Tree-Successor(n)

if $n.right \neq None$

then return (Tree-Minimum($n.right$))

$n' \leftarrow n.parent$

while $n' \neq None$ and $n = n'.right$ → evaluates to true only at most

$n \leftarrow n'$

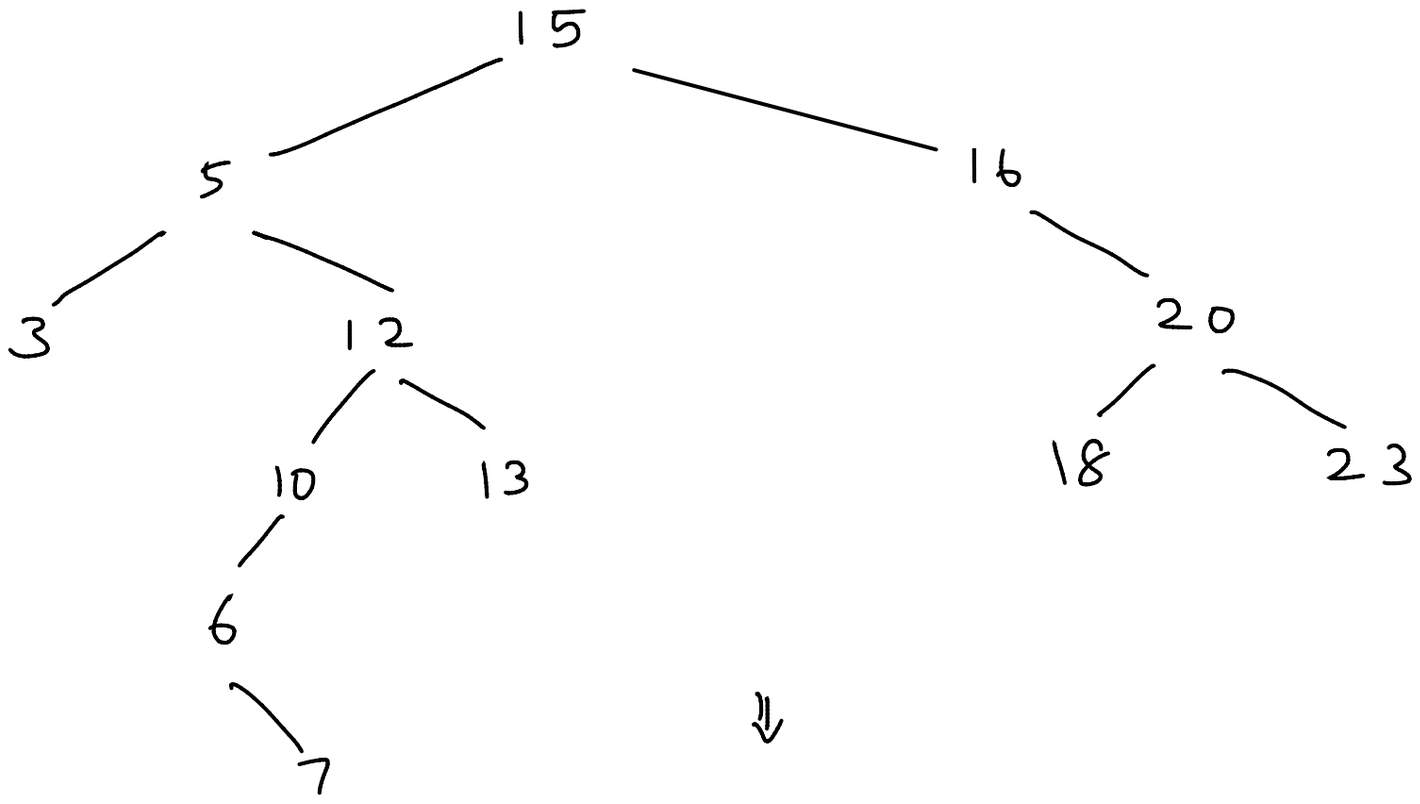
$n' \leftarrow n.parent$

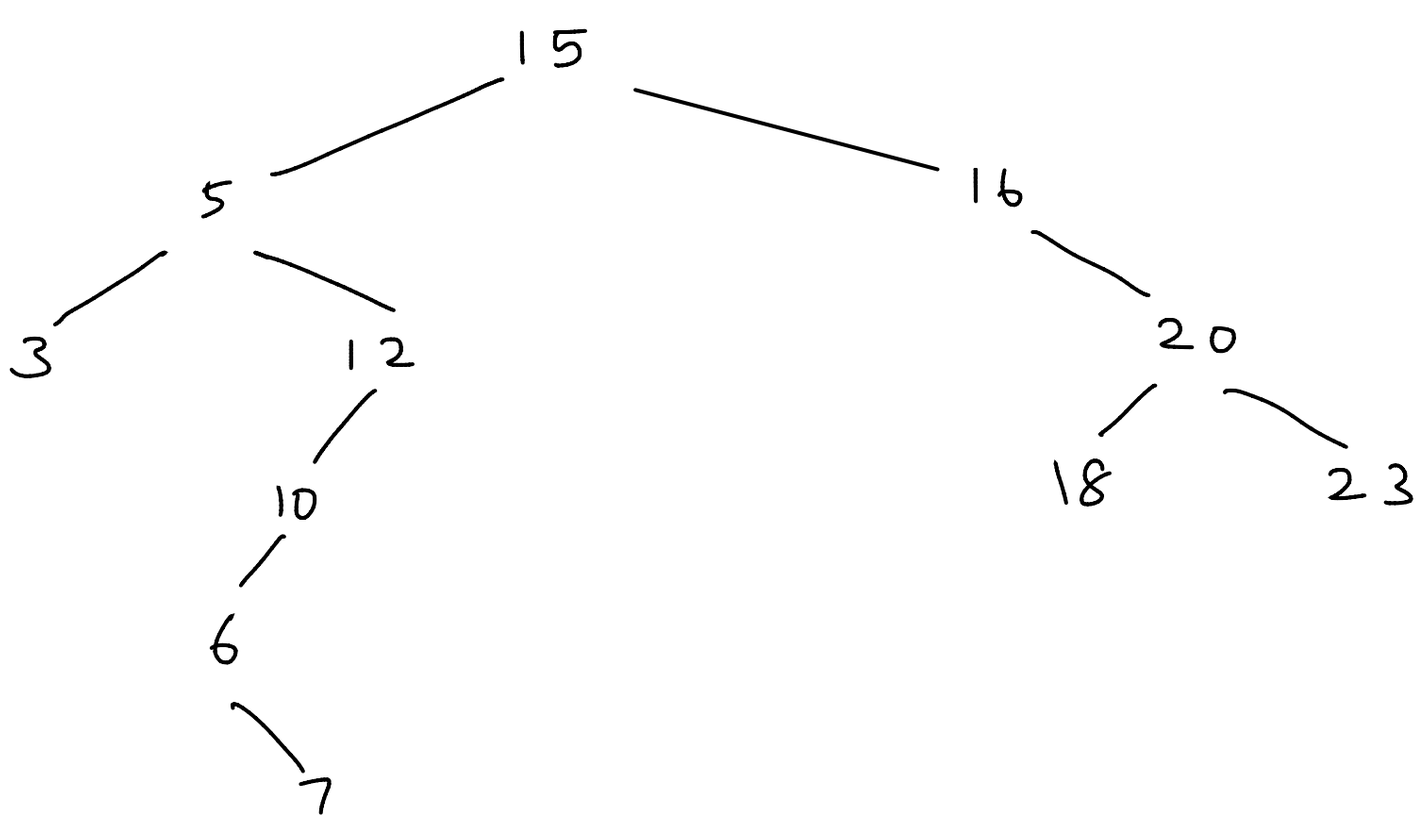
return n'

} climb up the tree until we find a left subtree

Deletion: Simplest Case Deletion at leaf Delete 13

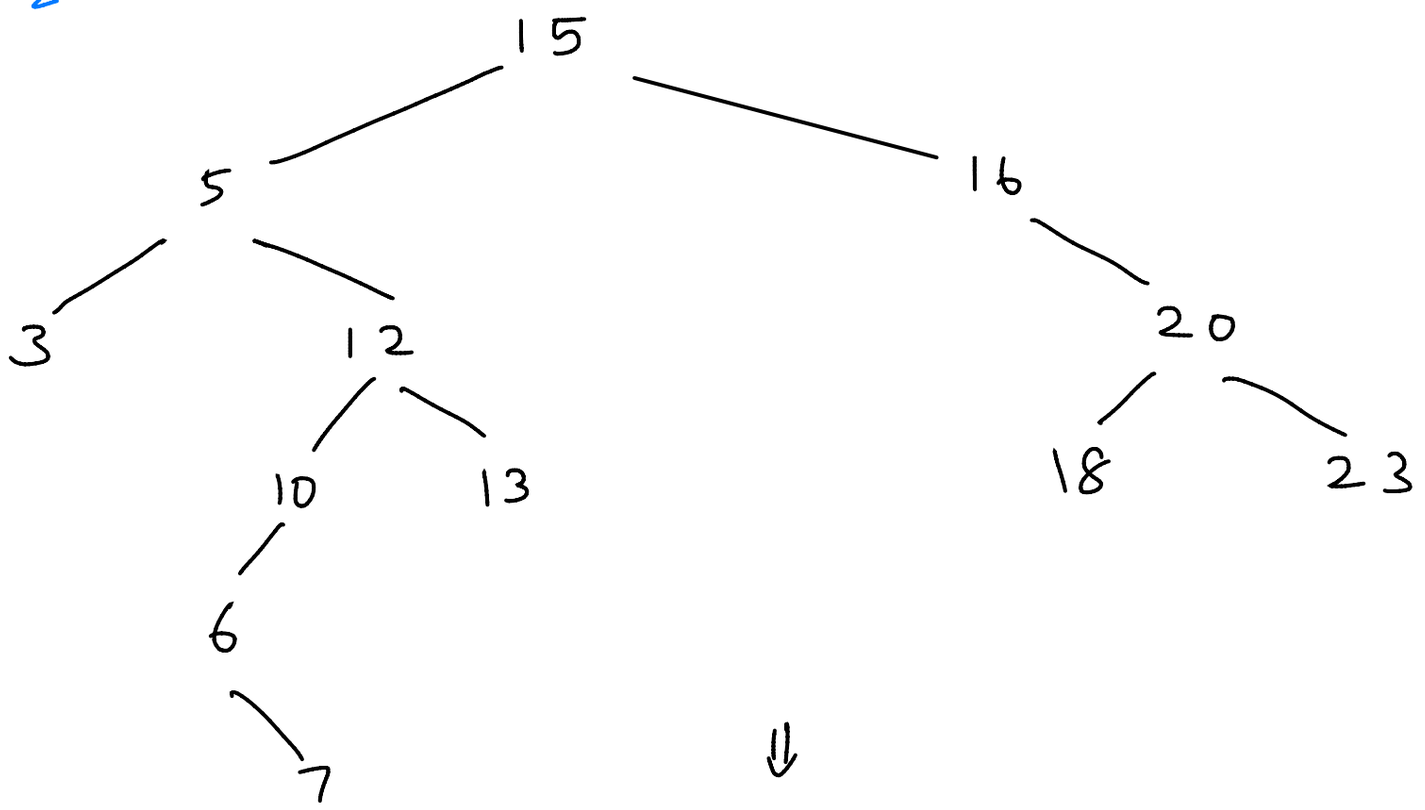
Case 1



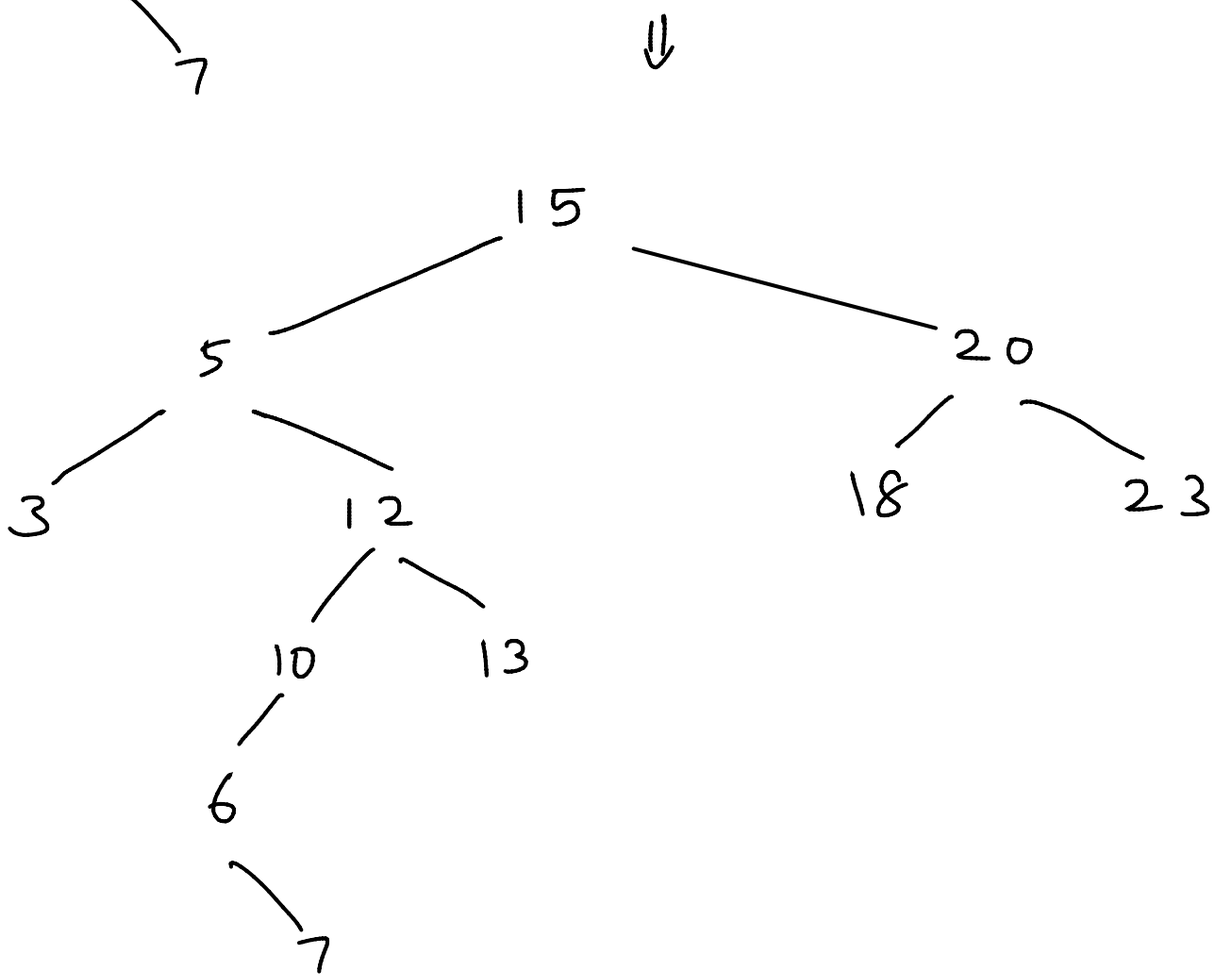


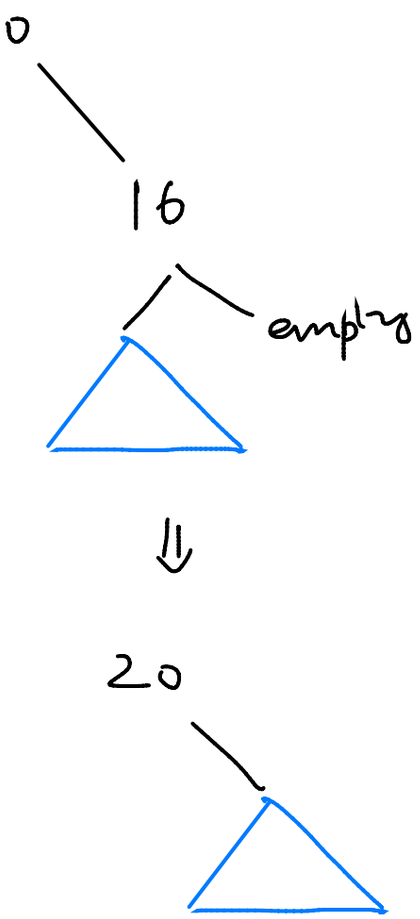
Deletion: Simplest case Deletion of node with one child Delete 16

case 2



Note: Only child is the important thing here.



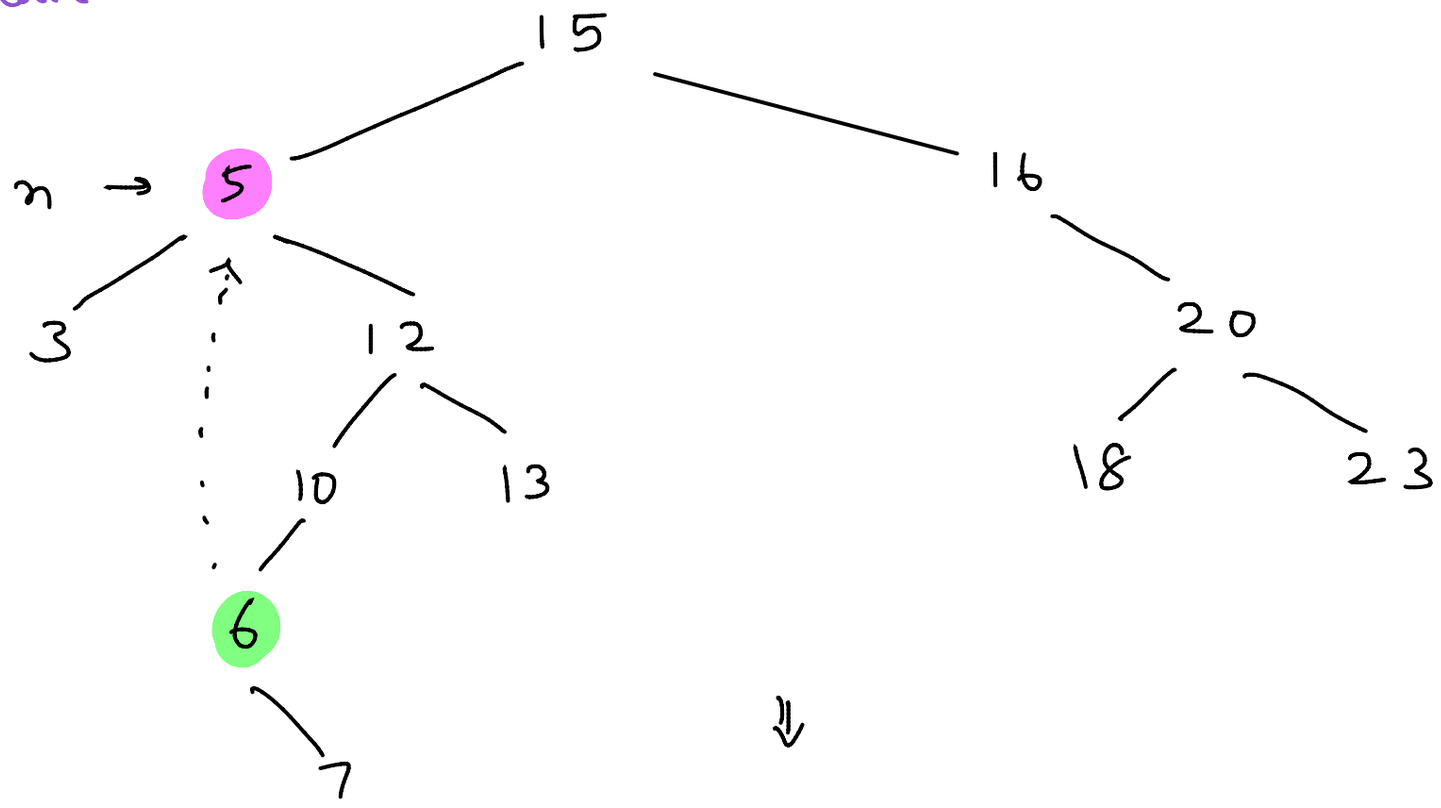


Deletion: Most involved case is deleting a node with two children

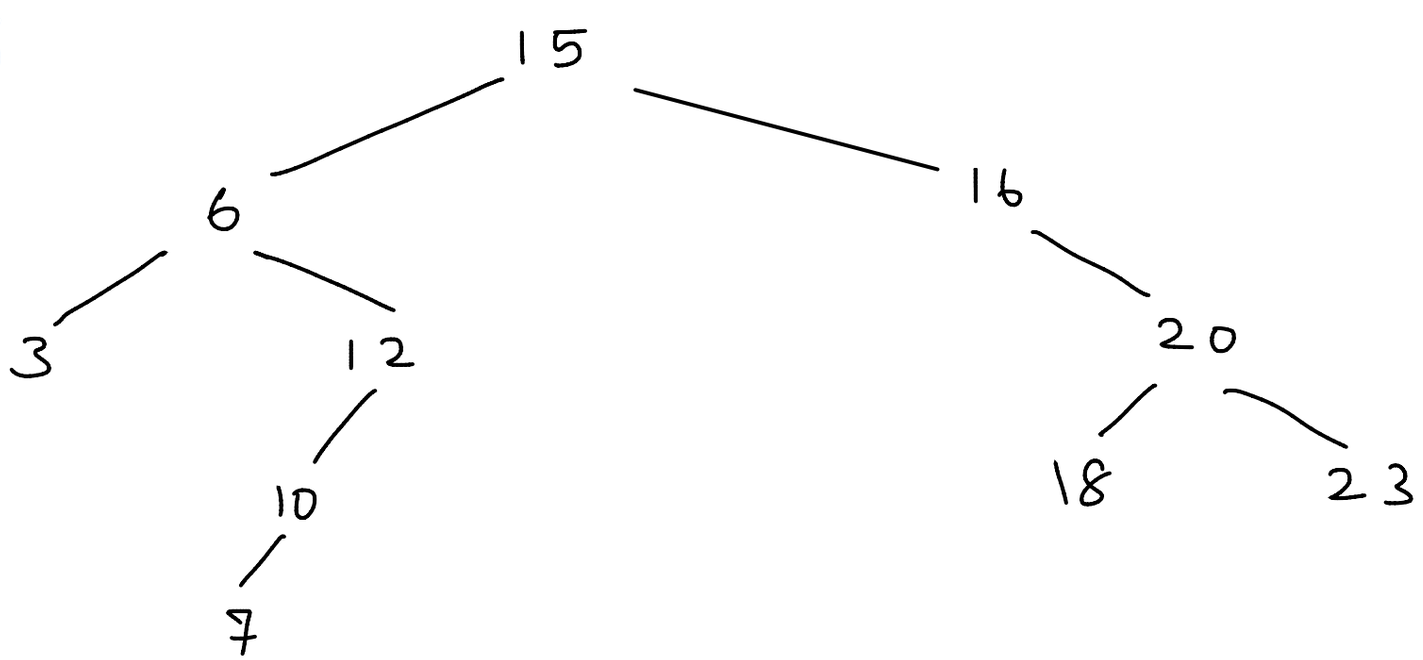
Case 3

Replace with successor

Delete 5



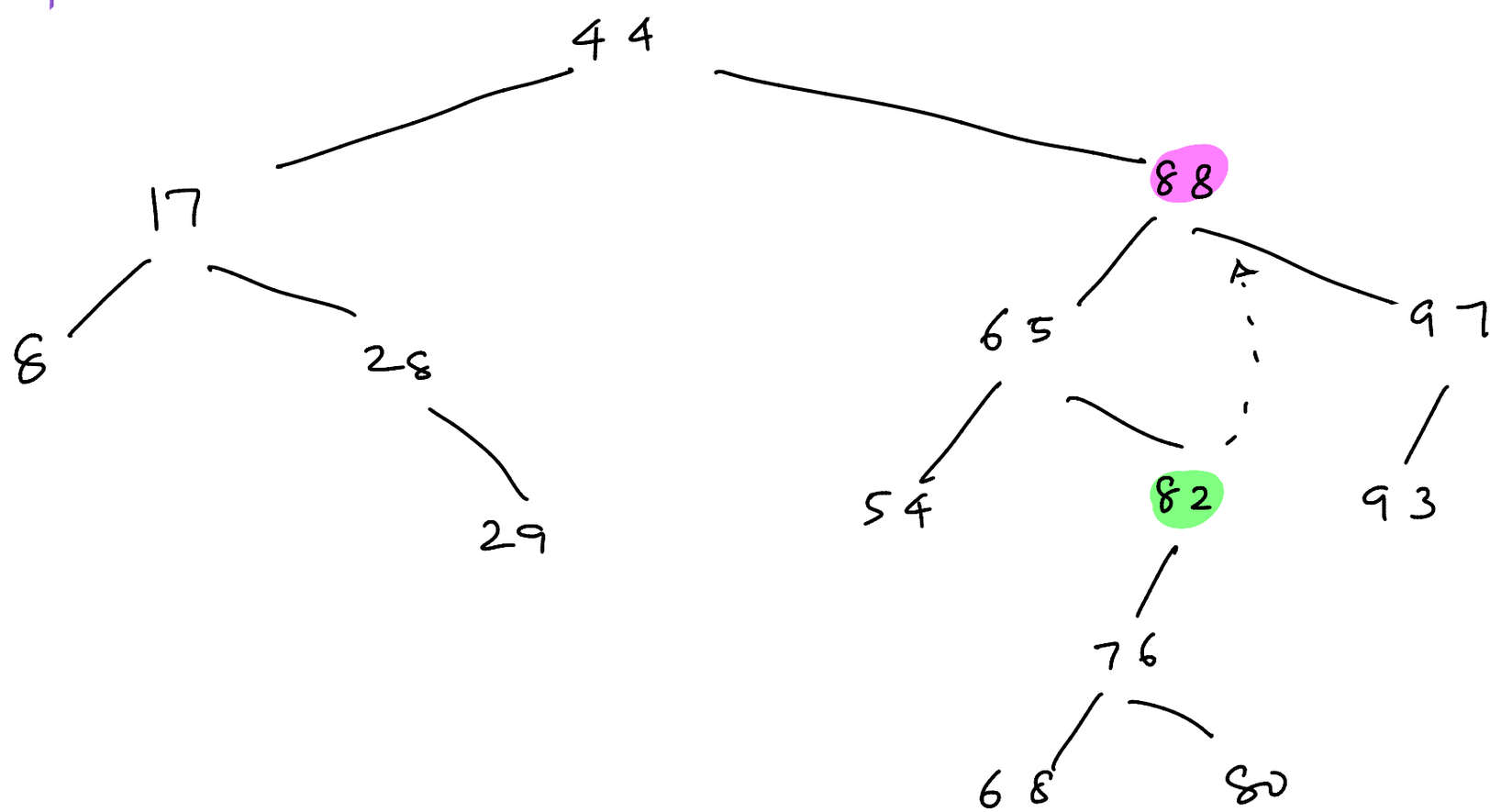
We know here the successor has no left subtree. Removing 6 is same as case 2



Deletion: Most involved case is deleting a node with two children

Case 3 Replace with predecessor

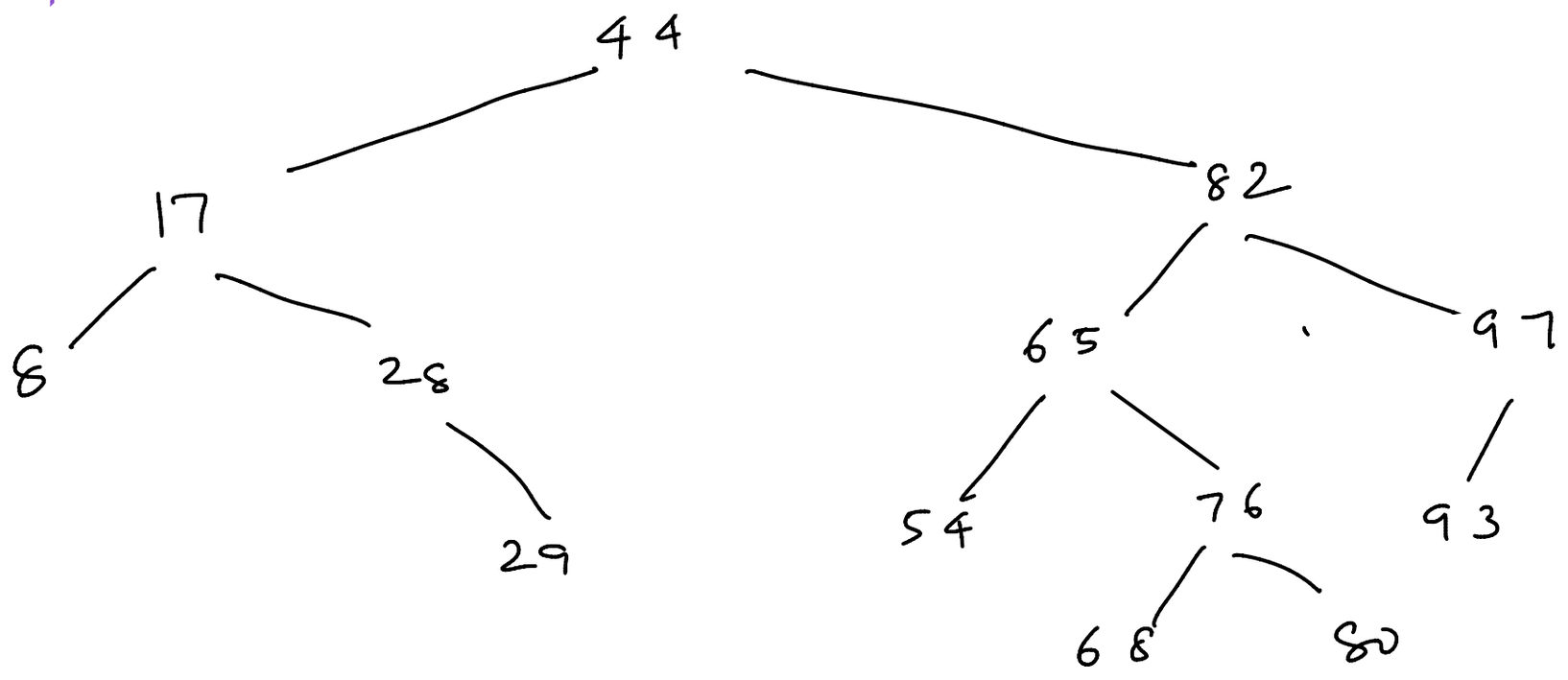
Delete 88



Deletion: Most involved case is deleting a node with two children

Case 3 Replace with predecessor

Delete 88



```
Tree-Delete (T, node)
  if   node.left = None   or   node.right = None
       then   node-to-remove ← node
       else   node-to-remove ← Tree-Successor(node)        has no left child
  if   node-to-remove.left ≠ None
       then   repeacing-node ← node-to-remove.left
       else   repeacing-node ← node-to.remove.right
  if   repeacing-node ≠ None
       then   repeacing-node.parent ← node-to-remove.parent
  if   node-to.remove.parent = None
       then   T.root ← repeacing-node
       else   if node-to-remove = node-to-remove.parent.left
              then node-to-remove.parent.left ← repeacing-node
              else node-to-remove.parent.right ← repeacing-node
```

case 3

if node-to-remove ≠ node

then node.key ← node-to-remove.key

node.value ← node-to-remove.value

return node-to-remove