* worst-case of Remove-Largest ? O(h) with h being height of T
  so can be O(n) for n nodes

③ how about requring a complete binary tree ?
$$O(h) = O(\log n)$$

called max-heap
but need to check how to maintain.

* Heap-Remove-Largest (T) {

```
    compare LastNode -> key with T -> left -> key and T -> right -> key
    if (LastNode -> key largest) {
        replace T -> key, T -> data w/ LastNode -> key / data;
    }
    otherwise {
        replace T -> key, T -> data w/ Child -> key / data;
        Heap-Remove-Largest (Child, Candidate)
    }
}
```

move back to the root, and trickle down

* Heap-Insert (T, Current) {

```
    while ( Current -> key > Current -> parent -> key ) {
        swap Current and Current -> parent;
        Current = Current -> parent;
    }
}
```

put in the back, and bubble up

* note: complete binary tree can be packed in an array
  ⇒ max-heap is essentially a special array
  (not completely ordered, but follow some rules)

Subject : .... No. : 5-12 Date : ..../........./..........

* case 2 : $\boxed{??}$ = k

e.g { keys are words / data are their explanations } dictionary

binary search revisited

Bin-Search (k, RangeL , RangeR) {   (k, T)

middle = $\boxed{\cdots}$ $\boxed{\text{root of tree}}$

if (k < middle)

Bin-Search (k, RangeL , RangeM −1) ; $\boxed{\text{left subtree}}$

else if (k > middle)

Bin-Search (k, RangeM +1, Range R) ; $\boxed{\text{right subtree}}$

else

return location of middle;

}

* need   left subtree < root < right subtree

called binary search tree

* worst-case search time : O(h) w/ h being height of tree

* insert (also O(h))

if (k < middle)

insert to left-subtree   (usually assume no duplicated keys)

else if (k > middle)

insert to right-subree

* delete (also O(h))

leaf : simple

one child : simple

two children : take right-most decendent of left-subtree as root

* join, split : READING ASSIGNMENT

* good binary search tree : balanced (h = O(log n)) , { randomly insert : yes in average / in practice : not always sure

challenge : maintain good binary search tree w/ still efficient insert / delete

* heap : specially arranged complete binary tree
w/ application in simple priority queue

BST : specially arranged binary tree
w/ application in search (dictionary)

selection : general complete binary tree to process "tournament" data
w/ application in merging ordered lists

* l_1 : 9 8 7 3 1
l_2 : 10 6 5 2

how to merge to
10 9 8 7 6 5 3 2 1 ?

output (max (head(l_1), head(l_2)))
remove the max from the associated list

O(N) for N elements (if removal is O(1))

* l_1, l_2, l_3, l_4 ?

output (max (head(l_1), head(l_2), head(l_3), head(l_4)))
remove ...

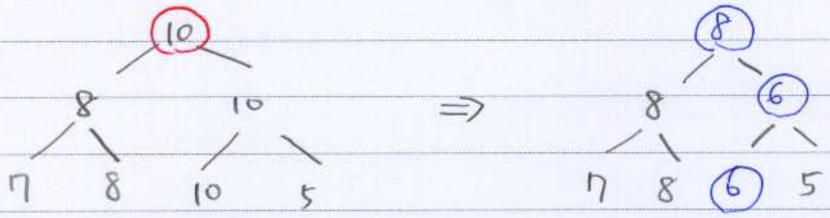
O(N · time(max))
for k lists, time(max) is O(k) ⇒ O(Nk)
for naive implementation

* l_1 7
l_2 8
l_3 10 6
l_4 5

naive:
$\boxed{(7, 8)}$, (8, 10), (10, 5) ⇒ 10
$\boxed{(7, 8)}$, (8, 6), (8, 5) ⇒ 8
(7, ·), (7, 6), (7, 5) ⇒ 7
⋮

repeatedly checked

* save time w/ tournaments



only (at most) the path from the new element to the root needs to be updated

$O(h) = O(\log_2 k)$ to maintain and $O(1)$ to find max

$\Rightarrow O(N \log k)$ to merge k ordered lists w/ a total of N elements

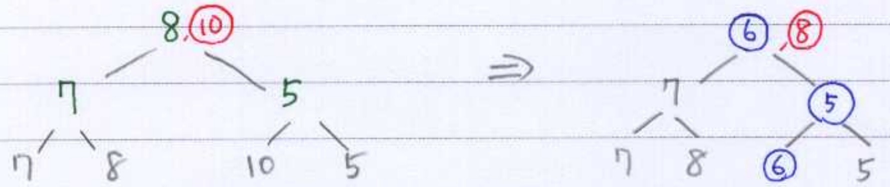
called max-winner tree (textbook: min-winner)

note: a bottom-up tree (leaf → root)

* the path from leaf-10 to root all stores leaf-10

to rematch, need to find "sibling" (e.g. 5, 8)

can simplify finding sibling by storing sibling in non-leaf nodes + overall winner



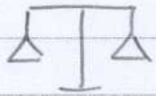
called (max-) loser tree
i.e. sibling

* Section 5.9 : Forest (READING ASSIGNMENT)

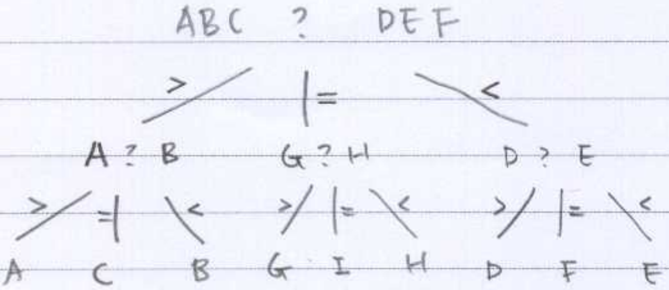
* balance game & trees

9 coins A B C D E F G H I

one of them heavier

two uses of balance 

how to find it ?



a complete trinary tree !

leaf: outcomes

non-leaf: conditions

* two uses of balance = two non-leaf levels

⇒ at most 9 possibilities

if 10 coins (outcomes), provably impossible

if 9 coins, need to physically check if reasonable (like above)

* brainstorm:

12 coins, 1 of them heavier or lighter, 3 uses of balance

13 " want to know heavier

14 " or lighter

→ impossible, why?