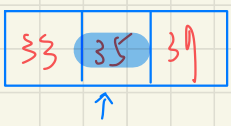
: key[1] (put 2 key at most).

→ : ptr[2] (pointer require one more than key).

(a)

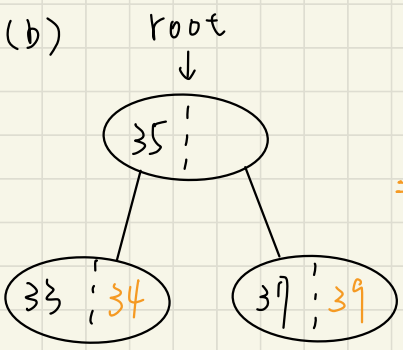


⇒ once 37 has pushed to the array, compare these keys.



root point to the middle and new a node.

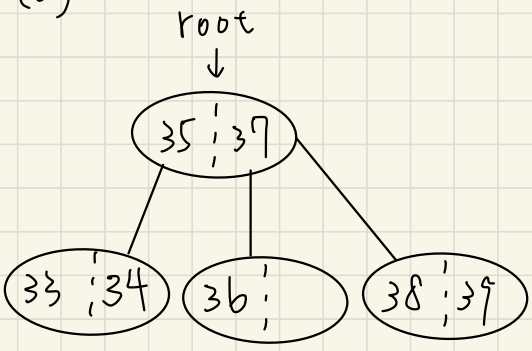
(b)



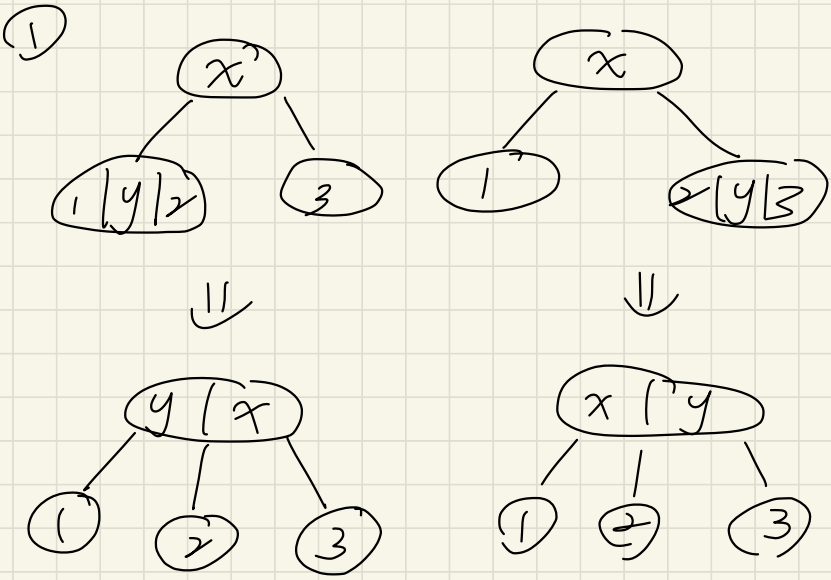
⇒ if there is any empty, push !

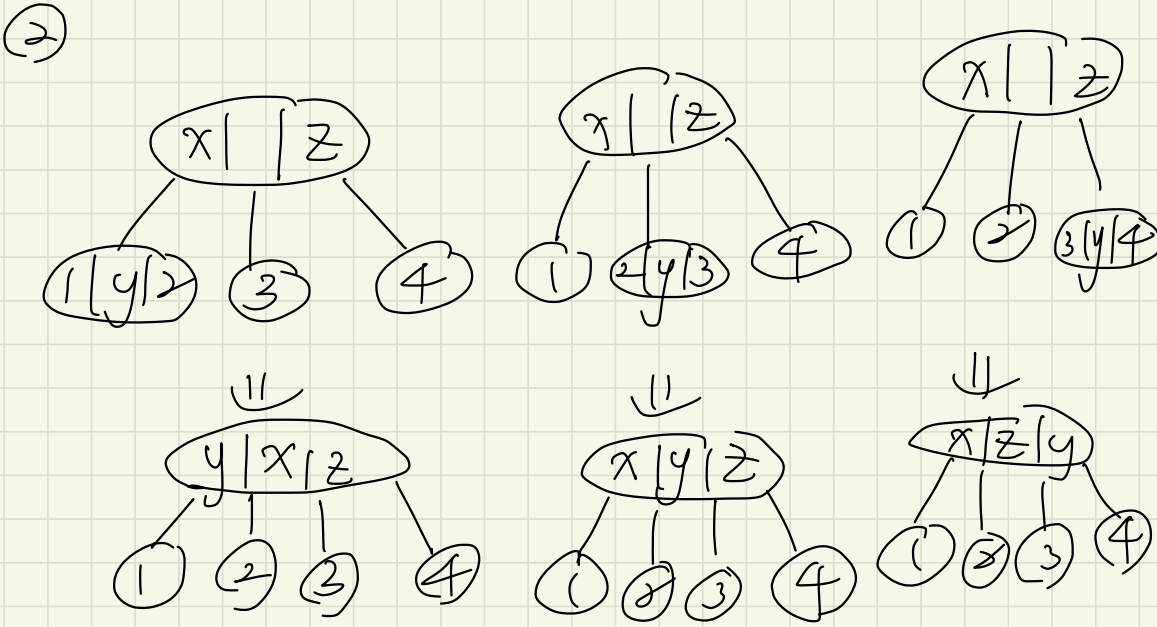
⇒ once 36 has pushed to the array, compare these keys.

(c).



⇒ done !


1
x
1 | y | 2
3
x
1
2 | y | 3
y | x
1
2
3
x | y
1
2
3


2
x | | z
1 | y | 2
3
4
y | | z
1
2 | y | 3
4
x | | z
1
2
3 | y | 4
y | x | z
1
2
3
4
x | y | z
1
2
3
4
x | z | y
1
2
3
4