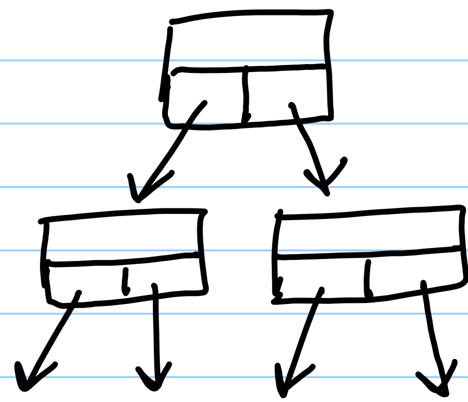
Trees (ch. 12, appendix B.5.3)

Matrices 28.1 - 28.2 in book.

Note Title 10/30/2007

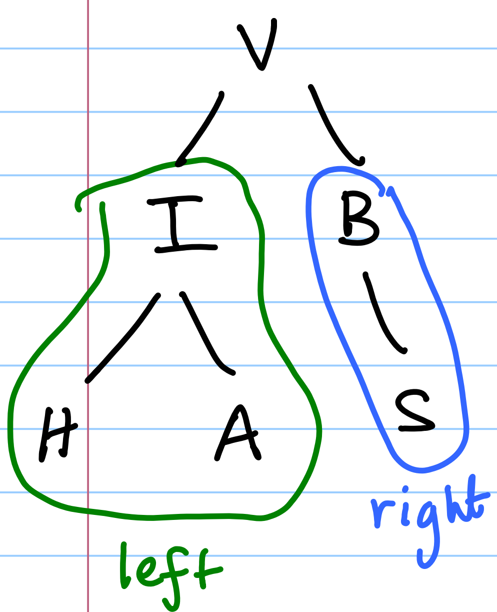
Binary Tree



each node has at most one left and one right child.

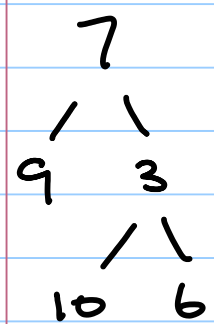
If you need to search in the tree for an item, you have to look at all the items.
(traverse every node)

Given a pointer to the root, you can search with



depth-first search root – children	VIHABS or VBSIAH ...
breadth first search root - level 1 - level 2 ...	VIBHAS or VBISAH ...
preorder root - left - right	VIHABS
postorder left - right - root	HAISBV
in order left - root - right	HIAVBS

Search for an item in a generic binary tree



generic binary tree

Worst case – $O(n)$
Best case – $O(1)$ ↙ find it right away

Expected case – $O(n)$

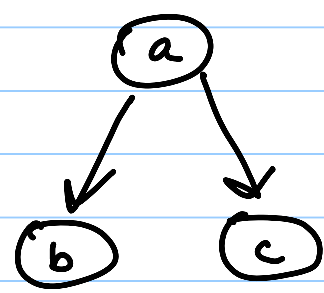
Find the min in a generic binary tree

- need to look at all elements

best case $O(n)$
worst case $O(n)$

Binary Search Tree

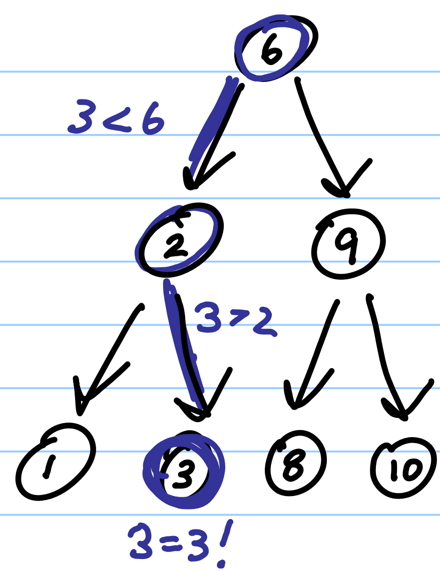
→ tree is sorted!



$b \leq a$
$c \geq a$

To find an element, you have to search but you <u>may</u> not need to traverse entire tree.

example search for 3

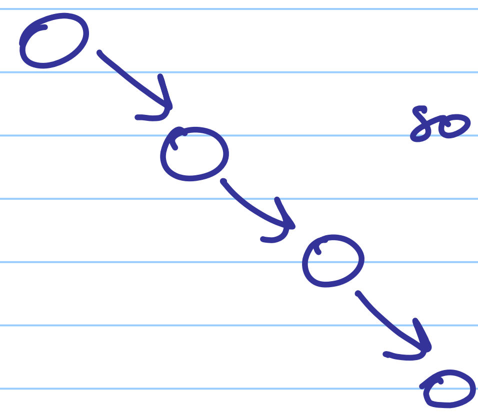


Because it is sorted, you only have to look at one node per level in your search...

worst case is O(# levels)

and # levels in worst case is with an unbalanced tree

unbalanced tree



so worst case O(n) even in binary search tree

Best case $O(1)$ as before
(find it right away)

Average case?

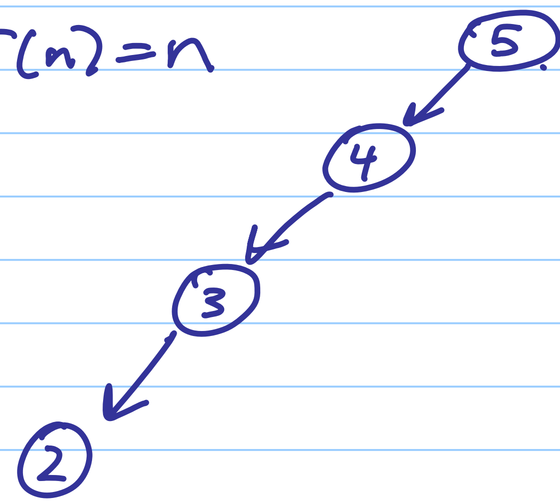
Really complex proof, but it turns out it is $O(\log n)$.

Find Min in a Binary Search Tree

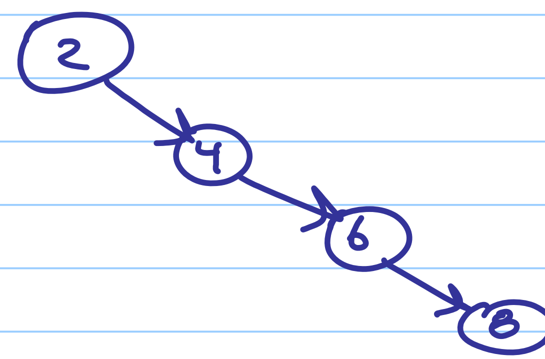
- find the left-most node.

```
BSTMin(Tree t) {
  while (t->left) {
     t = t->left;
  }
  return t;
}
```

worst case: $T(n) = n$



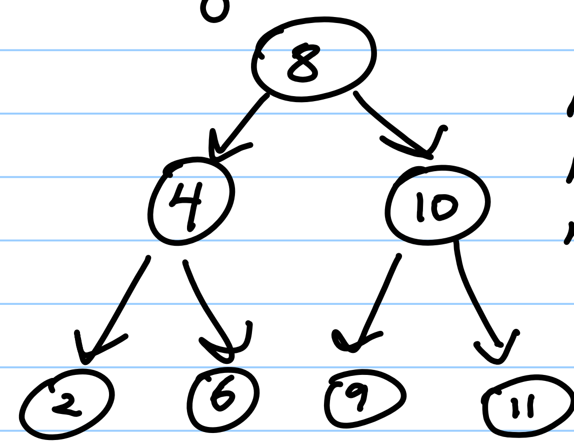
best case: $T(n) = 1$



Binary Search Tree :: Successor

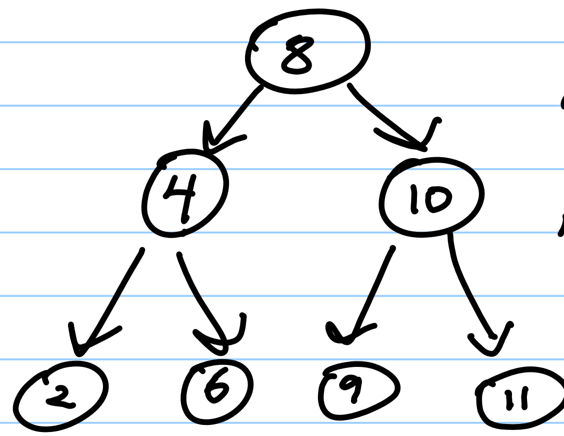
- given one element in a tree, how do you find the next largest element in the tree?
- assume distinct elements

Case I: there is a right subtree
next largest element is Min of right subtree.



next_largest(4) = 6
next_largest(8) = 9
next_largest(10) = 11

Case II : there is NO right subtree.
then next largest element is
lowest ancestor
whose left child is also an ancestor or self



next_largest(2) = 4

next_largest(9) = 10

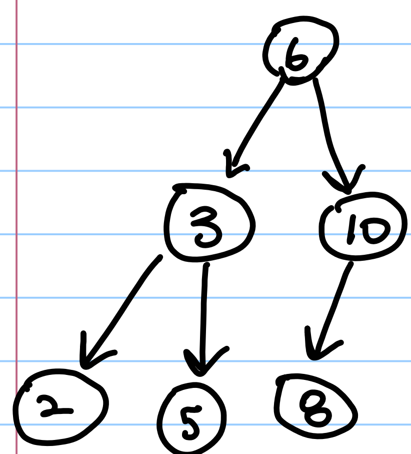
next_largest(6) = 8

next_largest(11) = nil

Runtime: O(# levels)
because we either go down the tree or up the tree.

Binary Search Tree ::Insert

assume tree is non-null.

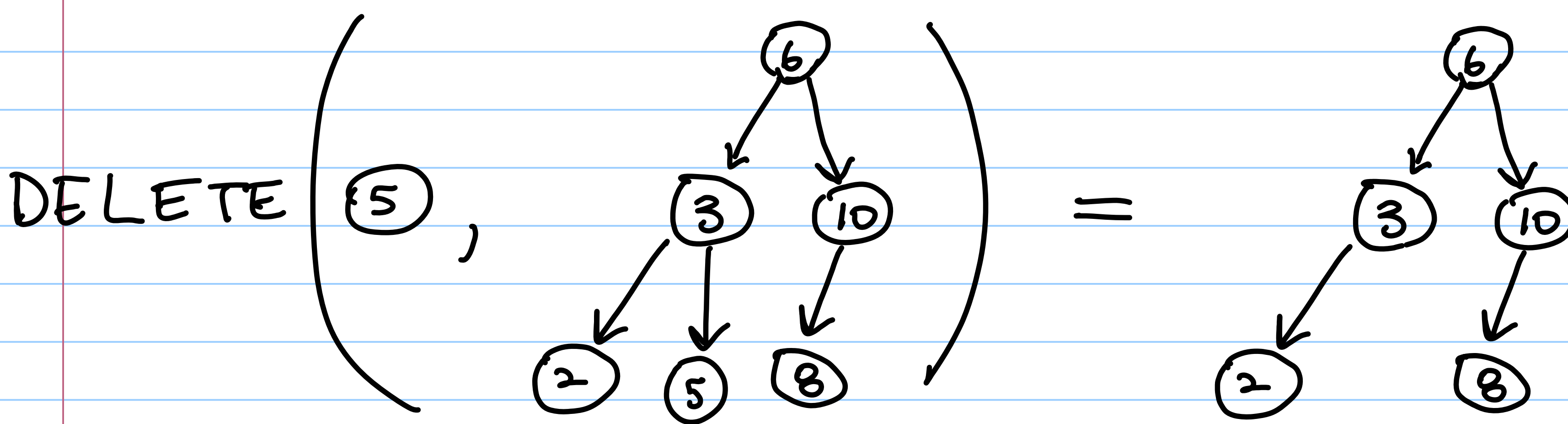


```
insert(Tree t, Node n) {

    NodeI'mAt   = t.root;
    while (NodeI'mAt  != null) {
        possibleParent = NodeI'mAt;
        if ( n.value < NodeI'mAt.value) {
            // go left
            NodeI'mAt = NodeI'mAt.left;
        }
        else {
            // go right
            NodeI'mAt = NodeI'mAt.right;
        }
    }
    add n as left or right child of possibleParent
}
```

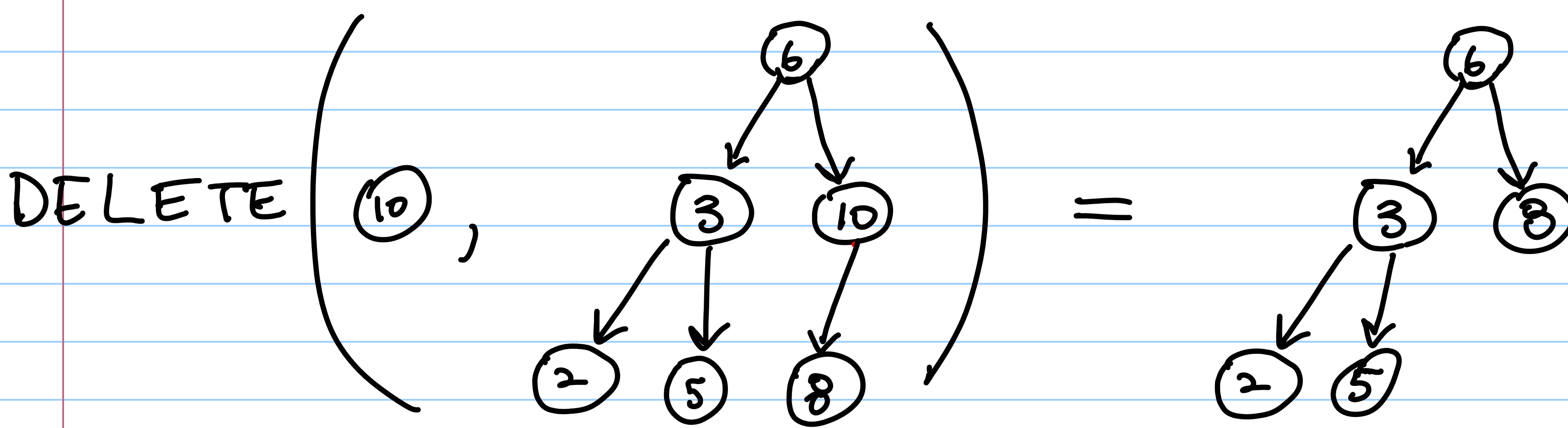
BinarySearchTree::Delete

Case 1: node to delete has no children
- just remove it



BinarySearchTree::Delete

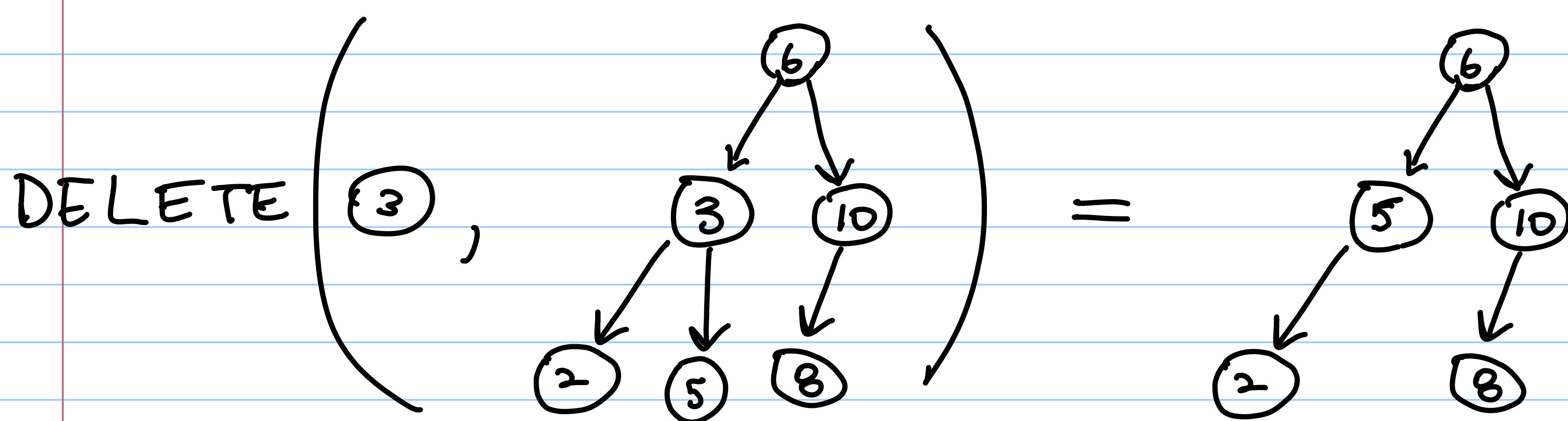
Case 2: node to delete has 1 child
- replace deleted node with child



BinarySearchTree::Delete

Case 3: node to delete has 2 children

- find deleted node's successor
  (will be in right subtree of deleted node, and will not have a left child)
- replace deleted node with successor



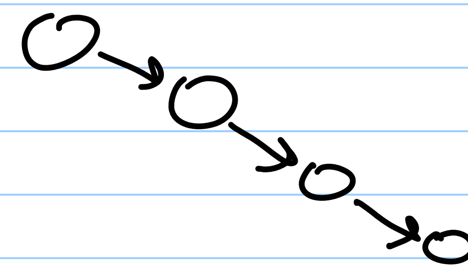
Runtime of Insert and Delete

Insert → $O(\#\text{ levels})$

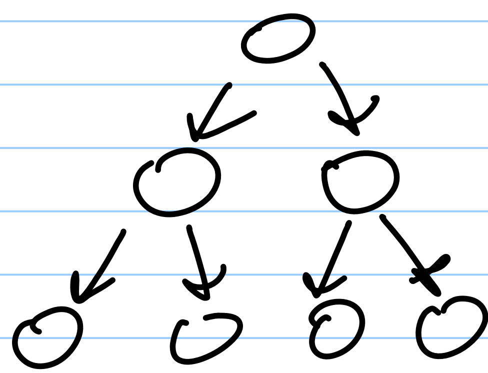
Delete
→ ① Find node to delete (search)
② in Worst-case, also need to find successor

both are $O(\#\text{ levels})$ so Delete is $O(\#\text{ levels})$

So many algorithms are $O(\# \text{ levels})$!
in Worst case, # levels = n



To minimize the # of levels in a tree,
we need a balanced tree.



How can we <u>ensure</u> good performance, without doing too much tree rebalancing work?

ргий мович
н- льский

and

Евгений Михайлович Ландис

came up with a balanced tree data structure in 1962. Published in English in Journal of Soviet Math, where their names were translated as

Georgii M. Adelson-Velskii and E.M. Landis.

we call their trees AVL trees. (why not АВЛ trees?)

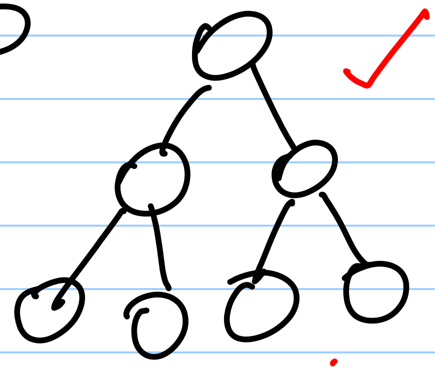
AVL Trees

tree height: maximum length of a path from root to any leaf

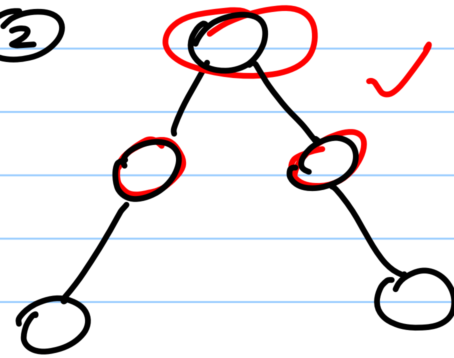
AVL rule: for any node,
$height_{left\ subtree}$ and $height_{right\ subtree}$
differ by at most ONE

Is this rule enough to guarantee the height is $O(\log n)$ in worst-case?

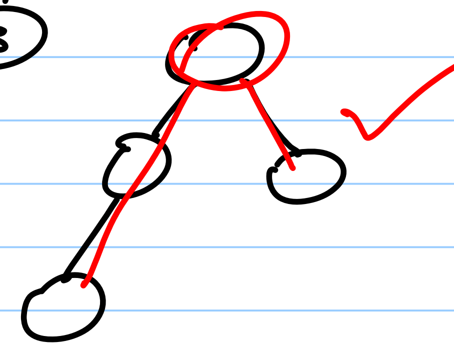
① AVL Tree?



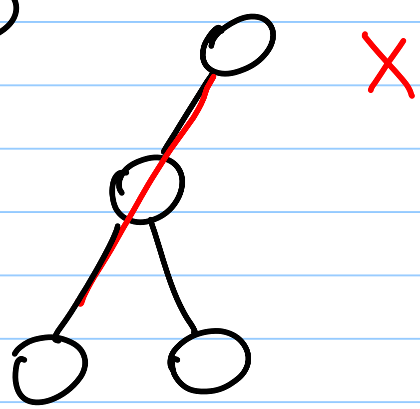
② AVL Tree?



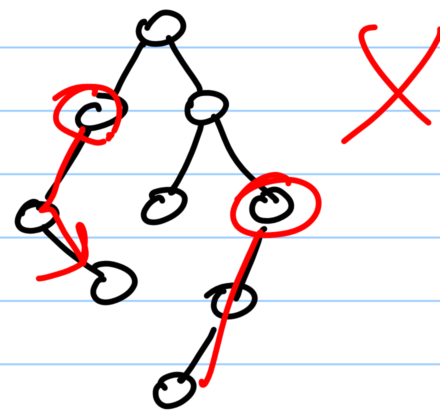
③ AVL Tree?



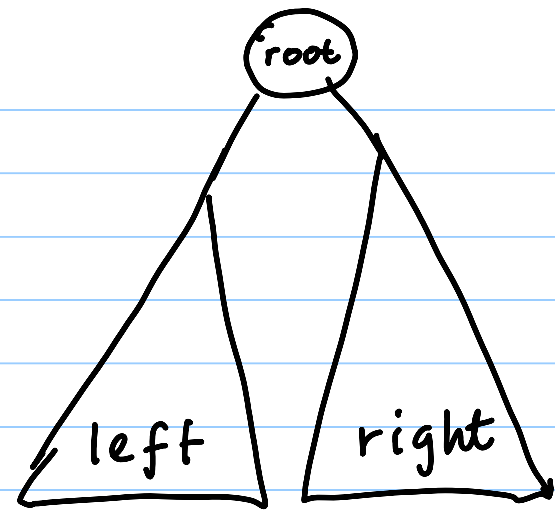
④ AVL Tree?



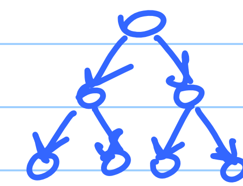
⑤ AVL Tree?



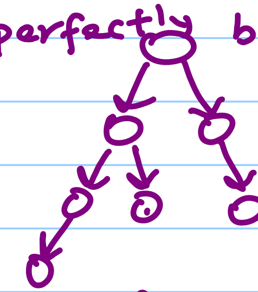
Does AVL rule guarantee height $O(\log n)$?



fully balanced tree: $h = \log_2 n$



but AVL tree? (not perfectly balanced)



is $h \in O(\log n)$?

Let # nodes in left subtree $= n_{left}$
nodes in right subtree $= n_{right}$

$n_{total} = n_{left} + n_{right} + 1$

(1) Because left and right subtree are AVL trees,

$$height_{left} = height_{right} \pm \{0, 1\}$$

worst case is when they are not even,
so let $height_{right} = height_{left} - 1$

or $h_{right} = h_{left} - 1$.

(2) $h_{total} = h_{left} + 1$

root has height h, left subtree has height $h-1$,
and right subtree has height $h-2$.

(3) So...

$$n = n_{left} + n_{right} + 1$$

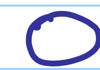
can be written as $n_h = n_{h-1} + n_{h-2} + 1$

worst case

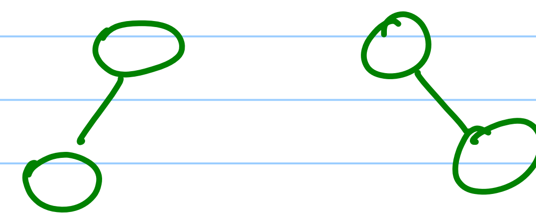
goal: is height $O(\log n)$?

$$n_h = n_{h-1} + n_{h-2} + 1$$

$n_0 = 1$

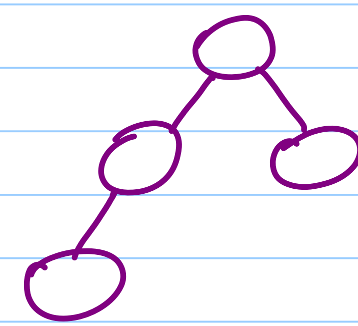


$n_1 = 1 + 0 + 1 = 2$



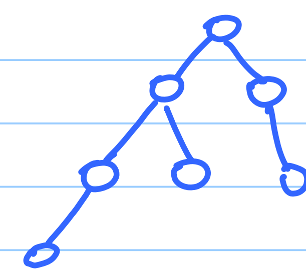
$n_2 = 2 + 1 + 1 = 4$

note: 4 is the minimum number of nodes needed to have a legal AVL tree of height 2



$n_3 = 4 + 2 + 1 = 7$

7 is the minimum # of nodes needed to have a legal AVL tree of height 3



$$n_h = n_{h-1} + n_{h-2} + 1$$

	0
	1
	1
$n_0 = 1$	2
$n_1 = 2$	3
$n_2 = 4$	5
$n_3 = 7$	8
$n_4 = 12$	13
$n_5 = 20$	21
$n_6 = 33$	34
⋮	⋮

Fibonacci numbers!

note:
minimum # of nodes needed to have a height h is worst case for that height because more nodes would mean the tree is <u>more</u> balanced...

$$n_h \geq \text{Fib}(h+3) - 1$$

$$n_h \geq \frac{\phi^{h+3} - \hat{\phi}^{h+3}}{\sqrt{5}} - 1$$

$$n_h \geq \frac{\phi^{h+3}}{\sqrt{5}} - \frac{\hat{\phi}^{h+3}}{\sqrt{5}} - 1$$

$$n_h > \frac{\phi^{h+3}}{\sqrt{5}} - 1 - 1$$

$$\sqrt{5}(n_h + 2) > \phi^{h+3}$$

$$\text{Fib}(i) = \frac{\phi^i - \hat{\phi}}{\sqrt{5}}$$

$$\phi = \frac{1+\sqrt{5}}{2} \quad \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

$$-1 < \hat{\phi} < 1$$

so $\frac{\hat{\phi}^{h+3}}{\sqrt{5}} << 1$

(either negative or very small positive)

$$\sqrt{5}(n_h + 2) > \phi^{h+3}$$

$$\log_\phi(\sqrt{5}(n_h + 2)) > h + 3$$

$$h < \log_\phi \sqrt{5} + \log_\phi(n_h + 2) - 3$$

$$h < \log_\phi(n_h + 2) - 1.33$$

$h \in O(\log n)$ $\qquad \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$

is $h \in \Omega(\log n)$? yes, because the best case is a complete balanced tree

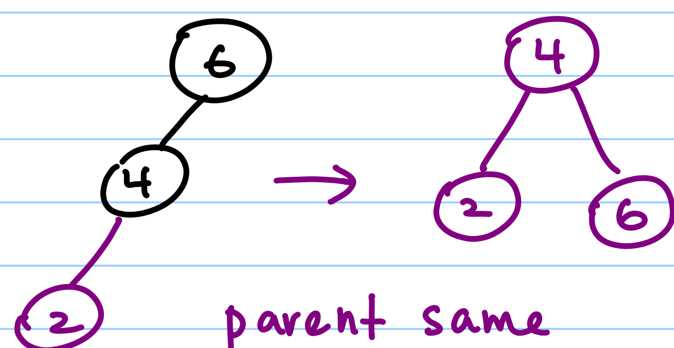
so $h \in \Theta(\log n)$

AVL Tree Runtimes

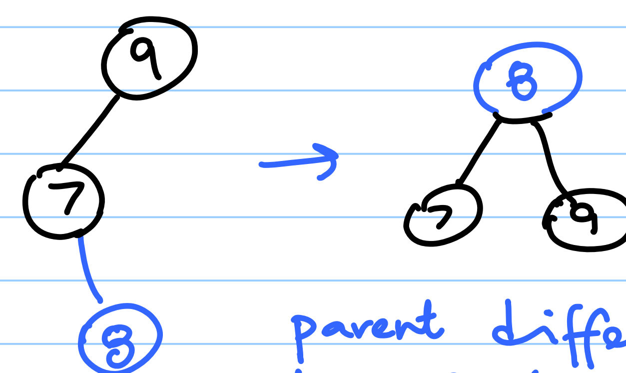
Search: $O(\#\text{ levels})$ like always

but # levels is $\Theta(\log n)$

Insert: insert as before
but then rebalance if AVL rule broken



parent same type of child as new node



parent different type of child than new node

Insert : $O(\log n)$ to search

$O(1)$ to do the insertion

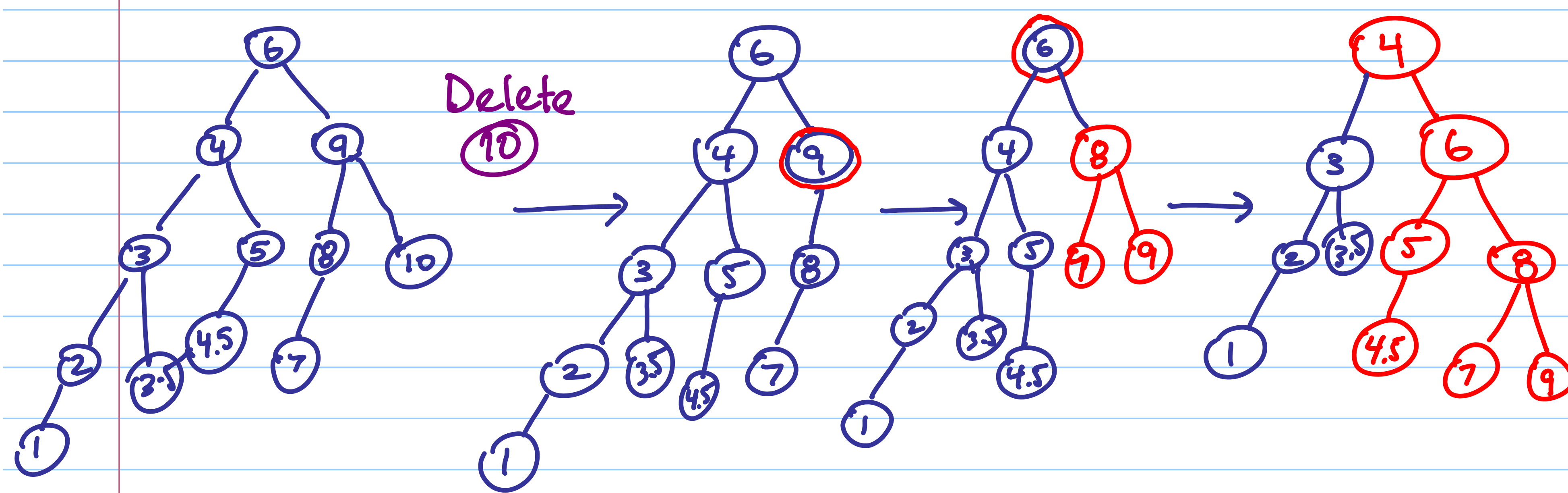
Traverse path from new node to root to find where to rebalance, so $O(\log n)$ to rebalance.

Total : $O(\log n)$.

AVL :: Delete

Delete as before, but then you may need to rebalance.

- again a $O(\log n)$ path traversed from deleted node to root to do rebalancing.



Traverse from 10 to root to rebalance.

Runtime of Delete

$O(\log n)$ to the delete.

then traverse path from deleted node to root to find rebalanceing candidates
→ $O(\log n)$
($O(1)$ work to do each rebalancing)

Total: $O(\log n)$ to delete

Other Binary Search Trees

① Red/black trees (ch. 13 of book)

paths alternate between black and red nodes

new nodes are red. then you rotate nodes if you need to.

Insertion / Deletion: $O(\log n)$

Height: At most $2\log(n+1)$.

② Splay Trees

- search target moved to root
  node just inserted moved to root

- not balanced, so worst case search is $O(n)$
  but AMORTIZED search is $O(\log n)$

Amortized Analysis,

time required to perform a <u>sequence</u> of operations is averaged.

This is not the performance of the expected case or a "typical case"

but the average of a sequence of operations' time in worst-case.

Good when the algorithm involves work done in one run to optimize a later run.