AUBURN UNIVERSITY

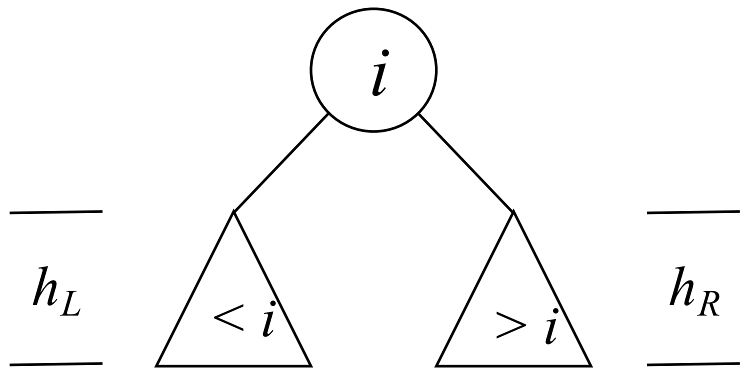
SAMUEL GINN COLLEGE OF ENGINEERING

AVL Trees

AVL trees

An AVL tree is a **binary search tree**

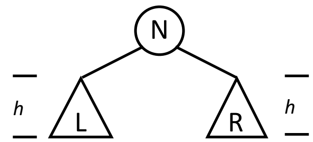
in which the heights of the left and right subtree of *every* node differ by at most 1.

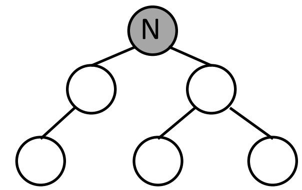


$$|h_R - h_L| \leq 1$$

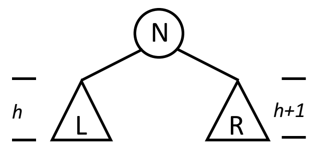
Structural possibilities

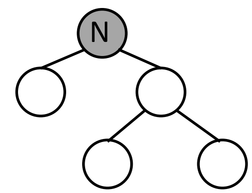
Equal heights



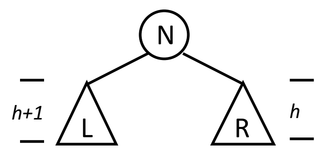


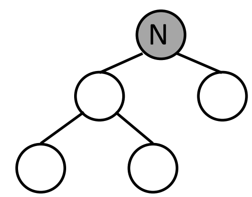
Right is 1 level taller





Left is 1 level taller





Balance factors

Every node in an AVL tree has a **balance factor**.

$$bf_N = h_R - h_L$$

! 

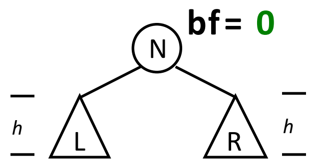
Remember to subtract heights, not balance factors.

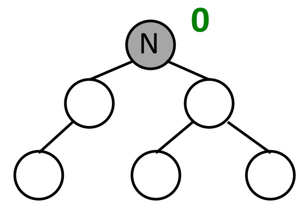
! 

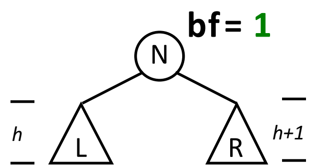
Some texts counts path lengths differently from me.

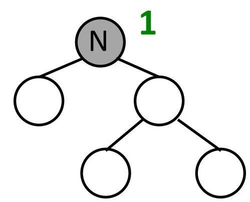
! 

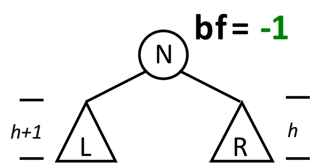
Balance factors are sometimes computed as h_L - h_R.

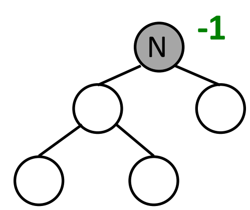






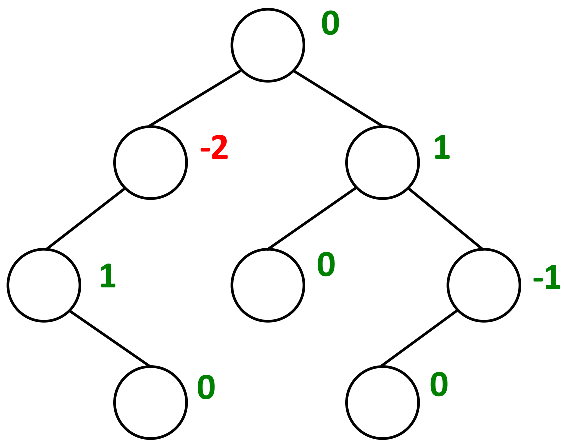






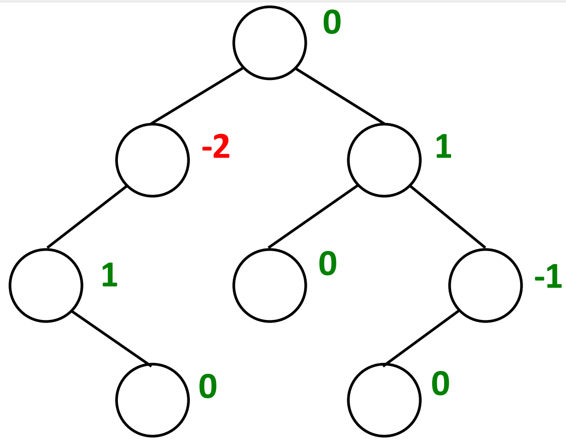
Balance factors

NOT an AVL Tree

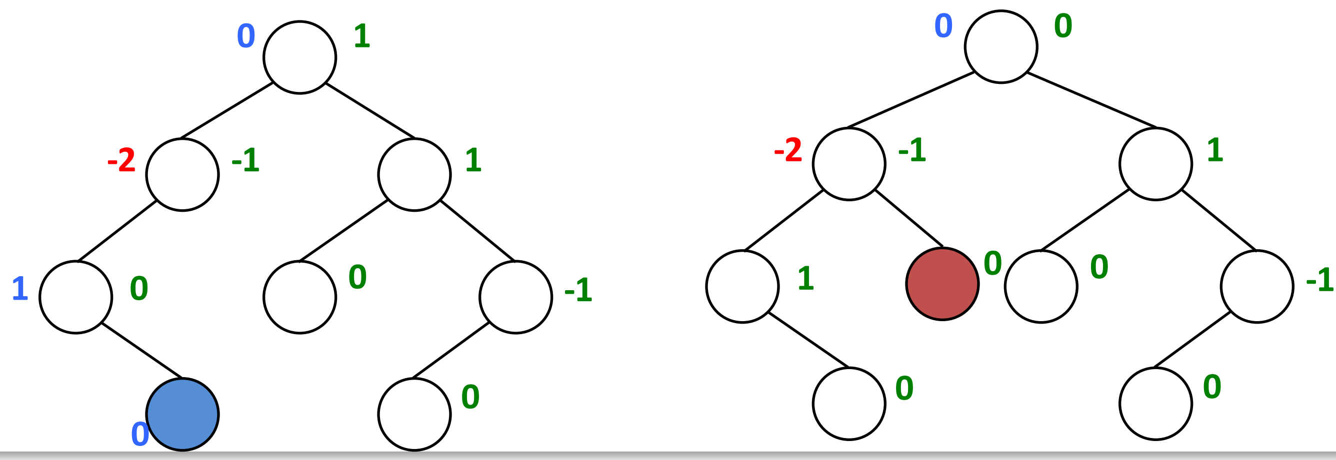


Balance factors

NOT an AVL Tree



But it could have been one ...



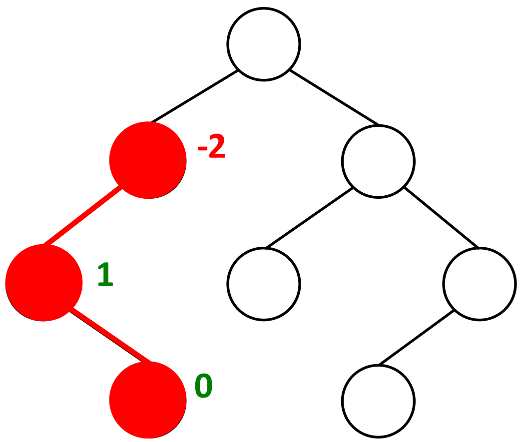
Rebalancing

Rebalancing

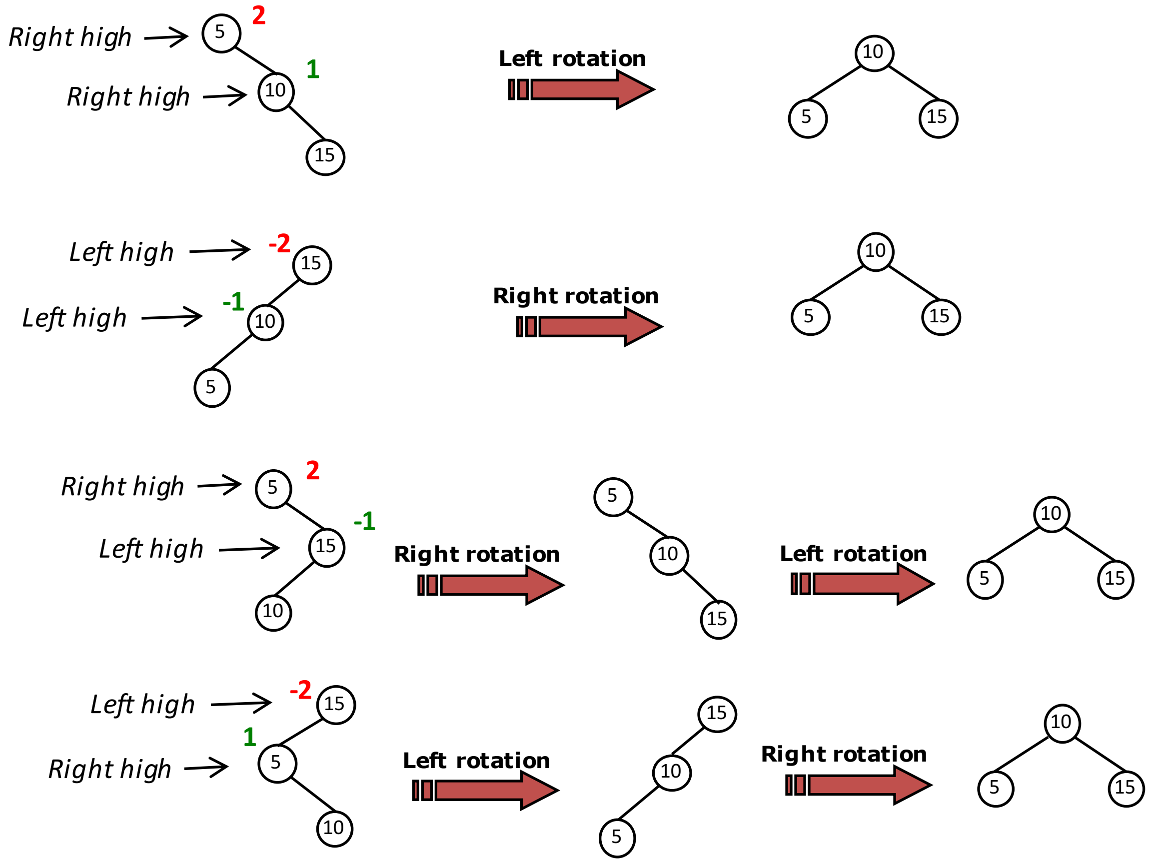
A bf of ±2 means that the subtree rooted at that node is out of balance.

Balance will be restored by subtree rotations.

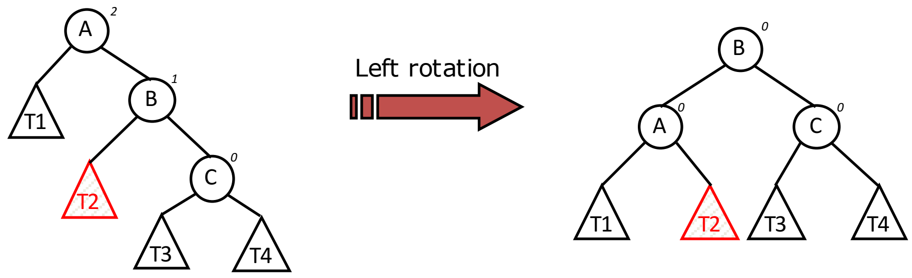
All rotations will occur in the context of a 3-node neighborhood.

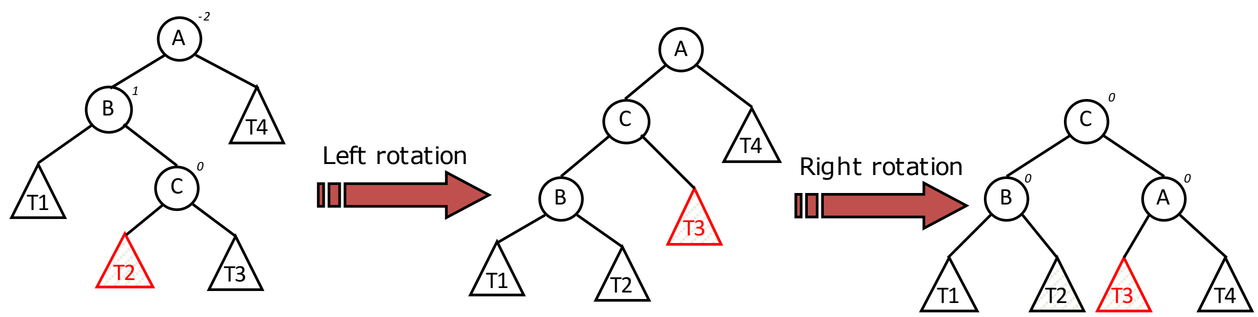


Rebalancing



Rebalancing

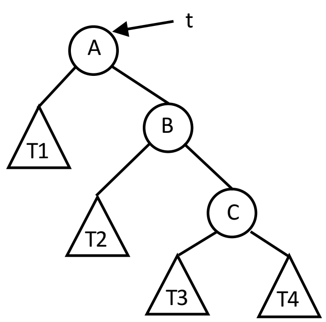




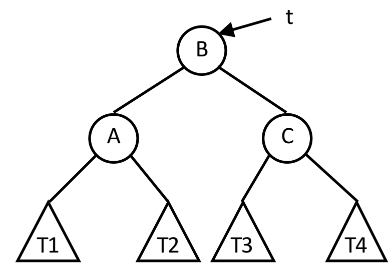
Coding rotations

`t = rotateLeft(t);`

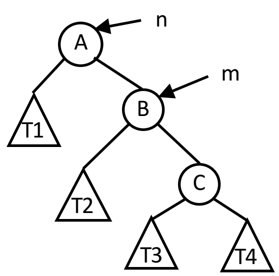
Left rotation around t



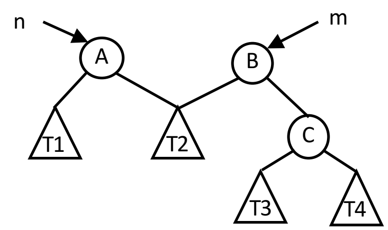

```
public  BTN  rotateLeft(BTN   n)
{
   BTN  m = n.right;
   n.right  = m.left;
   m.left  = n;
   return  m;
}
```



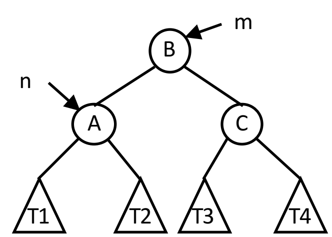
`BTN  m = n.right;`



`n.right  = m.left;`



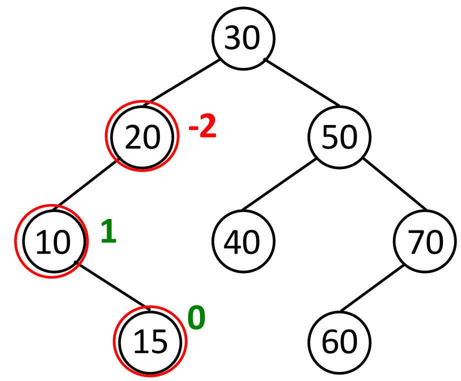
`m.left  = n;`



Adding values

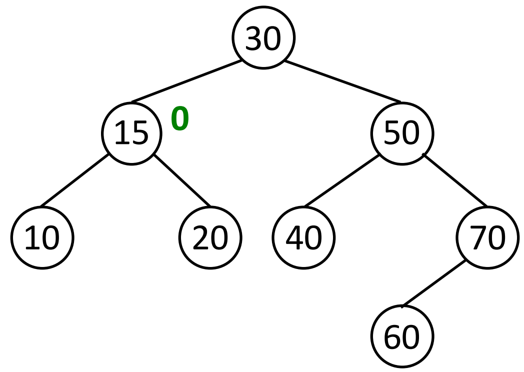
Adding values

Use the standard BST insertion algorithm to insert the new node. (Ex: 15)



Beginning with the node just inserted, walk the reverse path back toward the root, recalculating balance factors.

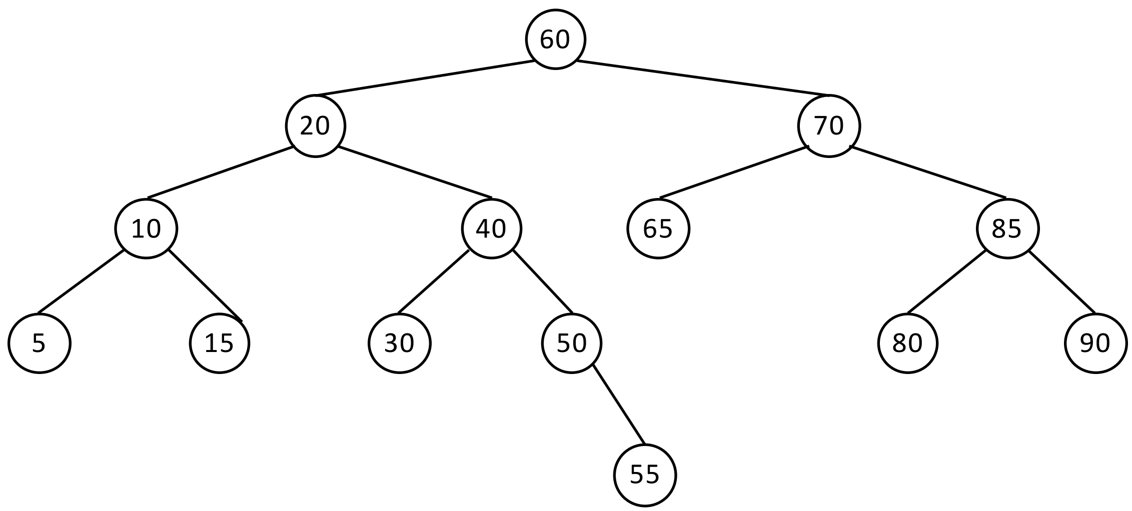
Stop at the first (lowest) node that has a balance factor of ±2. This node roots the 3-node neighborhood that will be rotated.



At most one rebalancing operation will be required per insertion.

Adding values

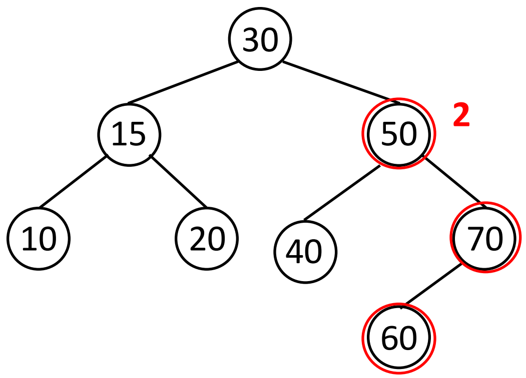
Insert: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



Removing values

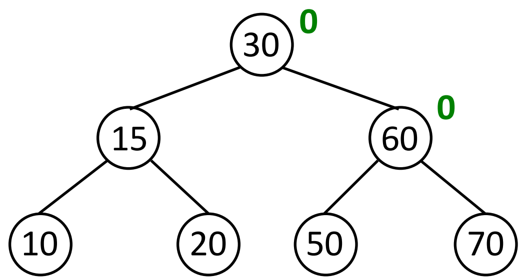
Removing values

Use the standard BST deletion algorithm to delete the element. Ex: 40



Beginning at the *point of deletion*, walk the reverse path back toward the root, recalculating balance factors.

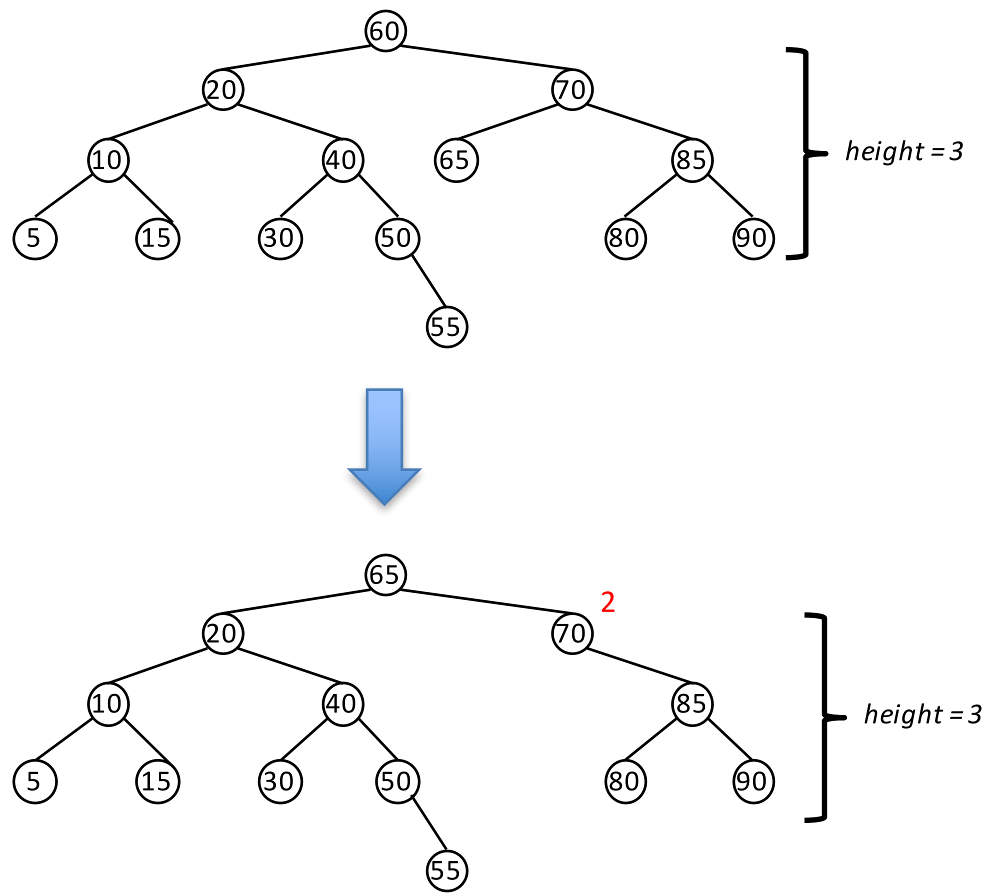
Stop at the first (lowest) node that has a balance factor of ±2. This node roots the 3-node neighborhood that will be rotated.



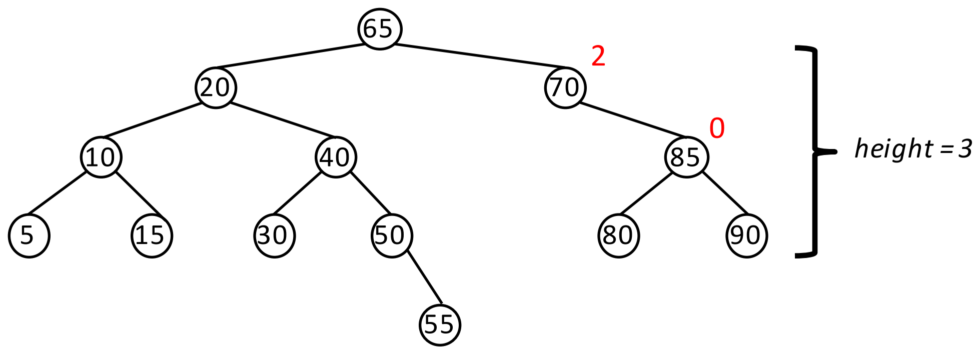
Multiple rebalancing operations may be required per deletion, so the reverse walk must go to the root each time.

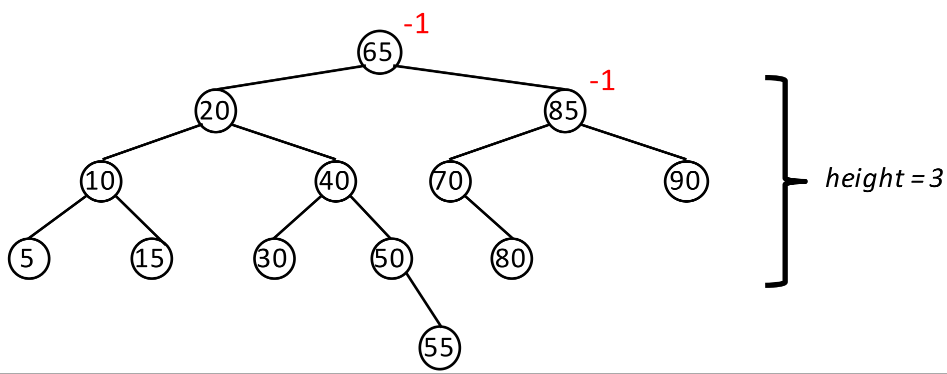
Removing values

Delete 60:
(use successor)



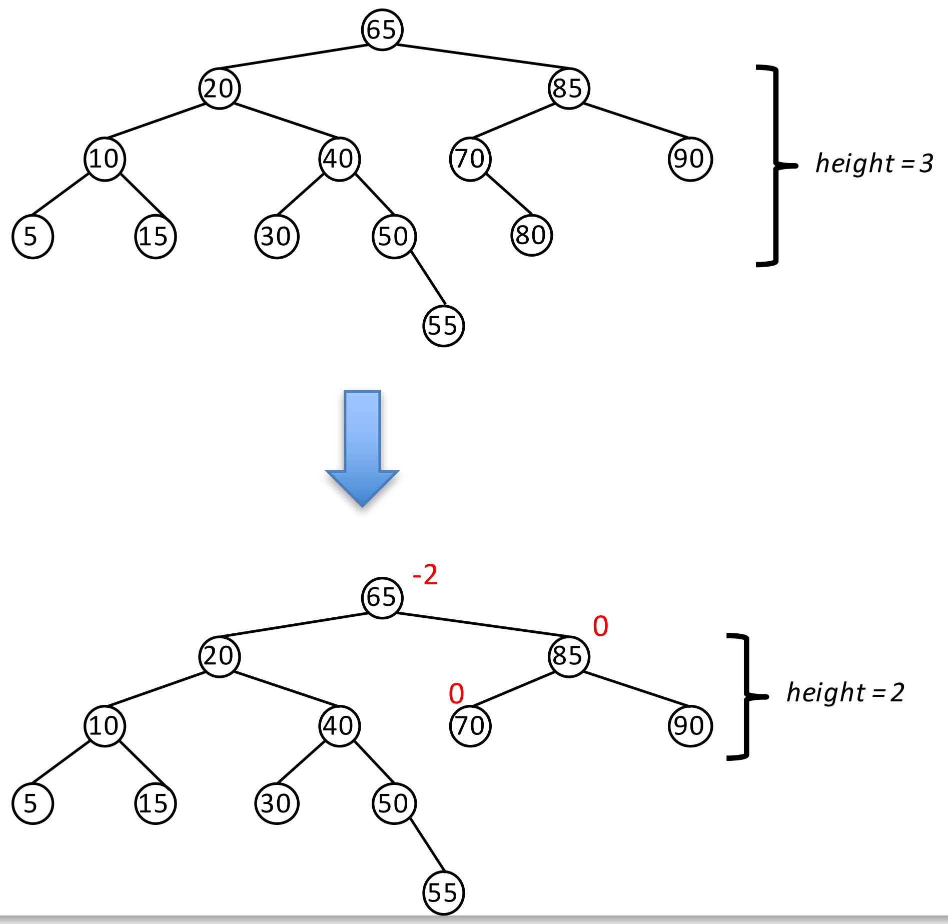
Removing values



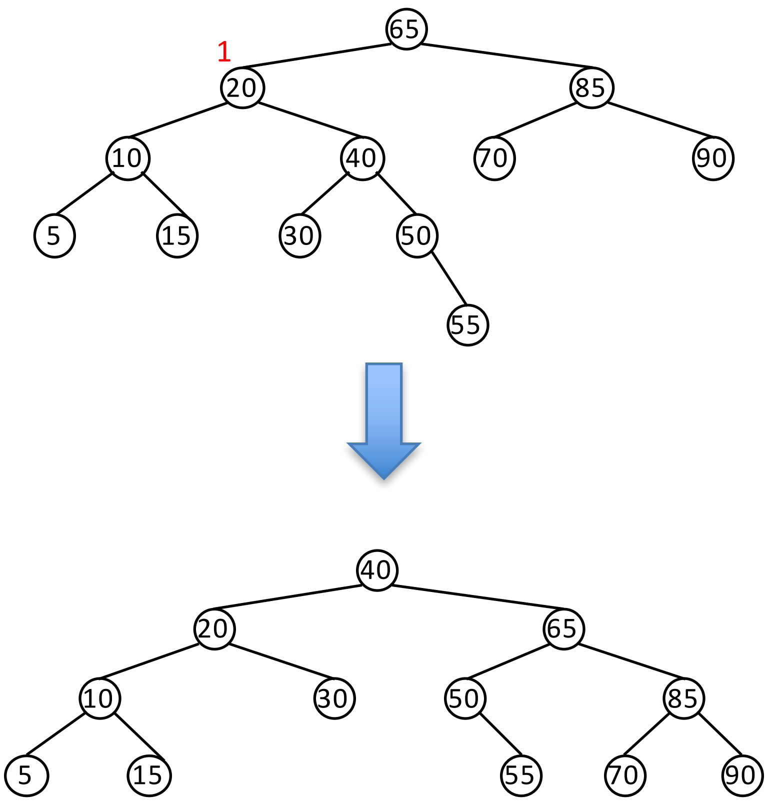


Removing values

Delete 80:

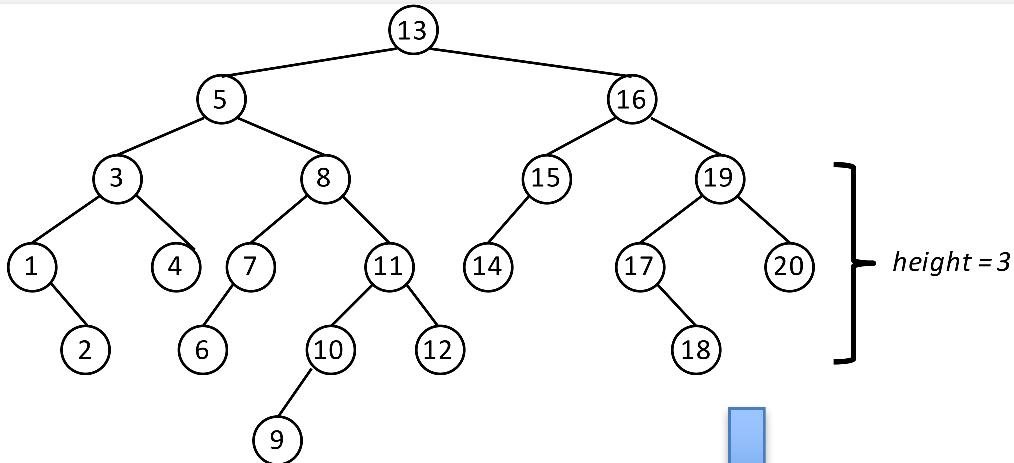


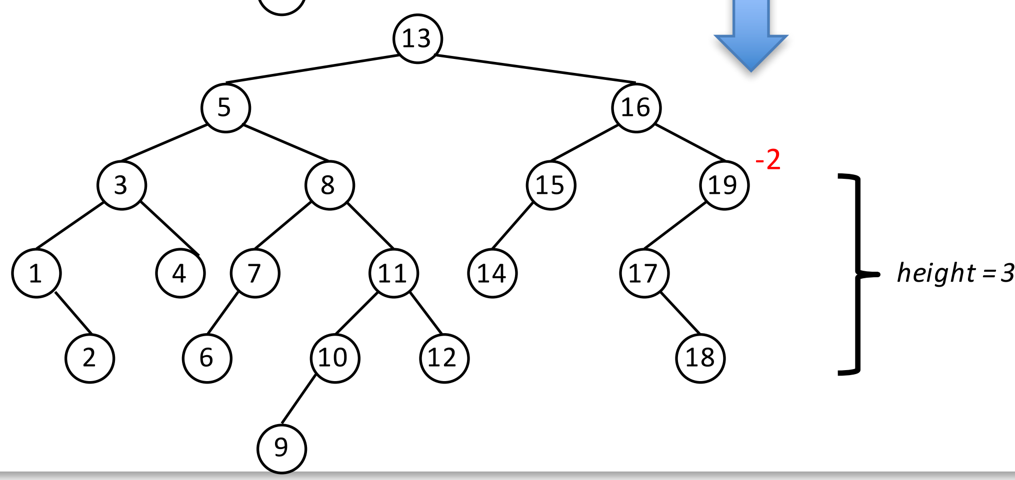
Removing values



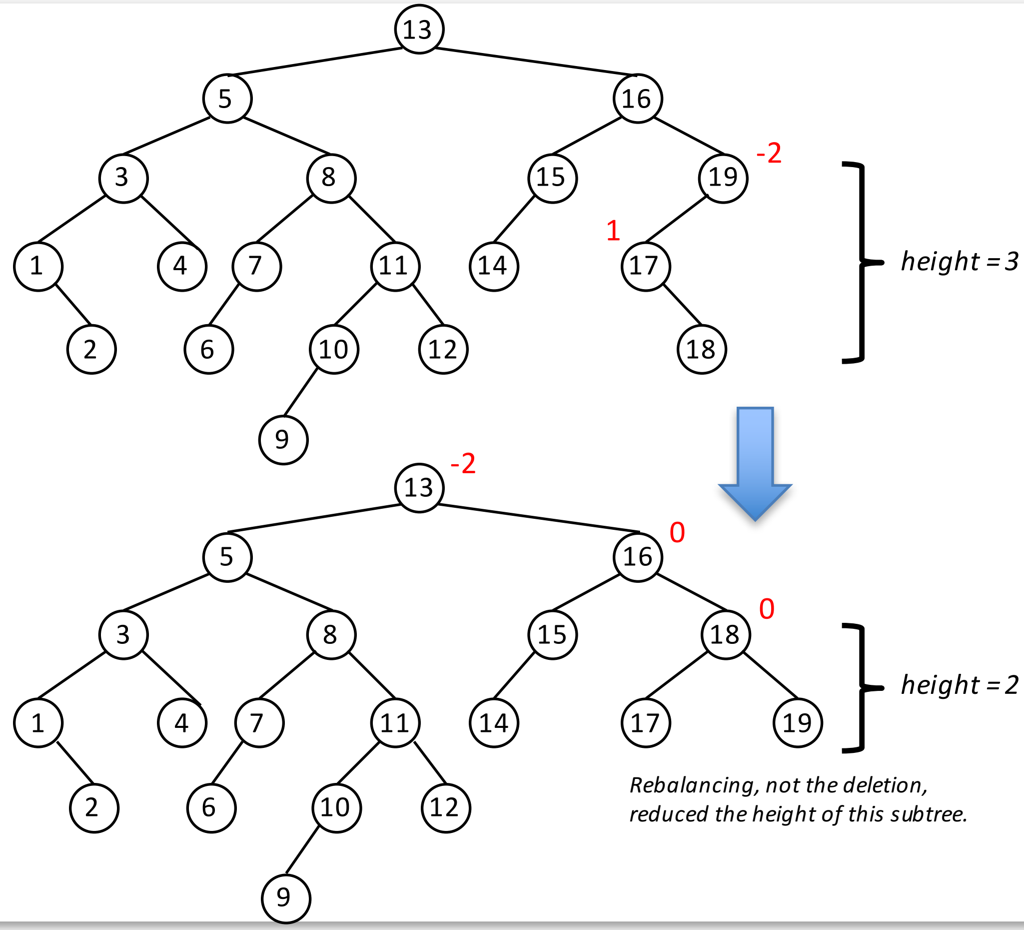
Removing values

Delete 20:

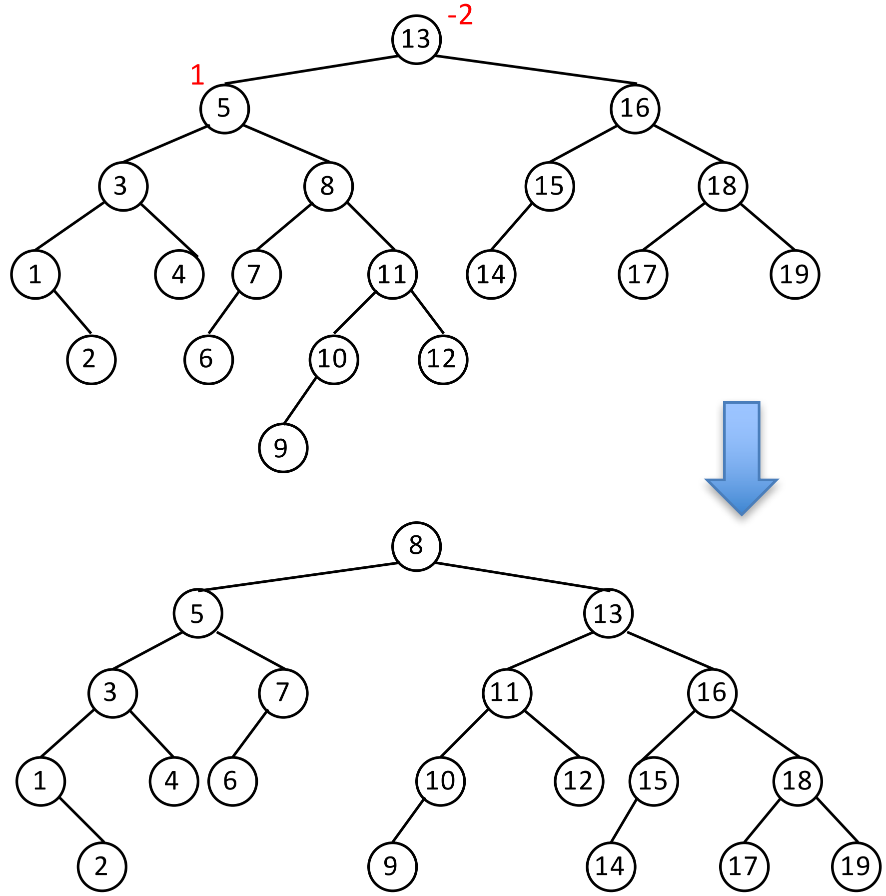




Removing values



Removing values



Summary

Summary

Balanced binary search trees are like a structural implementation of the binary search algorithm.

So, now we can use binary search on a structure built with linked nodes.

AVL trees offer guaranteed O(log N) performance on all three major collection operations: add, remove, and search.

	Self-Ordered Lists		
	Array	Linked List	AVL Tree
add(element)	O(N)	O(N)	O(log N)
remove(element)	O(N)	O(N)	O(log N)
search(element)	O(log N)	O(N)	O(log N)