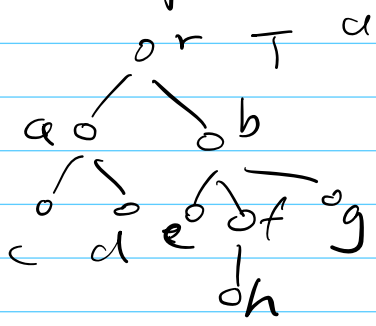
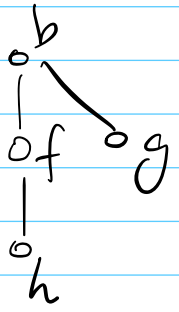
0. Warm up exercises:



(1) ancestors of h: r, b, f

(2) siblings of e: f, g

(3)  is this a subtree of T? X

b. Depth a a node: $dep(r) = 0$

$dep(v) = \#$ of edges from r to v

$dep(d) = 2$

$dep(h) = 3$.

Height of a tree: $\max_v dep(v)$

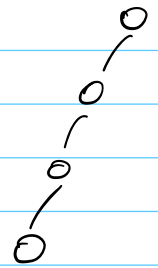
$Height(T) = dep(h) = 3$

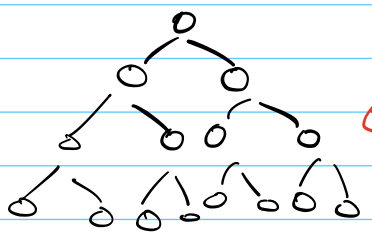
c. Consider a binary tree of height K

how many leaves does it have?

at least 1 at most 2^K

K=3



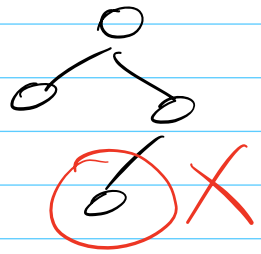


← Full & complete

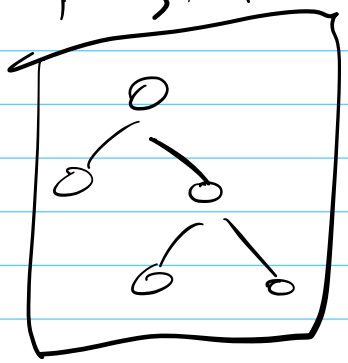
Full binary tree : every node has 0 OR 2 children.
(满二叉树)

vs,

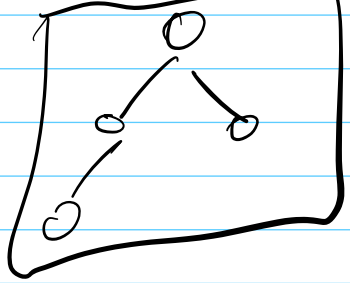
complete binary tree: completely filled except for last level & all nodes in last level are as far left as possible.
(完全二叉树)



Full, NOT complete

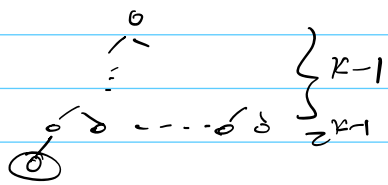


Complete, NOT Full



Q: Height k binary tree, how many leaves?

- Full B.T $\underline{\quad ? \quad} \leq$ #leaves $\leq 2^k$
- Complete B.T. $\underline{2^{k-1}+1} \leq$ $\leq$



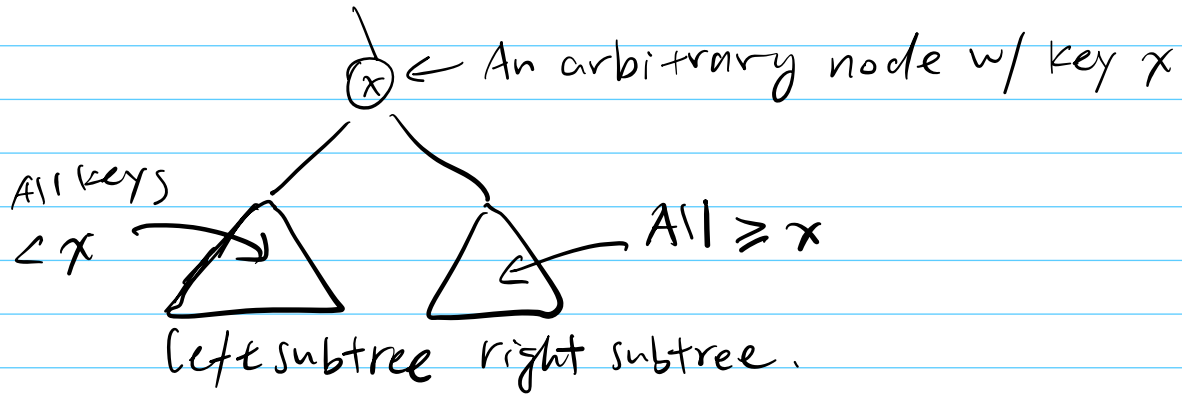
1. BST (Binary Search Tree) Cont'd

: Sorted array + fast (log) ins / del

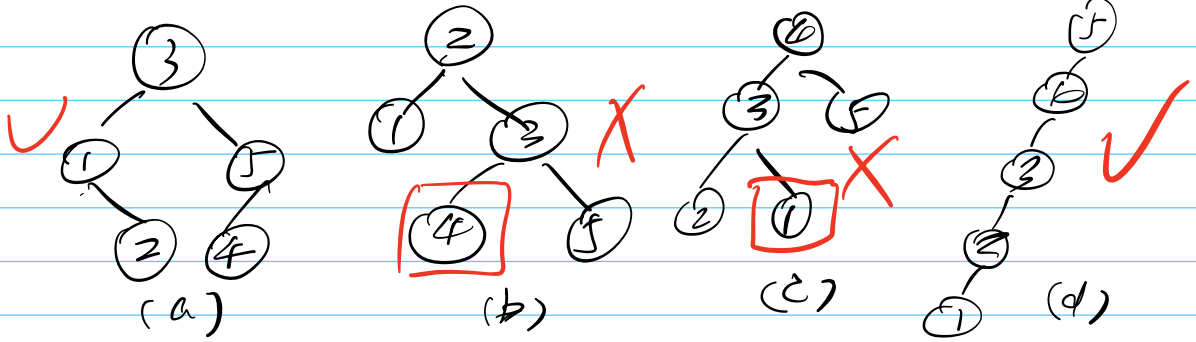
op's	Sorted Array	BST* (Balanced)	
Search	$\log n$	$\log n$	
select	1	$\log n$	} price
min/max	1	$\log n$	} price
Rank	$\log n$	$\log n$	
output (in sorted order)	n	n	
ins	n	$\log n$	} gain ✗
del	n	$\log n$	} gain ✗

- BST: structure.

Search Tree property.



- Ex: which ones are valid BSTs?



Height matters!

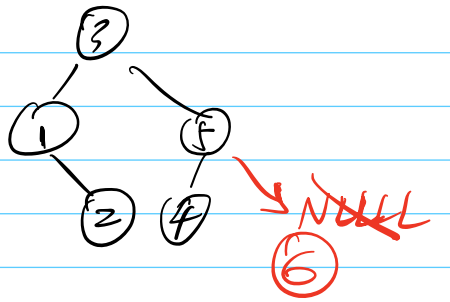
search(z) in (a), (d)

Ex: Binary tree T w/ n nodes.

$\log_2 n \leq \text{Height}(T) \leq n-1$

b. Supported op's.

- Search: key k in T
  - start at root
  - traverse left/right child
    - k < current key
    - k > current key
  - return node w/ key k OR NULL.



? 6>3 → >5 → NULL

6 ∉ T.

Ins(6)

Time: Θ(height)

- Insert: a new key k in T
  - search for k (fail)
  - rewire final NULL pointer to new node

Time: Θ(height)

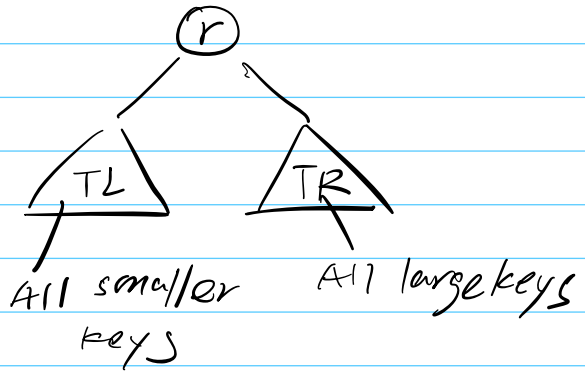
- min/max:
  - compute min/max key in T.
  - start at root
  - follow left child pointer for min (right for max)

until cannot anymore.
- return last key found.

Time: T w/ n nodes.

1 $\log_2 n$ n (height)

- output (in order)
  - recurse TL
  - output r's key
  - recurse on TR.

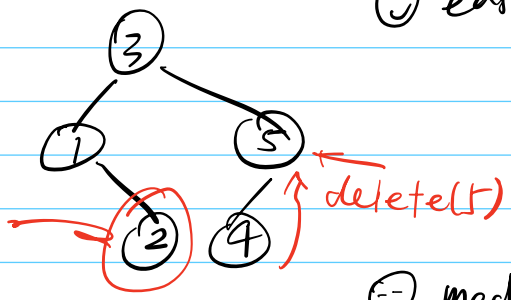


Time: $\Omega(n)$
$\Theta(n)$?

- Delete:

delete a key k from a search tree.

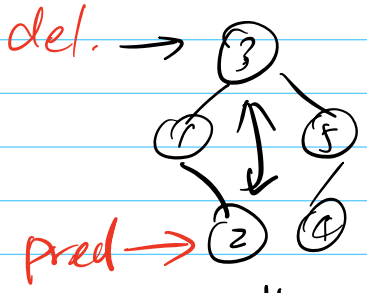
- search(k) k=2



☺ easy case: k's node has no children.

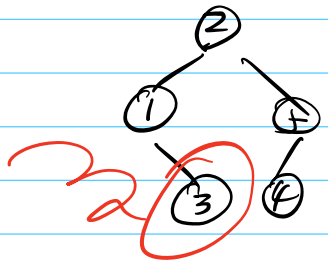
😐 medium case: k's node has 1 child
→ unique child assumes position previously held by k's node

③ difficult case: k's node has 2 children



Idea: reduce to easy cases above

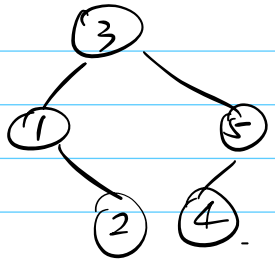
- Find k's predecessor i
- swap k & i



⇒ Delete(k) becomes one of the 2 easy cases above.

Time: Search + Find Predecessor

- Pred(k): predecessor of key k



Pred(3) → 2

Pred(2) → 1 Pred(4) = 3

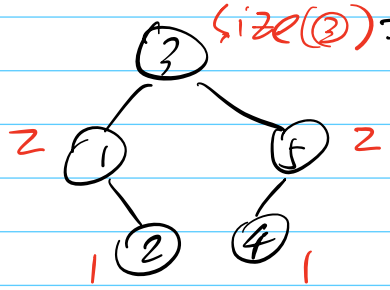
- easy case: k's left subtree non-empty.
  return max key in left subtree.
- o.w.: follow parent ptrs until you get to a key less than k.
  i.e. tracing upward, 1st left turn

Time: Θ(height)

- Rank(k): #nodes w/ key $\leq k$.

Idea: Augment BST at every node x

size(x) := # of nodes in subtree rooted at x (including x).



- start at root.
- if $k >$ current key, increment rank by size? move to right child.
- o.w move to left child.

Time: Θ(height)

c. Balanced Search Tree.

Goal: ensure that height is always (log n) [Best possible]

✓ mult. soln's exist: Red-Black tree.
AVL tree,
B tree ...

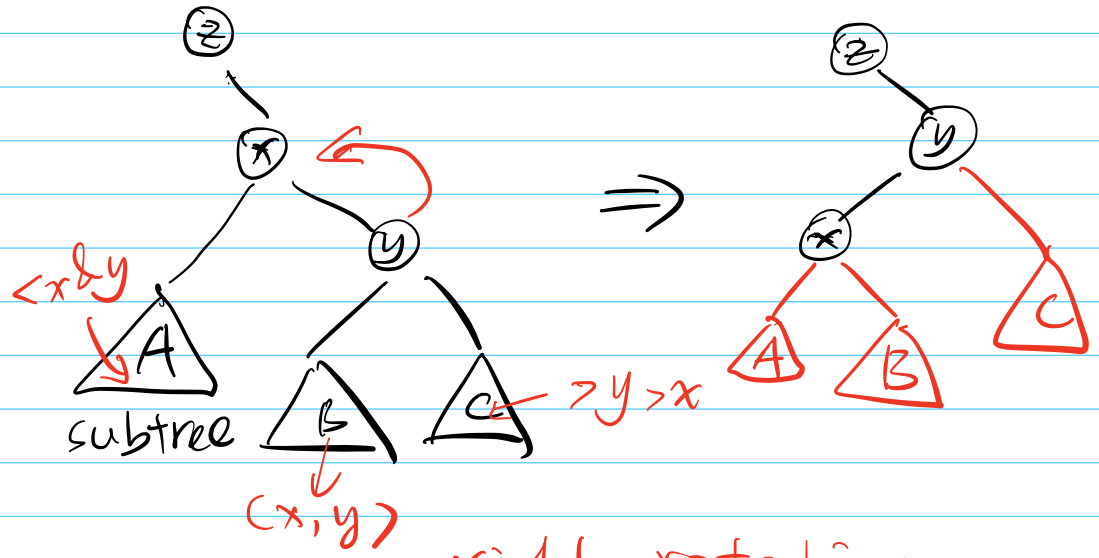
😐: details are tricky!

😊 You don't need to code up from scratch.

Idea: Rotation

locally rebalance subtrees at a node. in O(1) time

Left rotation



(x, y)

right rotation