

Binary Search Trees

Data structures

Fall 2018

Binary Search Trees



- Dictionary Operations:
 - `get(key)`
 - `put(key, value)`
 - `remove(key)`
- Additional operations:
 - `ascend()`
 - `get(index)` (indexed binary search tree)
 - `remove(index)` (indexed binary search tree)

Complexity Of Dictionary Operations

get(), put() and remove()

Data Structure	Worst Case	Expected
Hash Table	$O(n)$	$O(1)$
Binary Search Tree	$O(n)$	$O(\log n)$
Balanced Binary Search Tree	$O(\log n)$	$O(\log n)$

n is number of elements in dictionary

Complexity Of Other Operations

ascend(), get(index), remove(index)

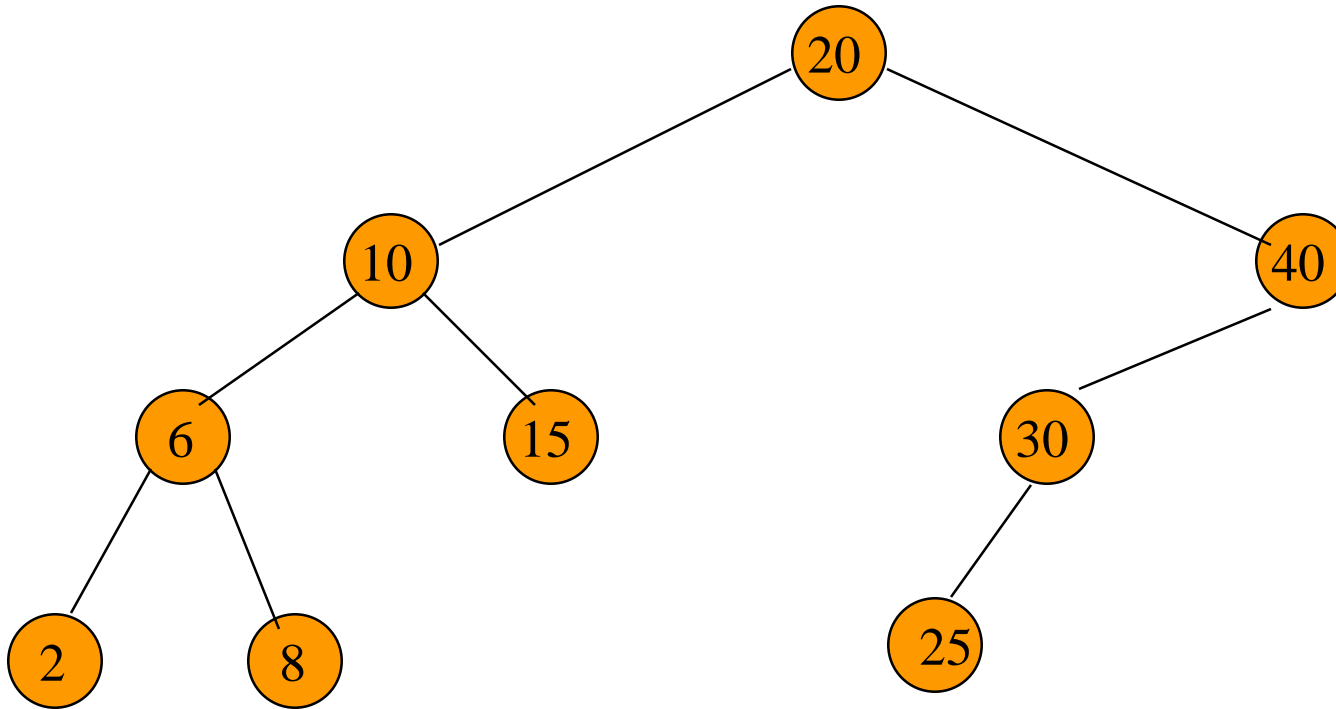
Data Structure	ascend	get and remove
Hash Table	$O(D + n \log n)$	$O(D + n)$
Indexed BST	$O(n)$	$O(n)$
Indexed Balanced BST	$O(n)$	$O(\log n)$

D is number of buckets

Definition Of Binary Search Tree

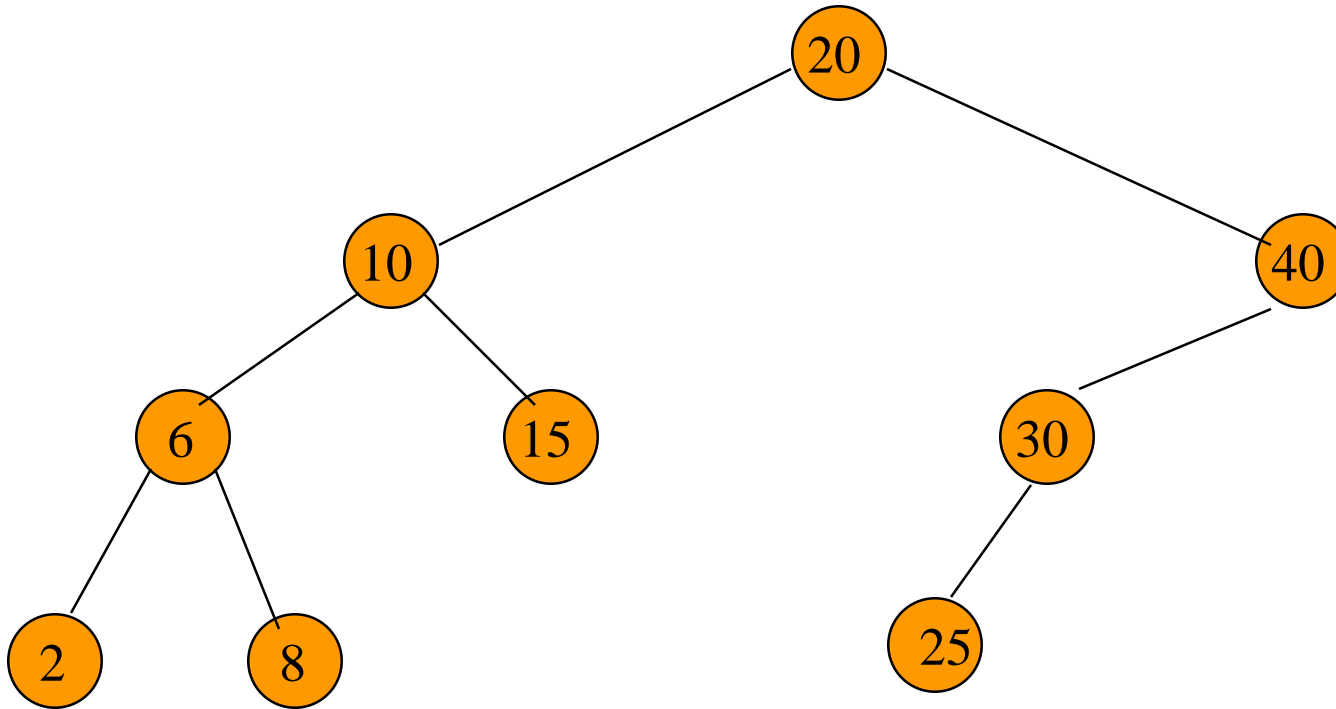
- A binary tree.
- Each node has a (key, value) pair.
- For every node x , all keys in the left subtree of x are smaller than that in x .
- For every node x , all keys in the right subtree of x are greater than that in x .

Example Binary Search Tree



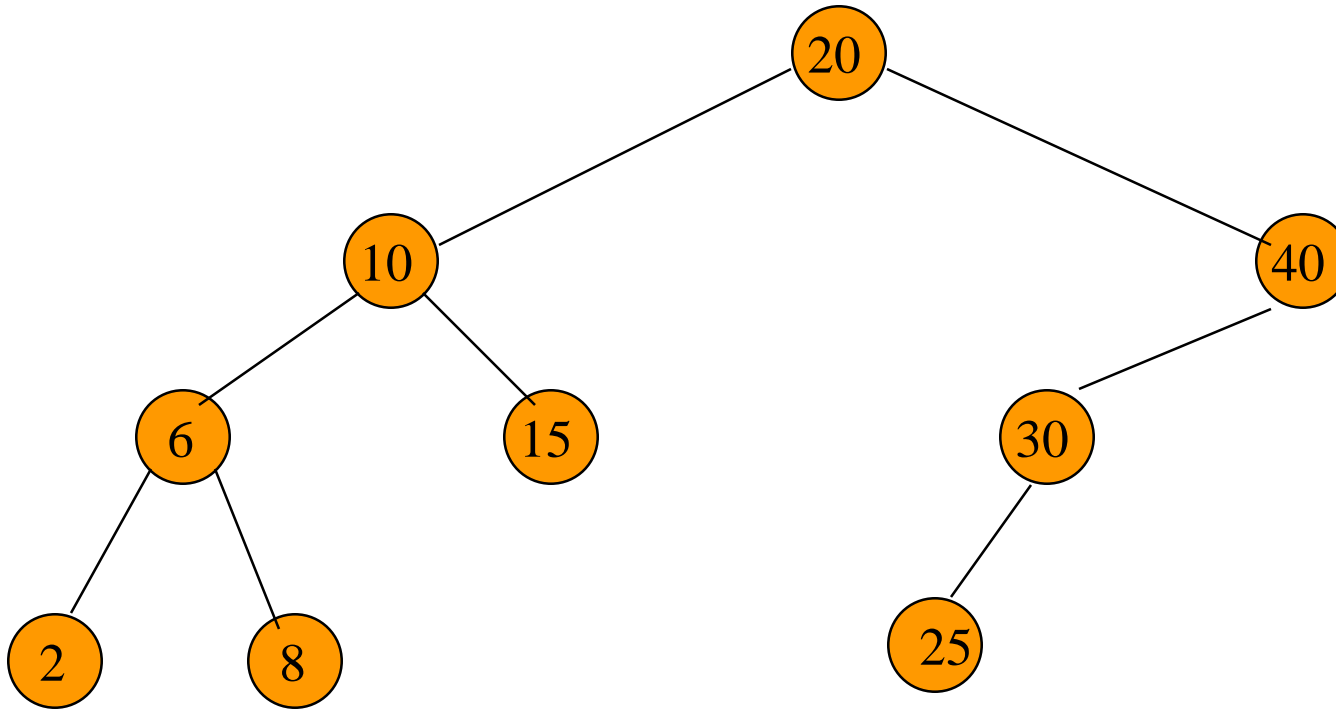
Only keys are shown.

The Operation ascend()



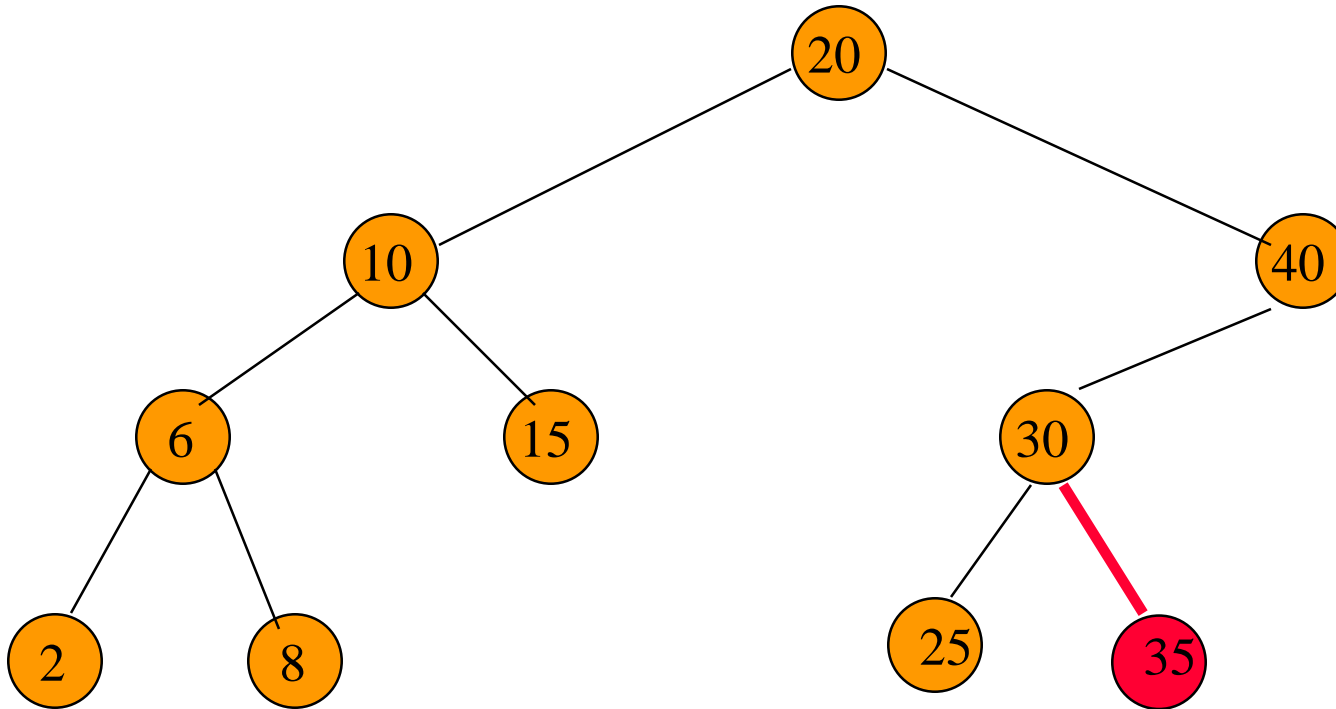
Do an inorder traversal. $O(n)$ time.

The Operation get()



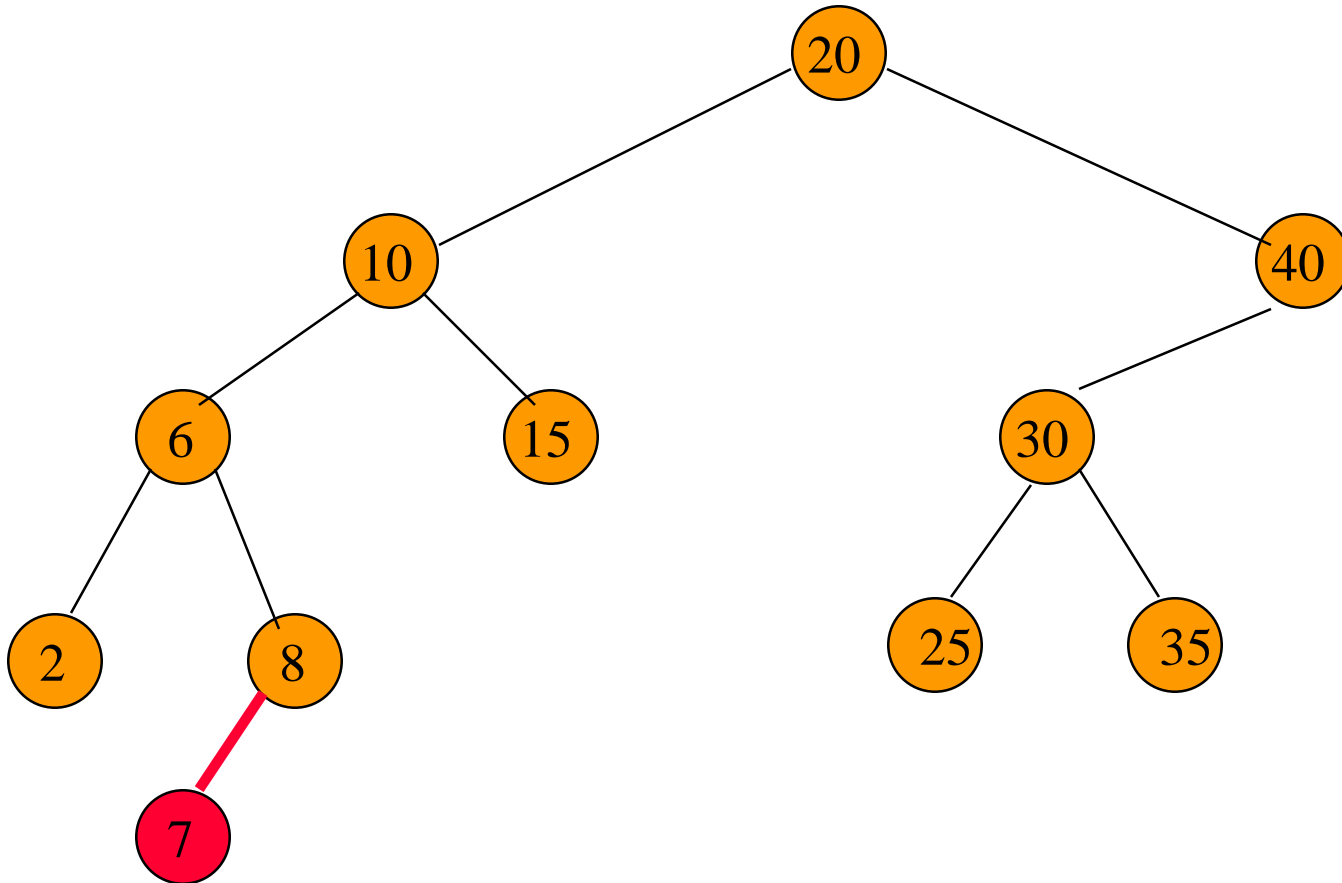
Complexity is $O(\text{height}) = O(n)$, where n is number of nodes/elements.

The Operation put()



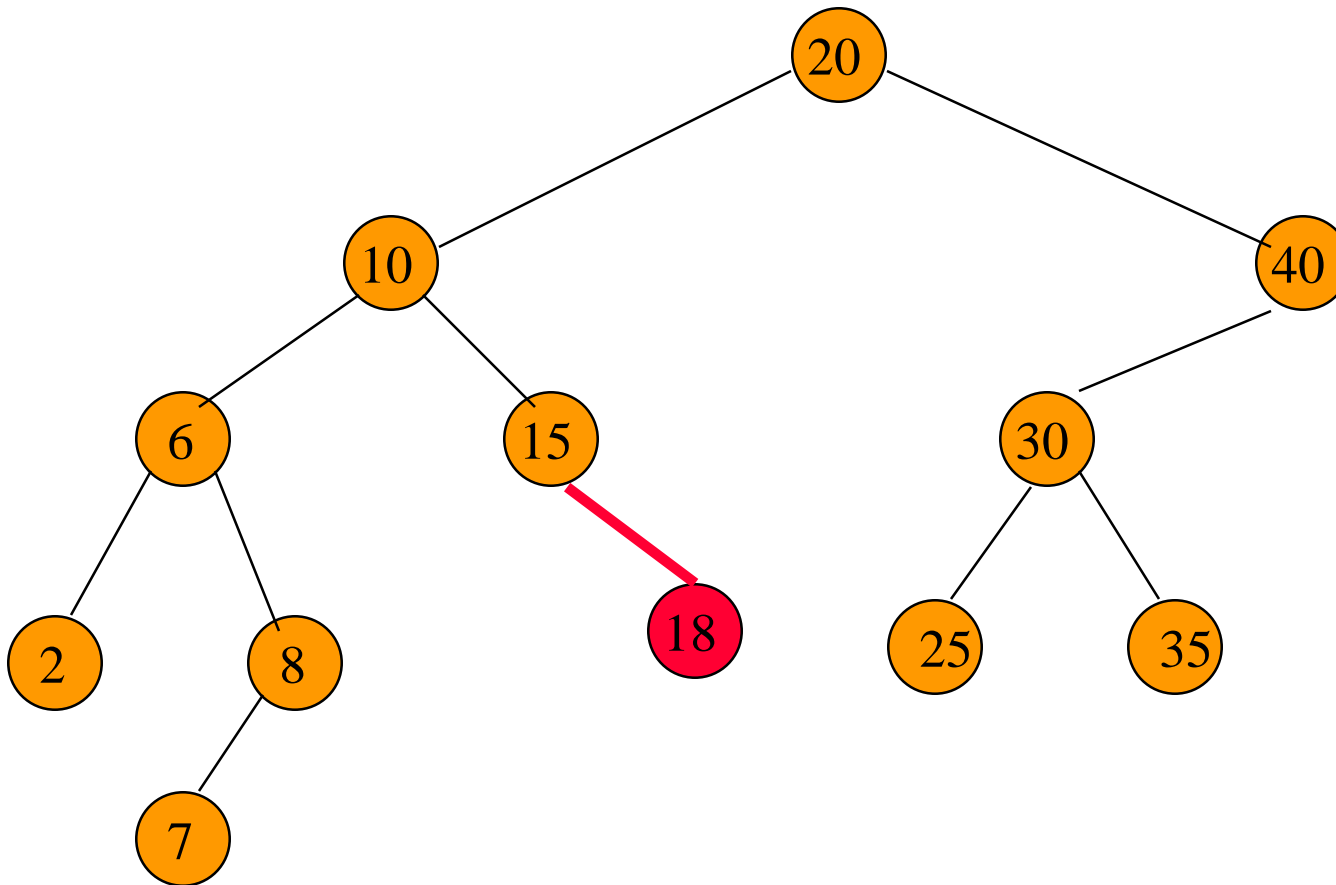
Put a pair whose key is **35**.

The Operation put()



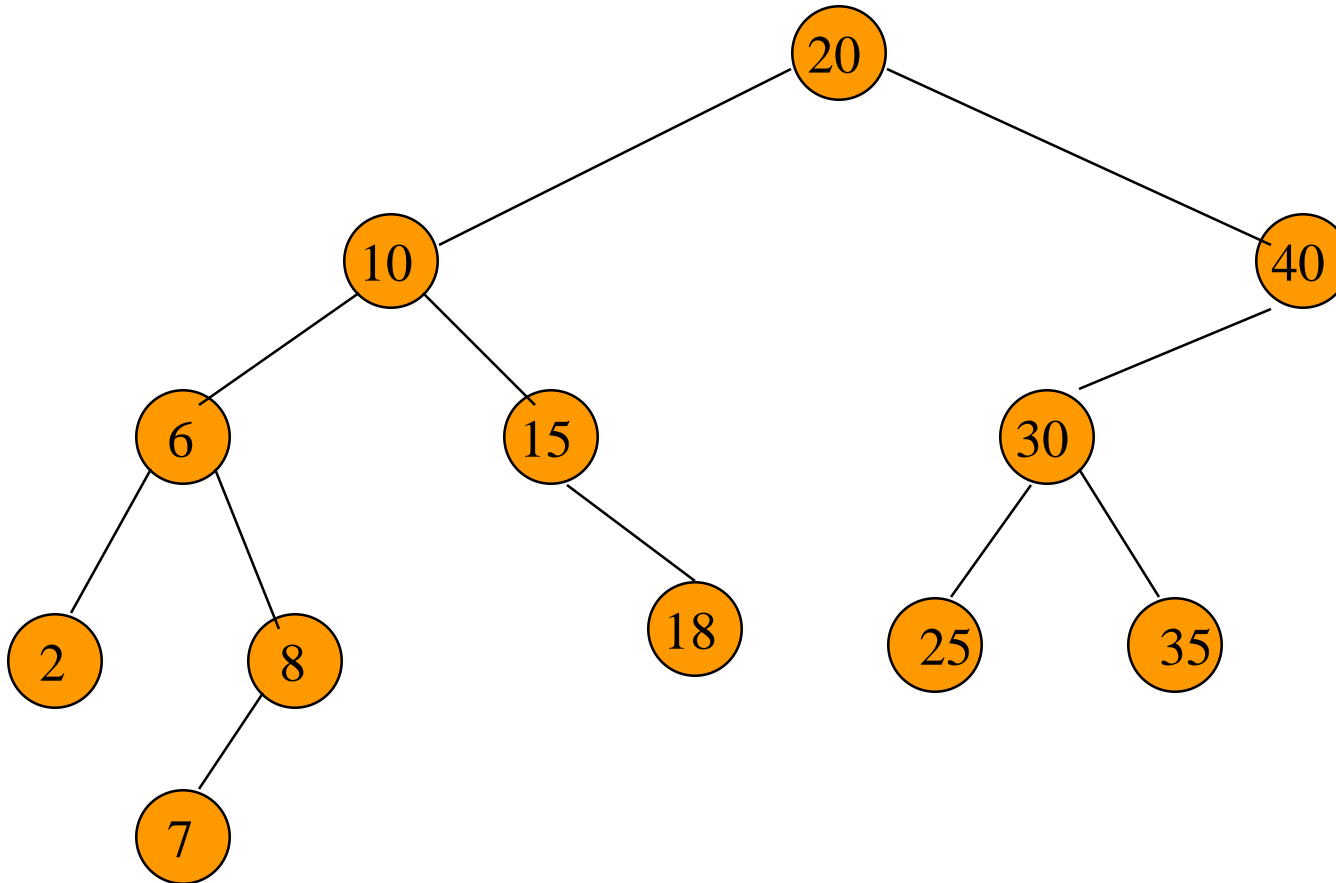
Put a pair whose key is **7**.

The Operation put()



Put a pair whose key is **18**.

The Operation put()



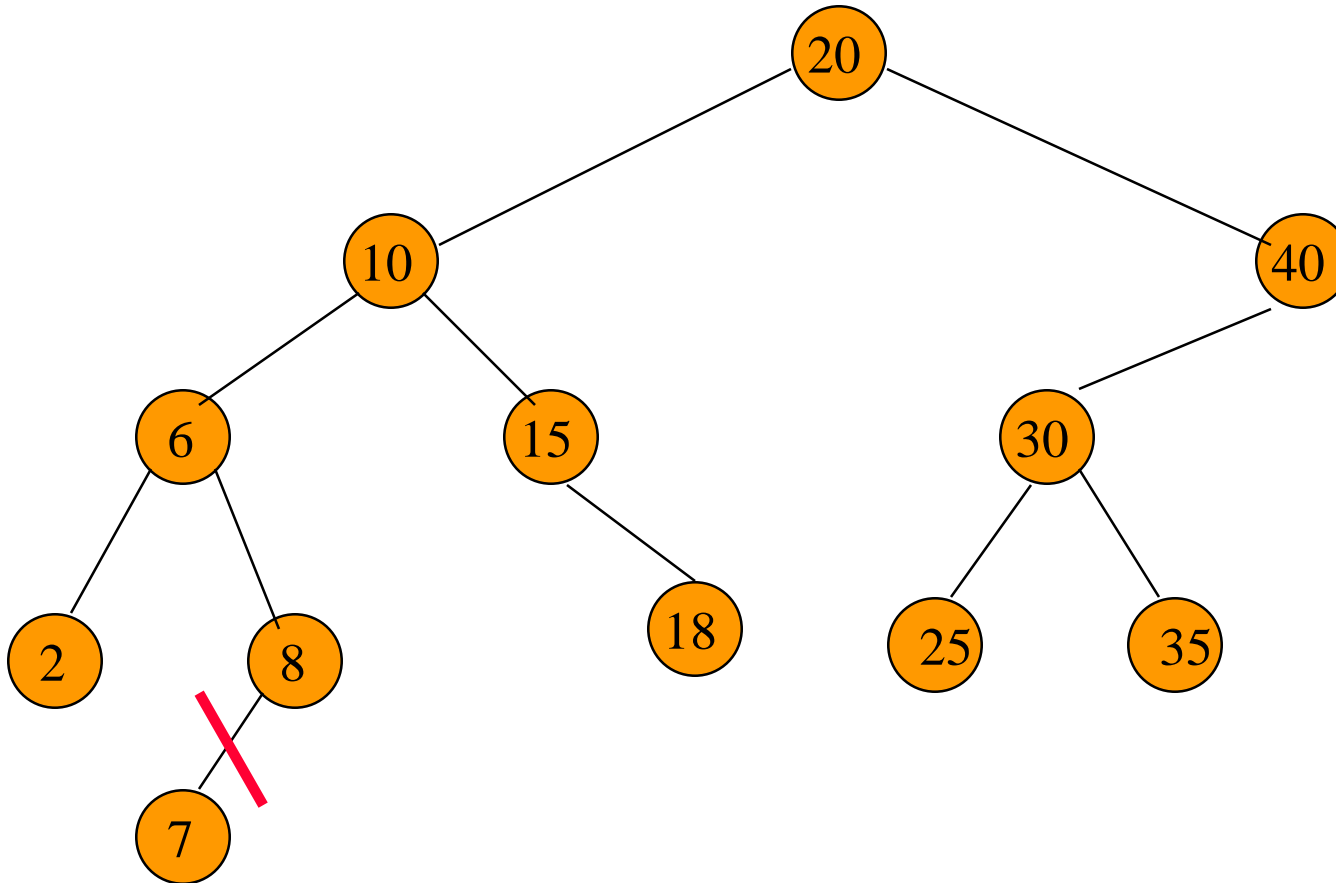
Complexity of `put()` is $O(\text{height})$.

The Operation remove()

Three cases:

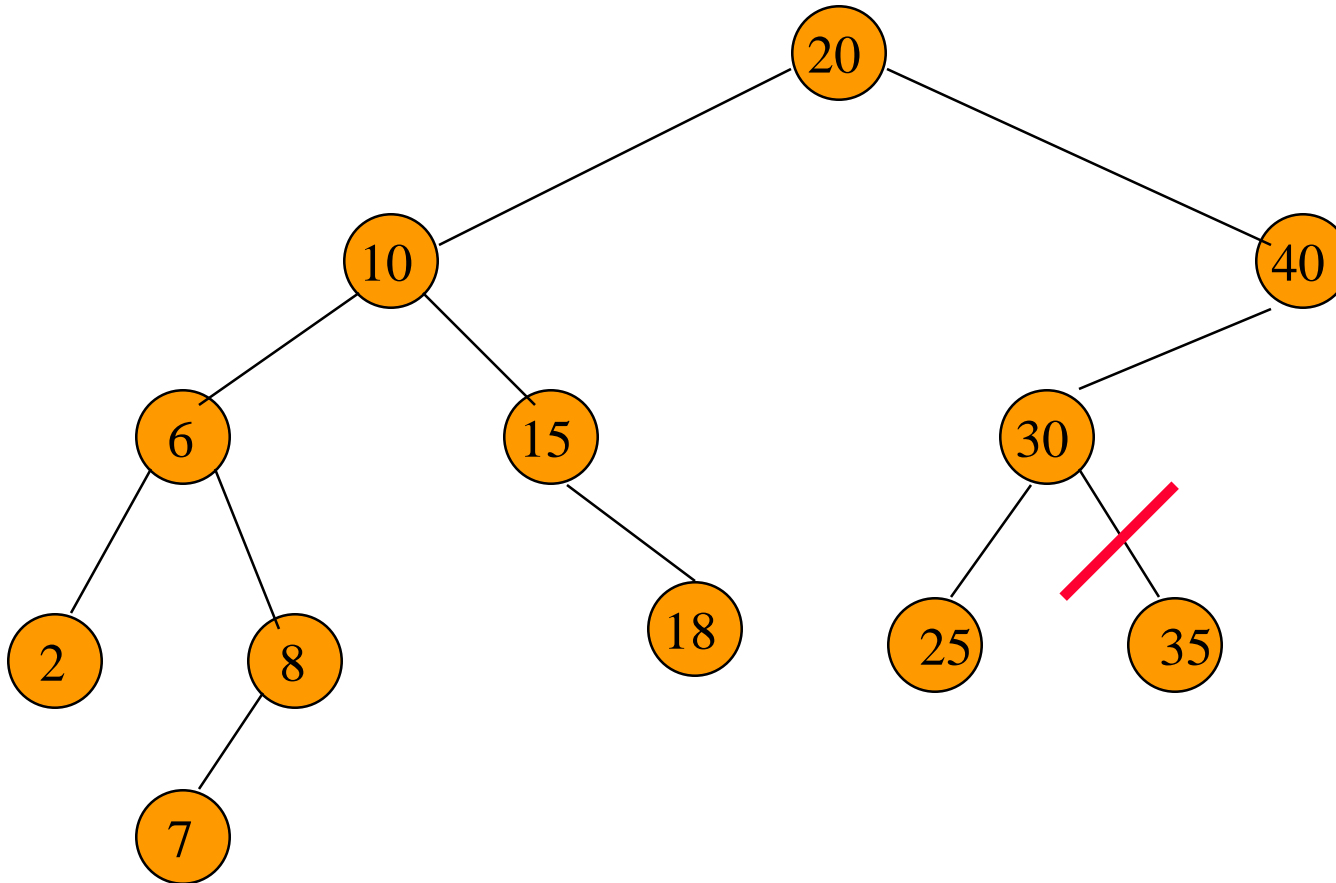
- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

Remove From A Leaf



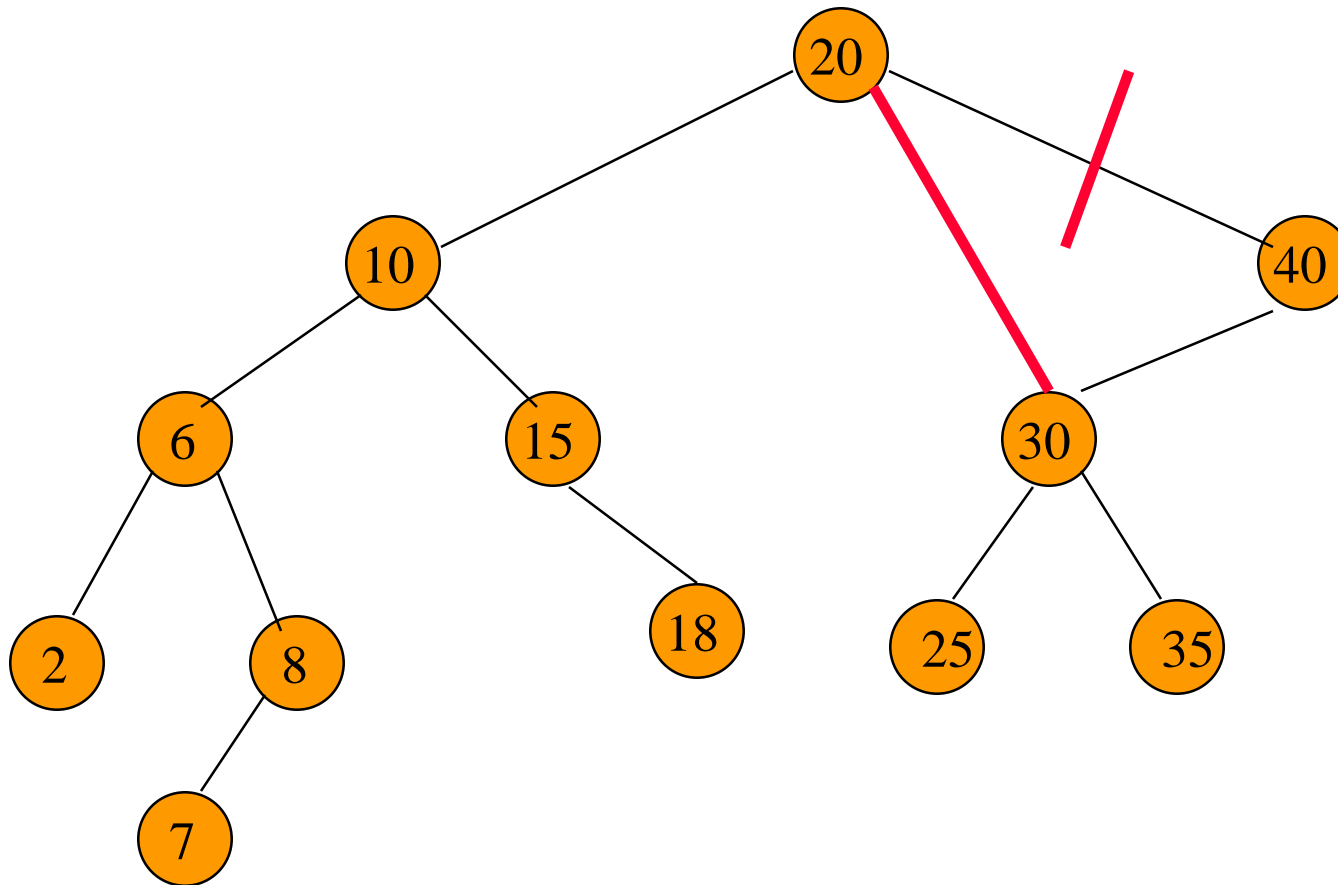
Remove a leaf element. key = 7

Remove From A Leaf (contd.)



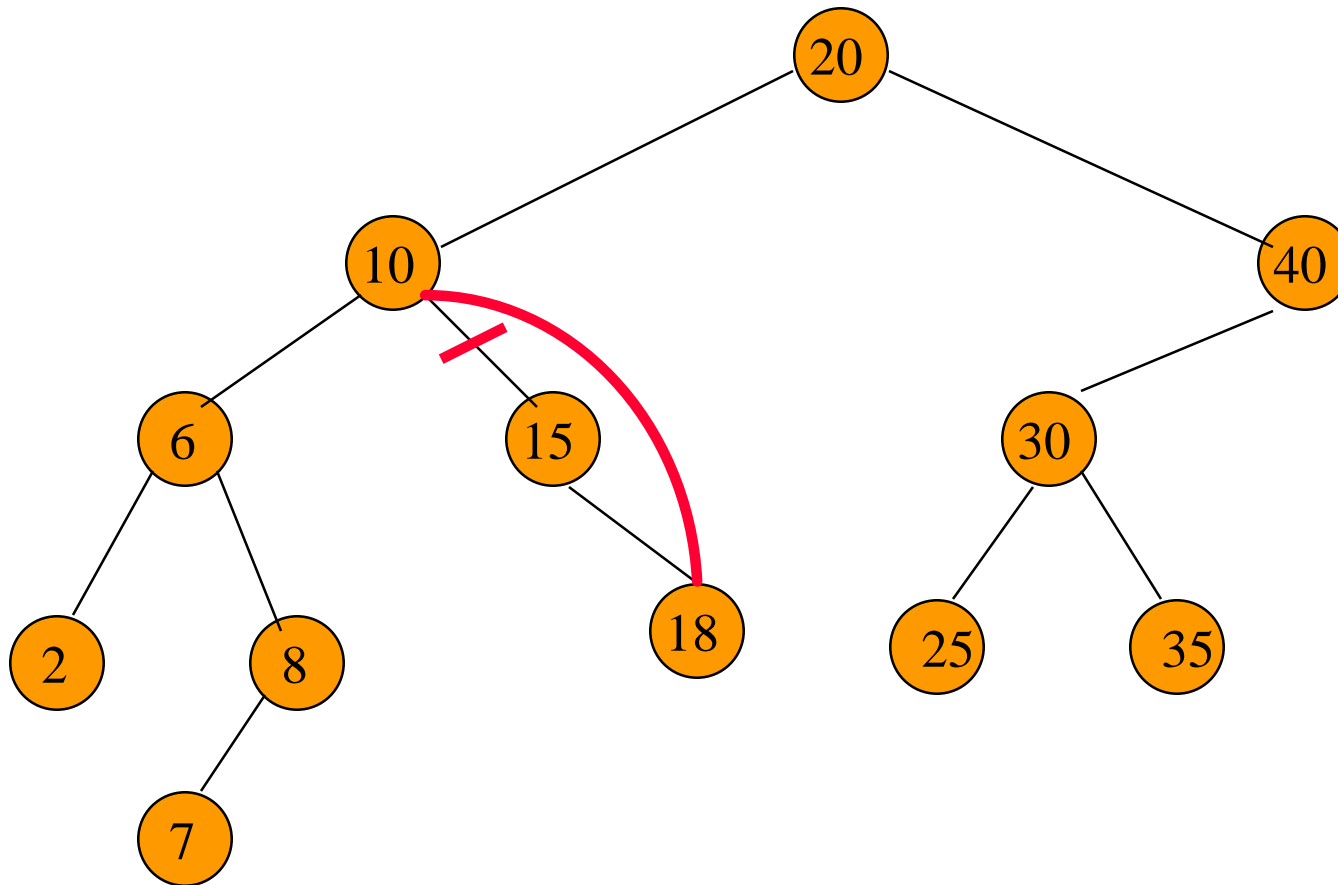
Remove a leaf element. key = 35

Remove From A Degree 1 Node



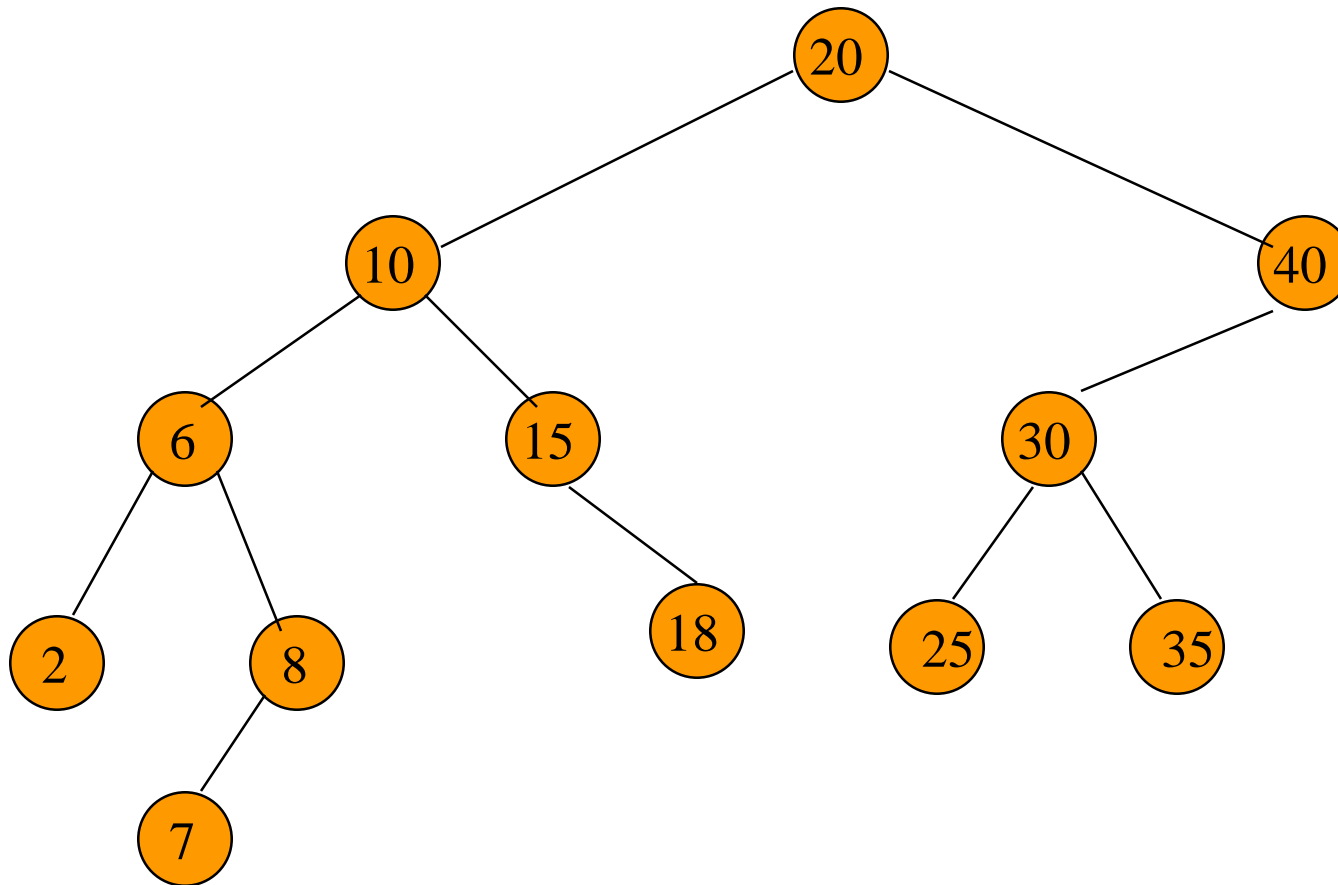
Remove from a degree 1 node. key = 40

Remove From A Degree 1 Node (contd.)



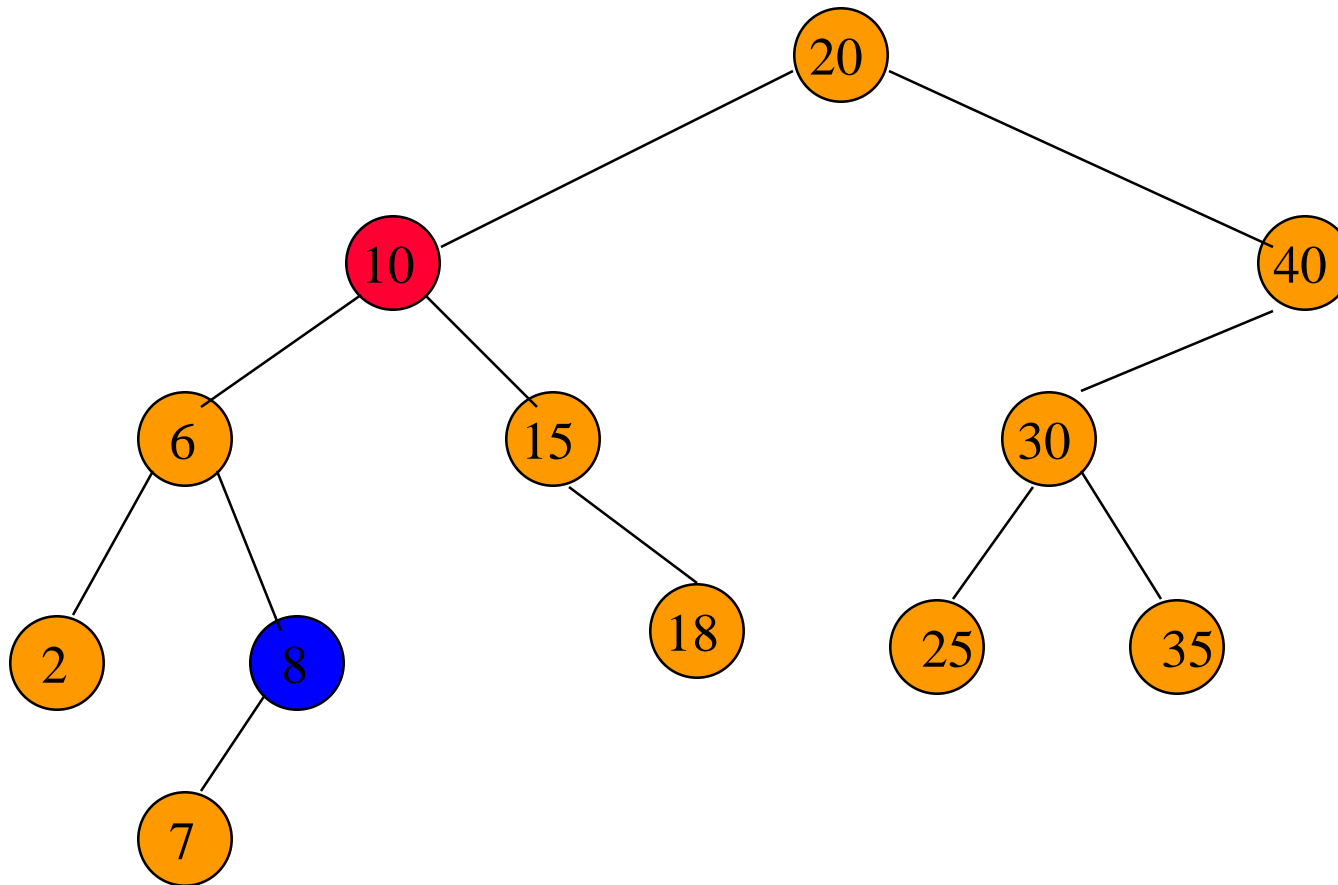
Remove from a degree 1 node. key = 15

Remove From A Degree 2 Node



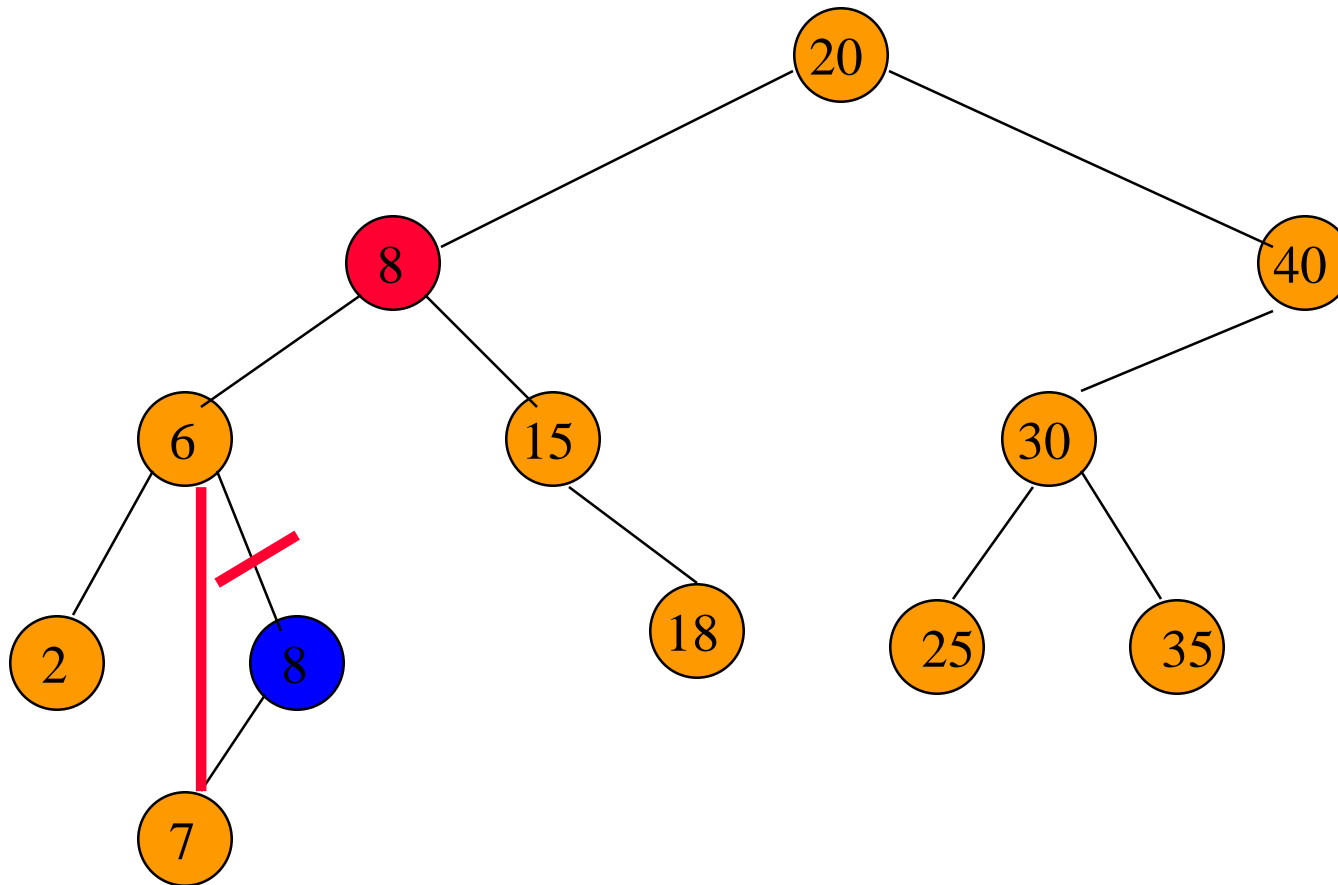
Remove from a degree 2 node. key = 10

Remove From A Degree 2 Node



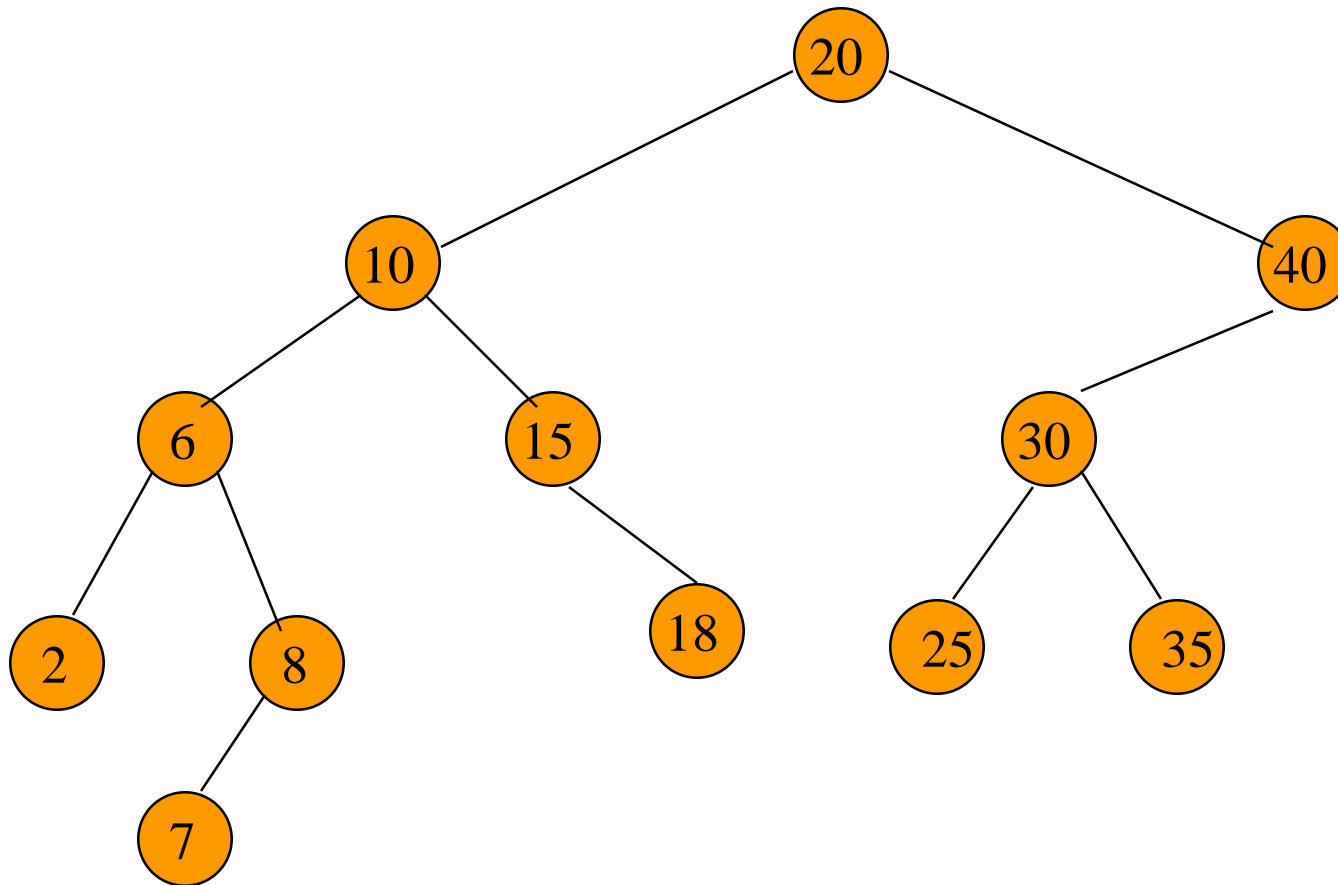
Replace with largest key in left subtree (or smallest in right subtree).

Remove From A Degree 2 Node



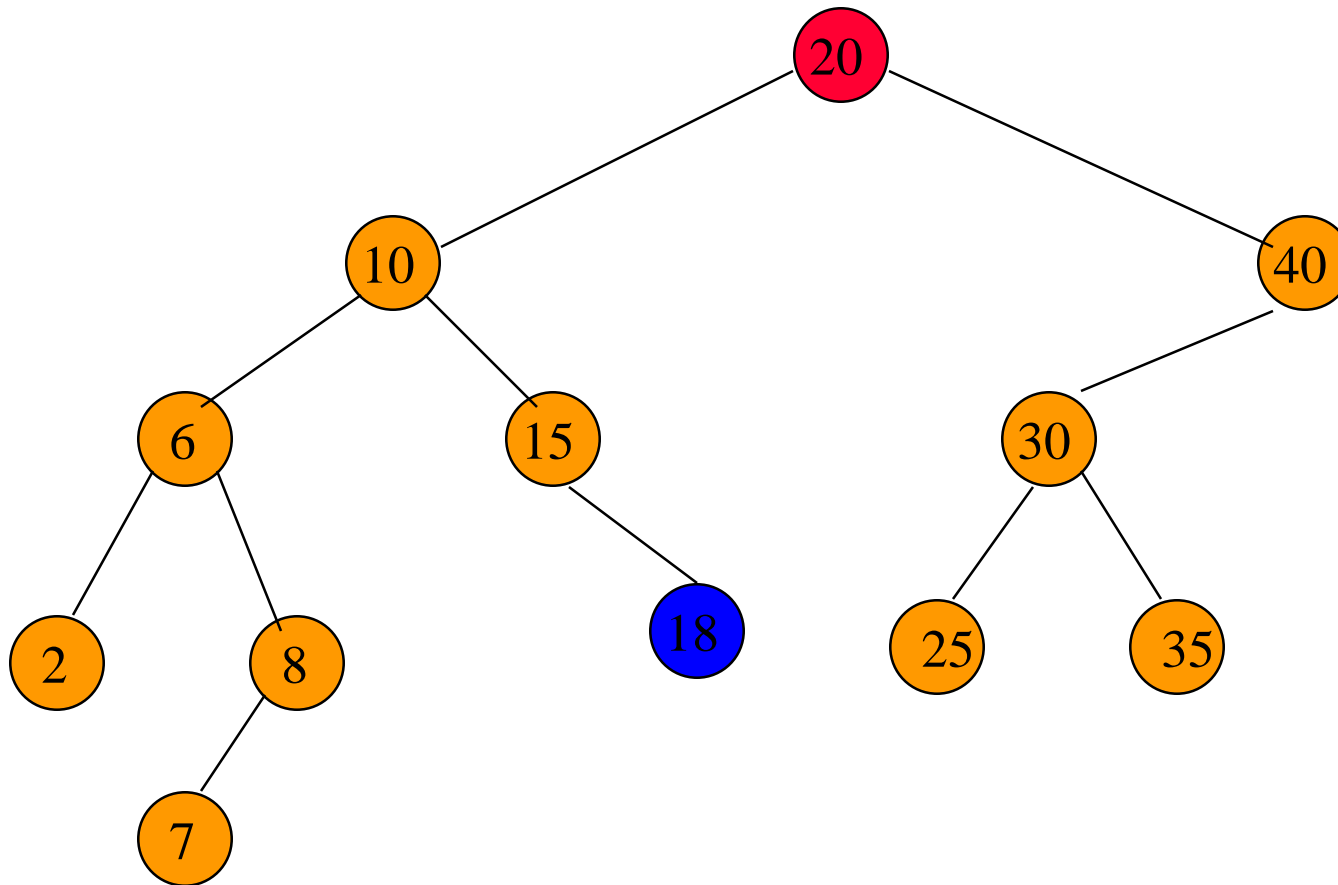
Largest key must be in a leaf or degree 1 node.

Another Remove From A Degree 2 Node



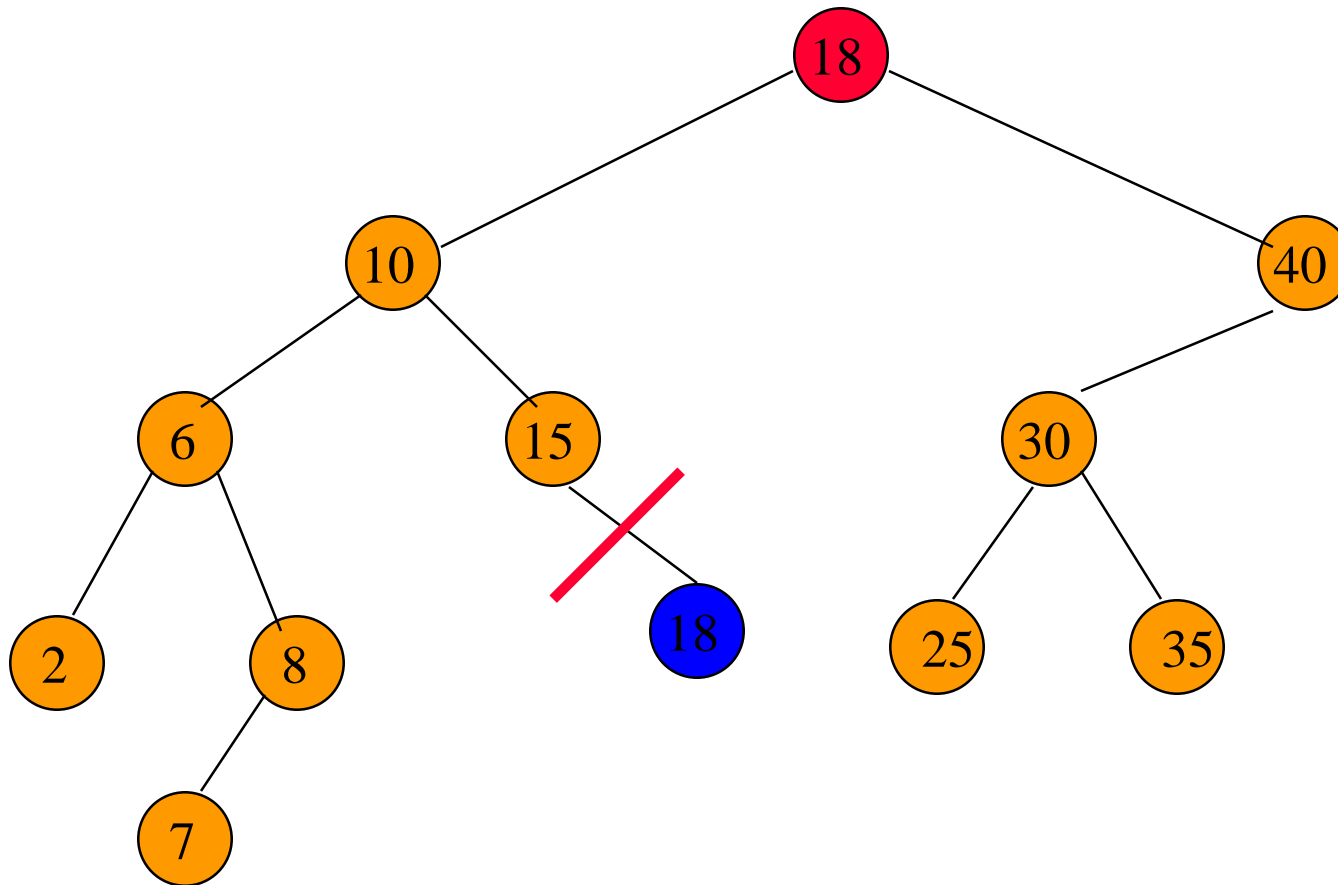
Remove from a degree 2 node. key = 20

Remove From A Degree 2 Node



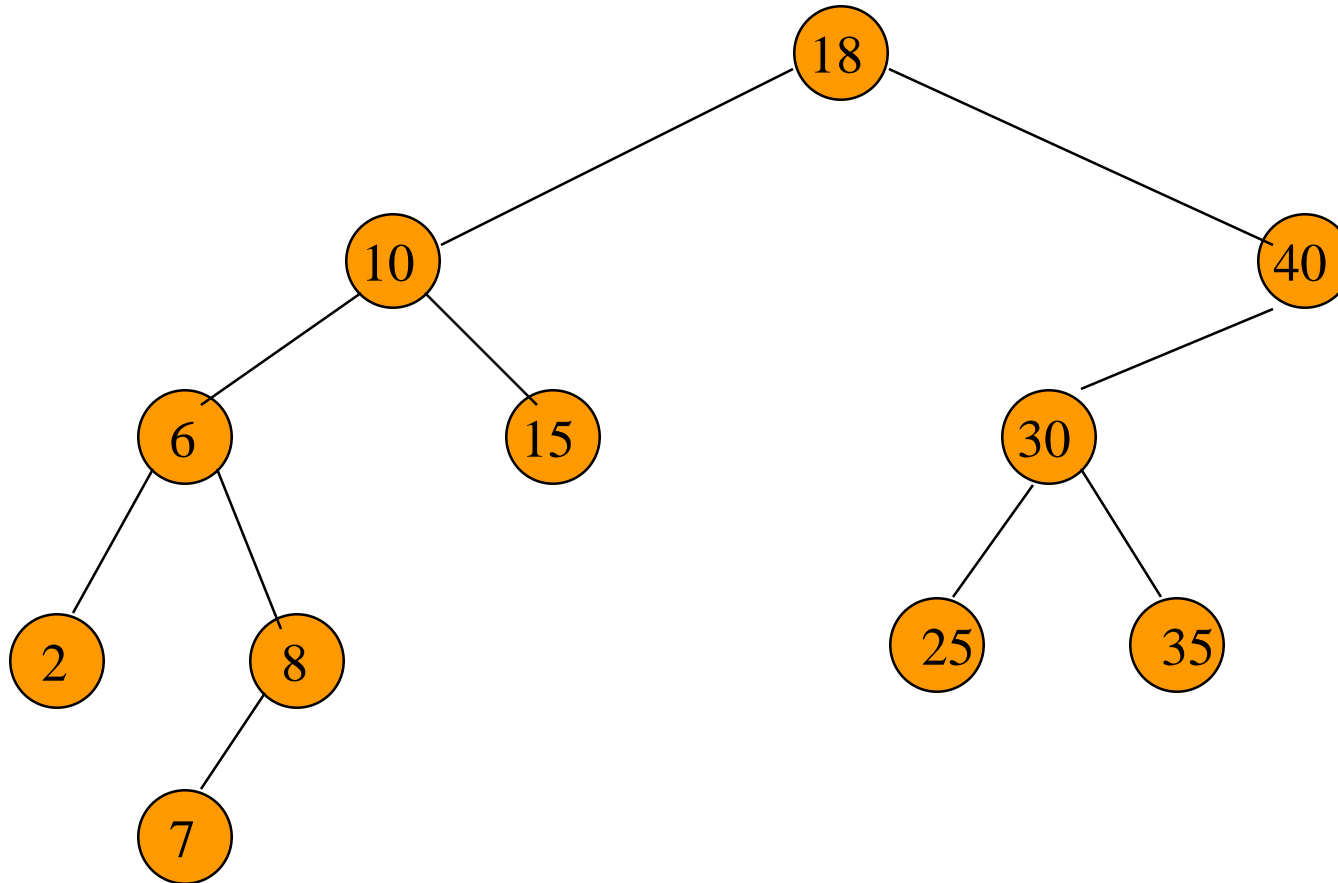
Replace with largest in left subtree.

Remove From A Degree 2 Node



Replace with largest in left subtree.

Remove From A Degree 2 Node

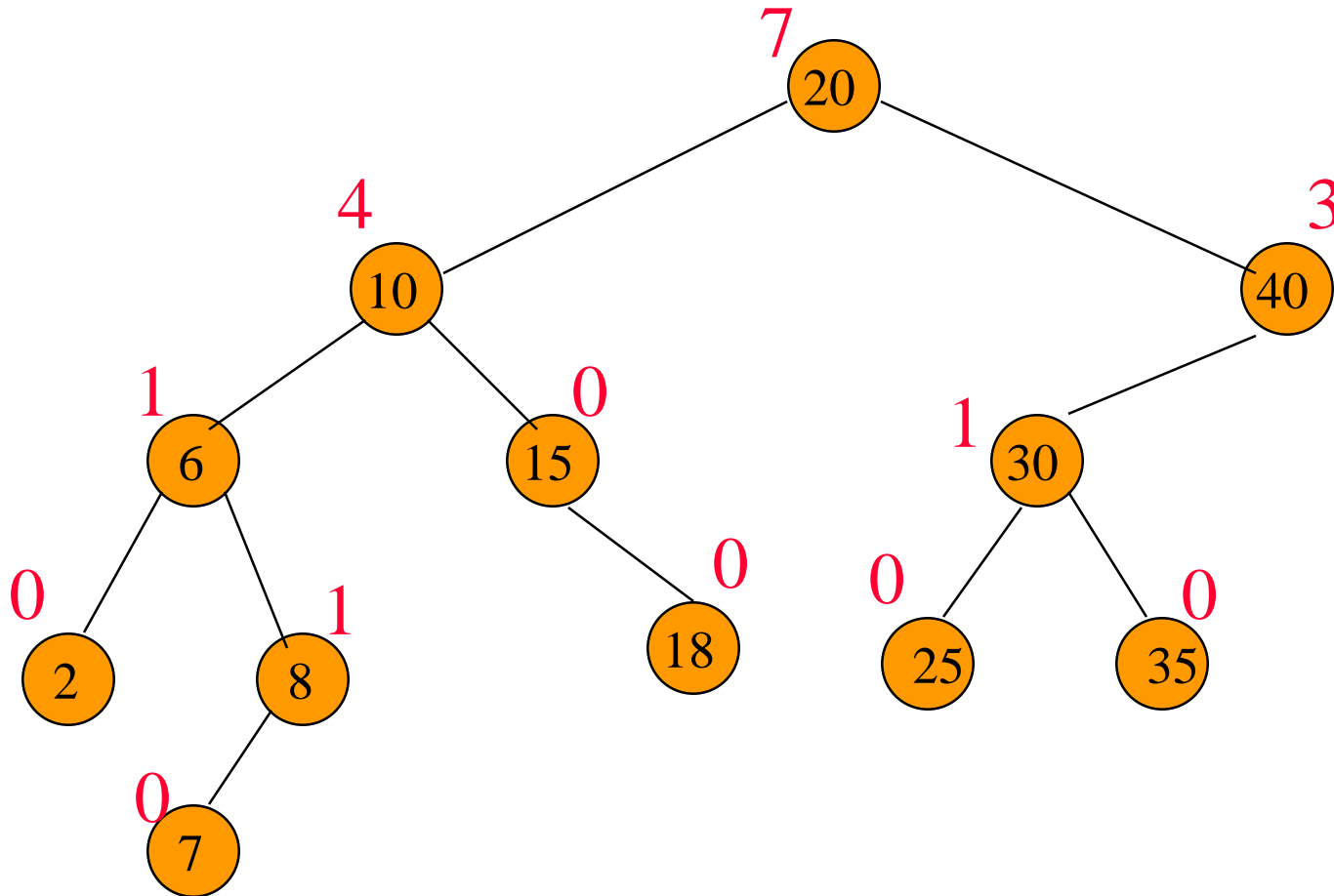


Complexity is $O(\text{height})$.

Indexed Binary Search Tree

- Binary search tree.
- Each node has an additional field.
 - **leftSize** = number of nodes in its left subtree

Example Indexed Binary Search Tree



leftSize values are in red

leftSize And Rank

Rank of an element is its position in inorder (inorder = ascending key order).

[2,6,7,8,10,15,18,20,25,30,35,40]

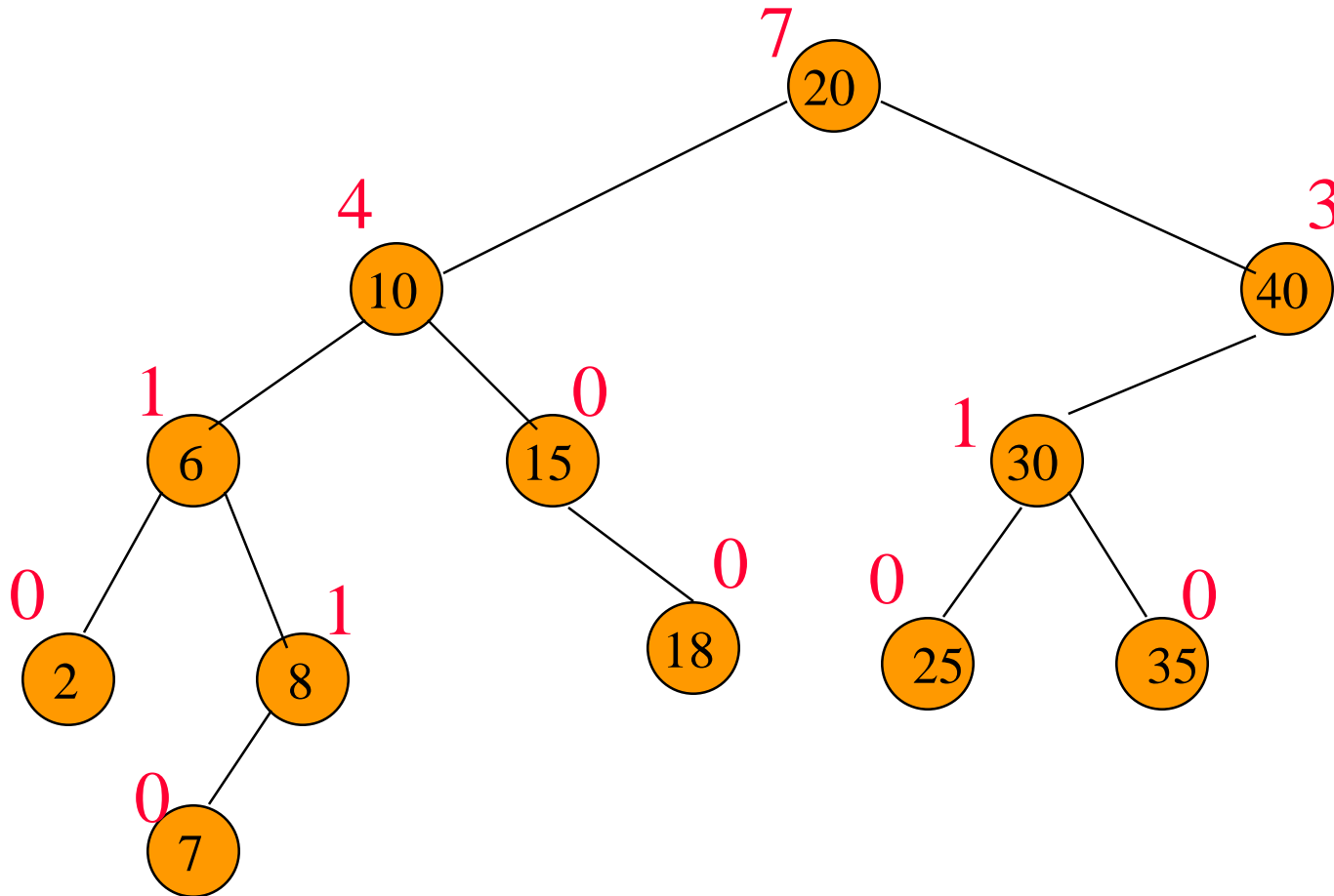
$\text{rank}(2) = 0$

$\text{rank}(15) = 5$

$\text{rank}(20) = 7$

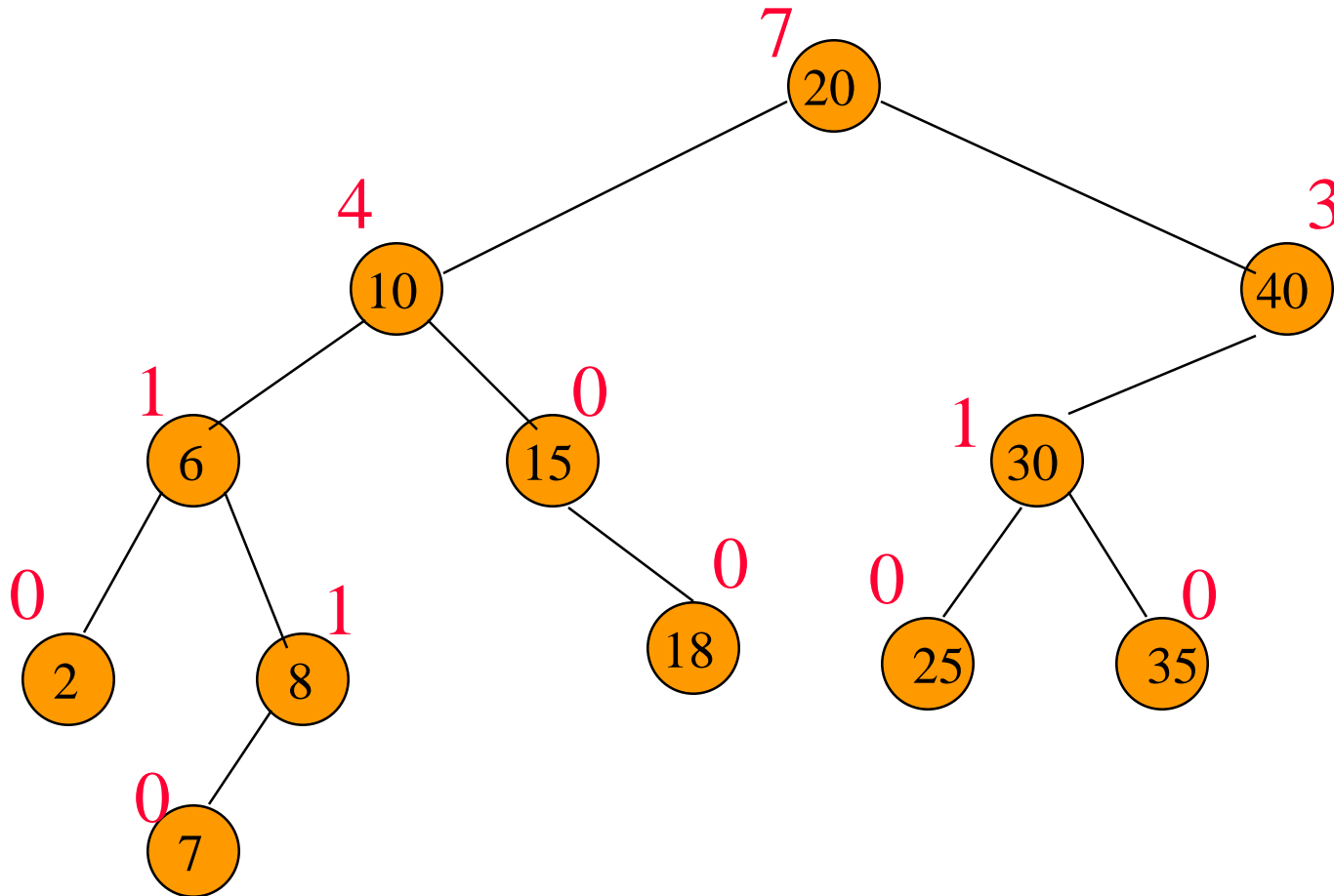
$\text{leftSize}(x) = \text{rank}(x)$ with respect to elements in subtree rooted at x

leftSize And Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

get(index) And remove(index)



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

get(index) And remove(index)

- if $\text{index} = \text{x.leftSize}$ desired element is x.element
- if $\text{index} < \text{x.leftSize}$ desired element is index 'th element in left subtree of x
- if $\text{index} > \text{x.leftSize}$ desired element is $(\text{index} - \text{x.leftSize} - 1)$ 'th element in right subtree of x

Applications

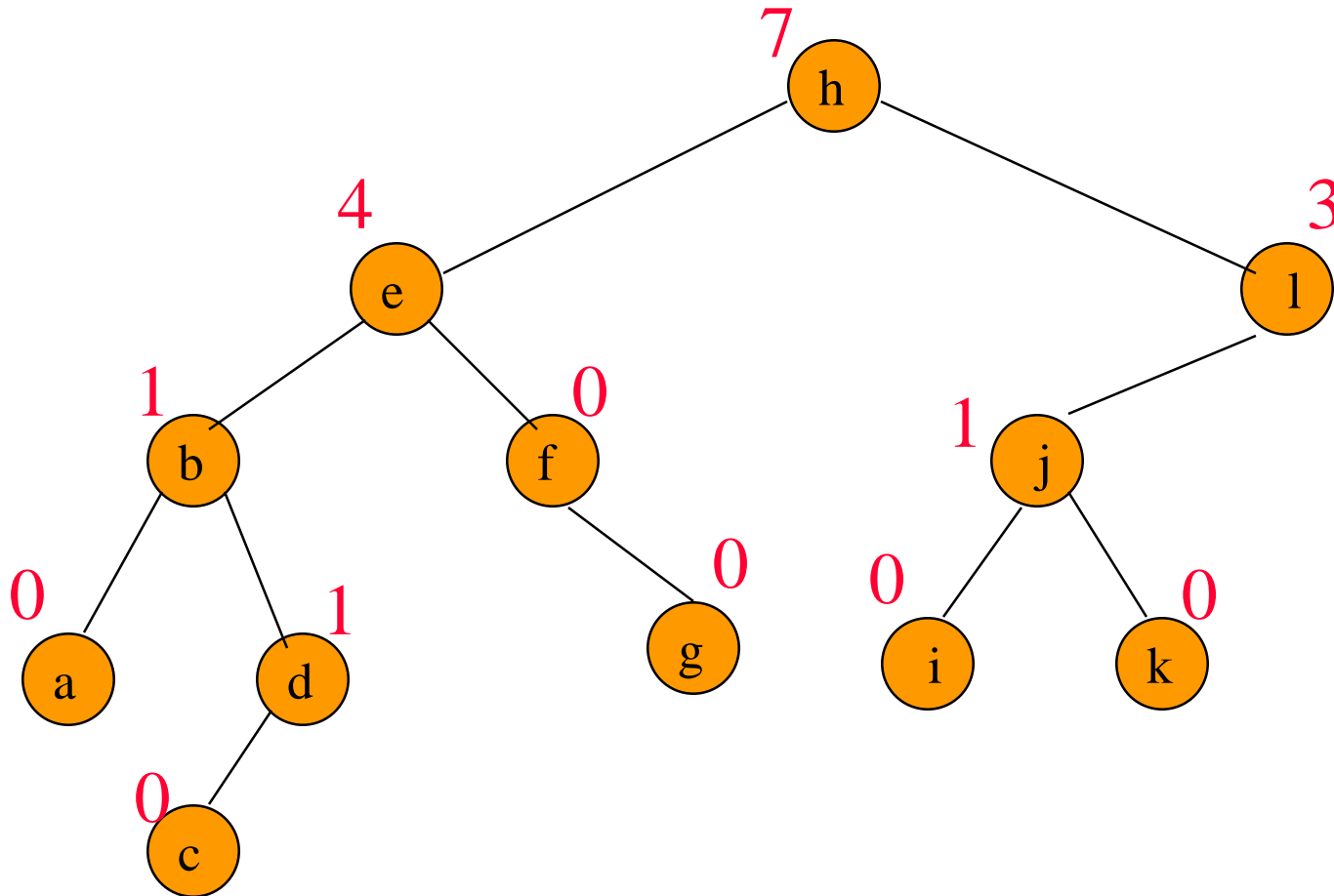
(Complexities Are For Balanced Trees)

//Best-fit bin packing in $O(n \log n)$ time.

Representing a linear list so that **get(index)**,
add(index, element), and **remove(index)**
run in $O(\log(\text{list size}))$ time (uses an
indexed binary tree, not indexed binary
search tree).

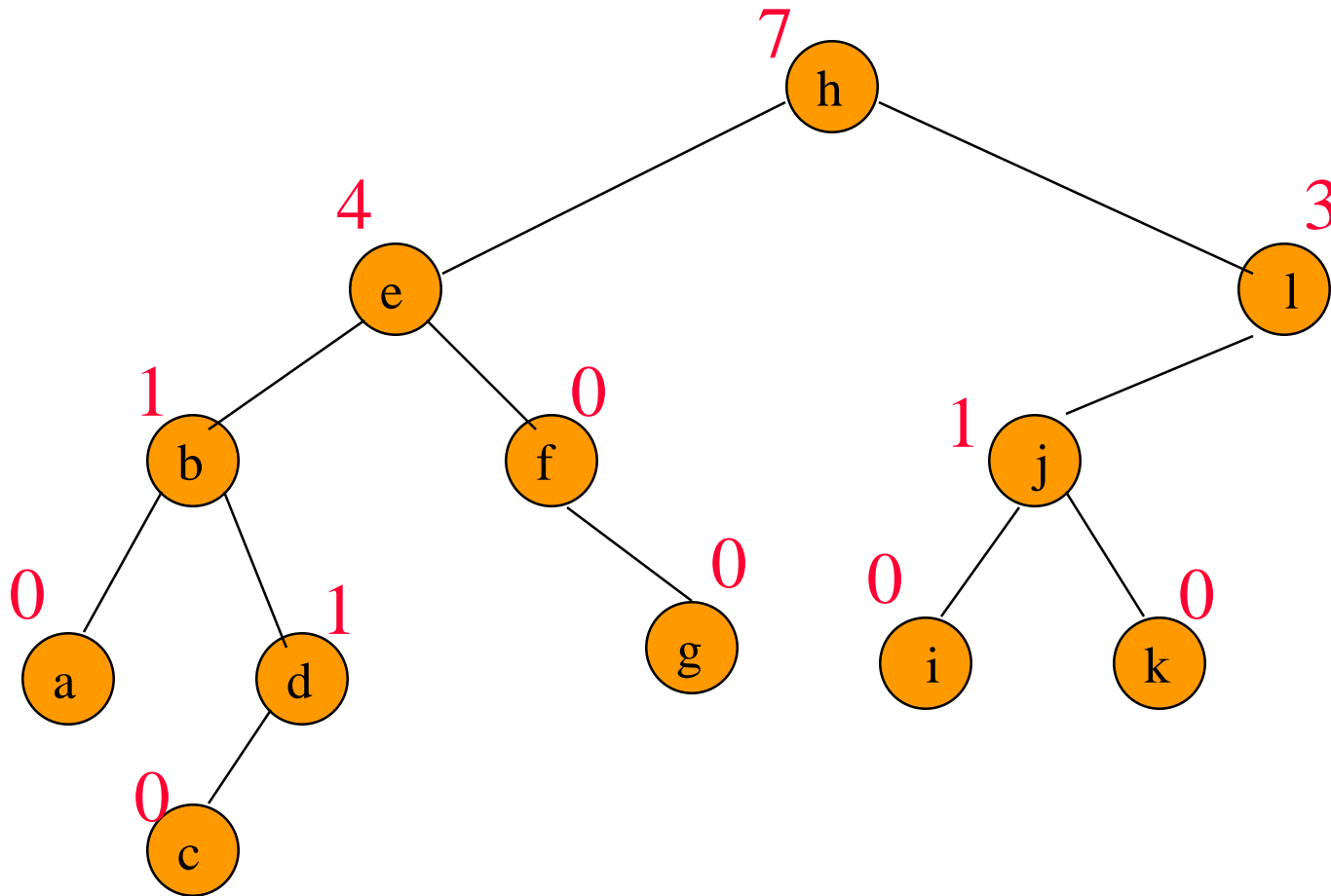
Can't use hash tables for either of these
applications.

Linear List As Indexed Binary Tree



list = [a,b,c,d,e,f,g,h,i,j,k,l]

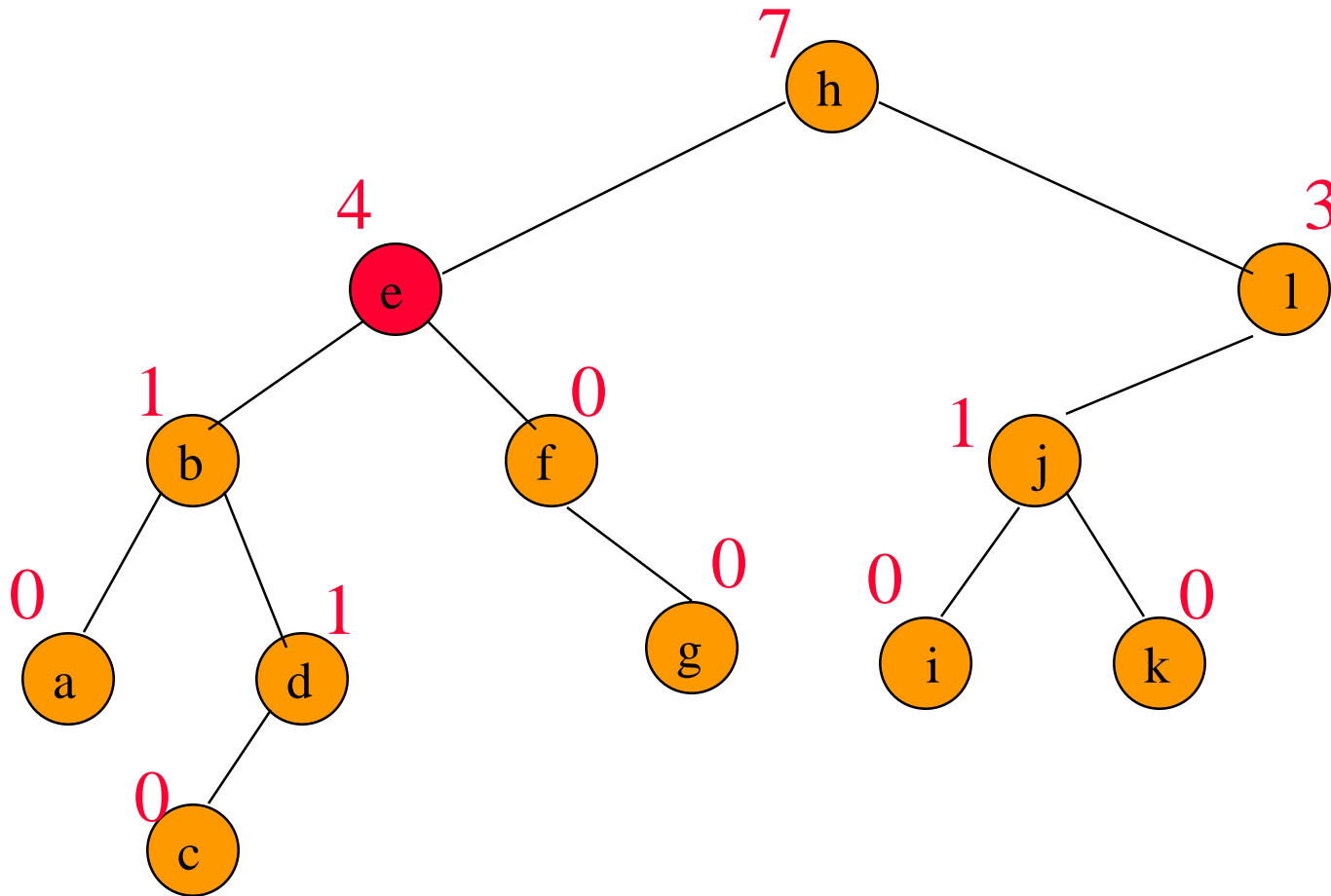
add(5, 'm')



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

find node with element 4 (e)

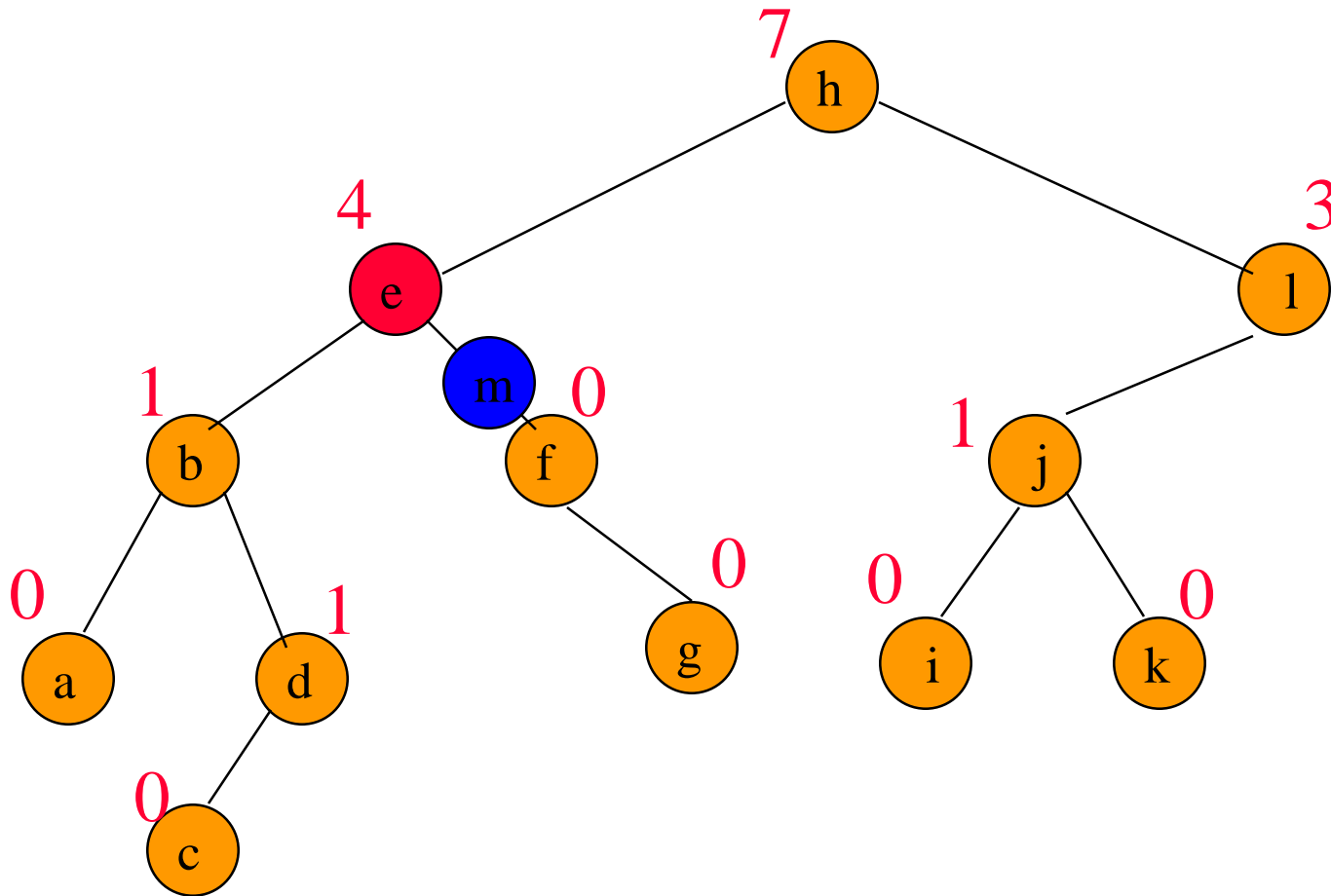
add(5, 'm')



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

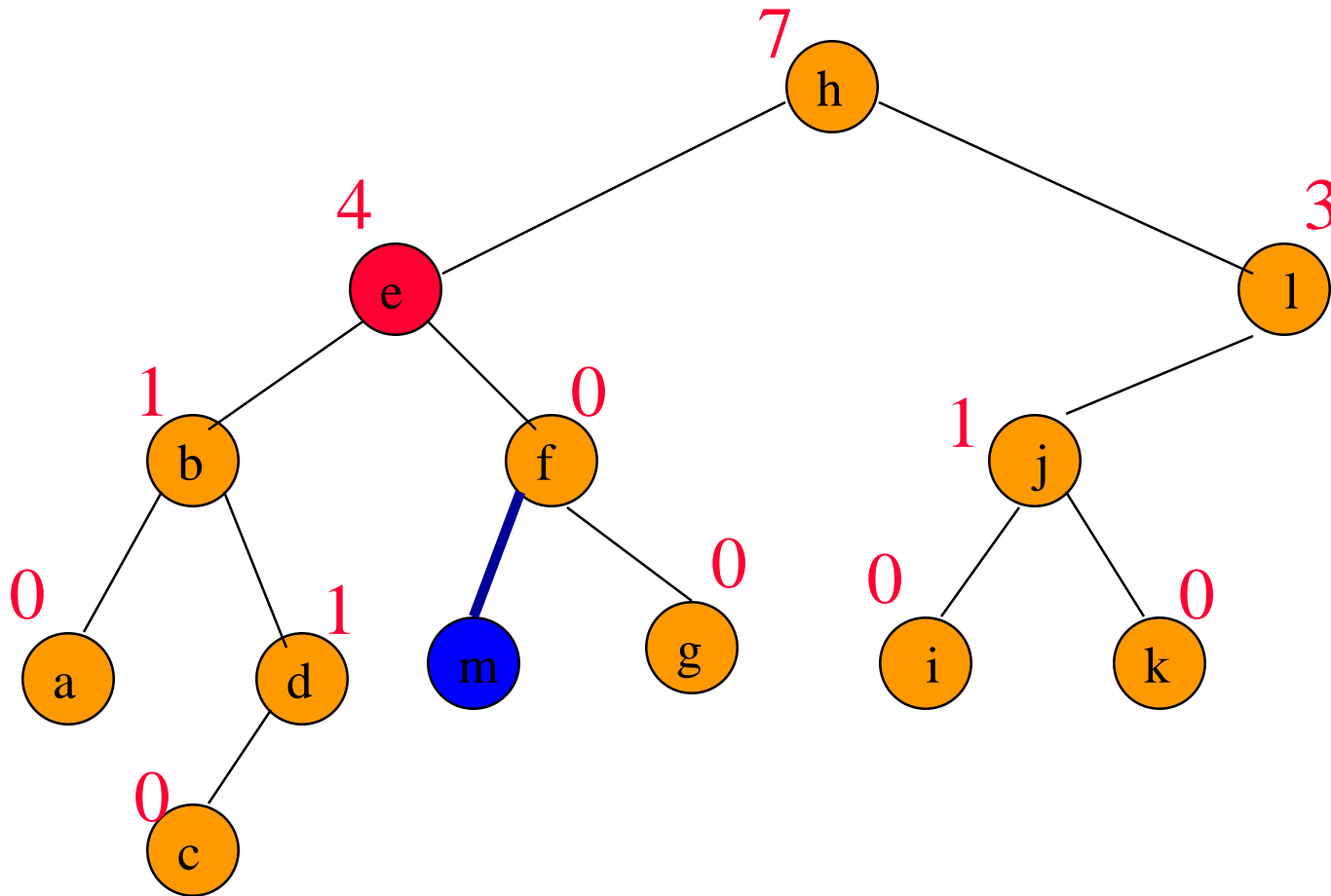
find node with element 4 (e)

add(5, 'm')



add **m** as right child of **e**; former right
subtree of **e** becomes right subtree of **m**

add(5, 'm')



add **m** as leftmost node in right subtree
of **e**

`add(5, 'm')`

- Other possibilities exist.
- Must update some **leftSize** values on path from root to new node.
- Complexity is **$O(\text{height})$** .