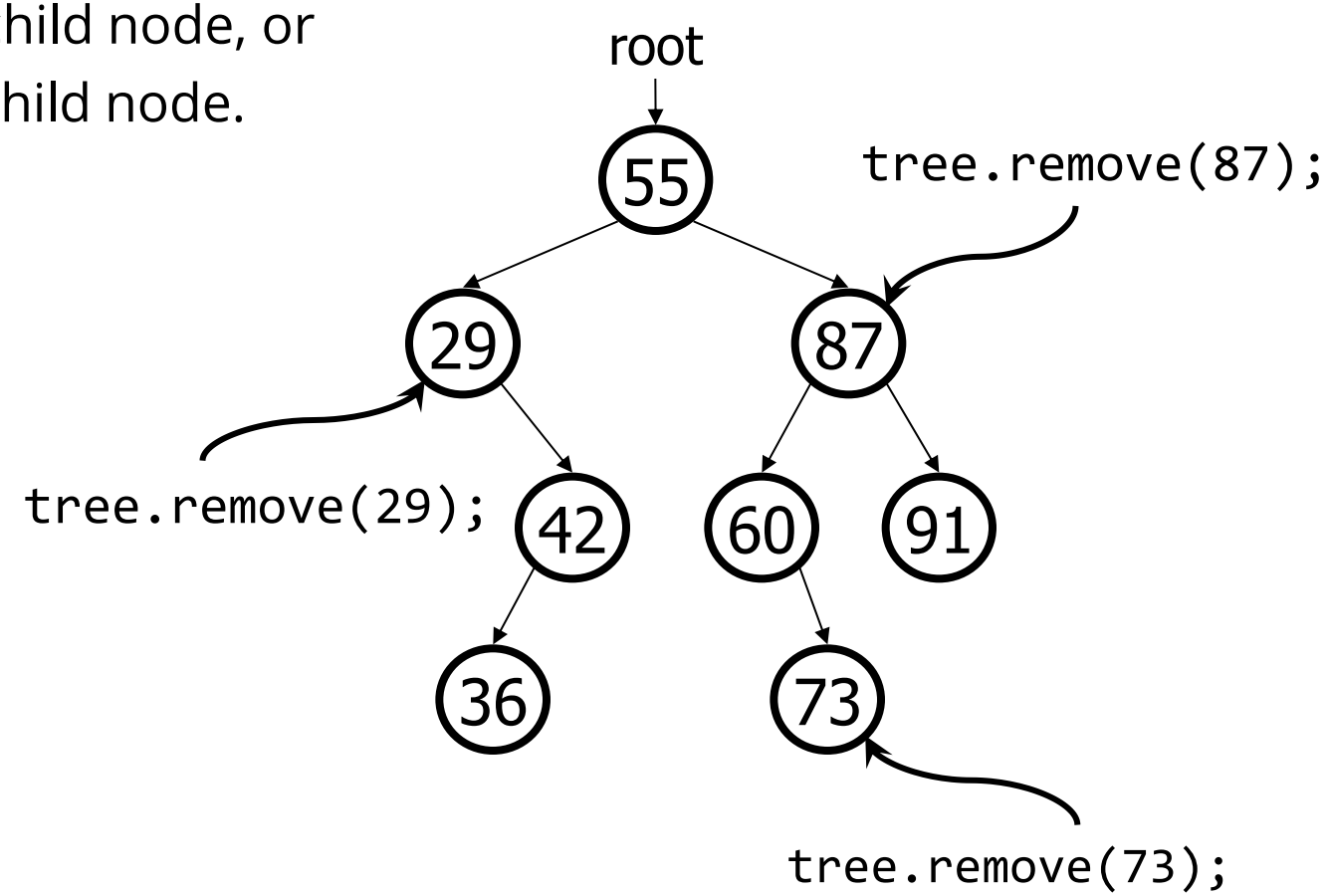
COSC 222 Data Structure

Trees – Deletion in BSTs

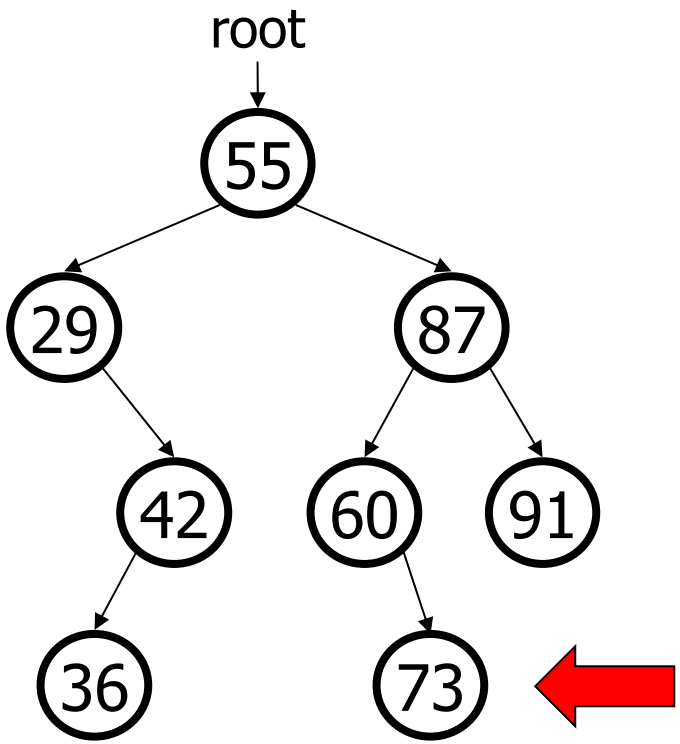
Deletion in BSTs

- First step: Search for the item to be deleted.
- Once you've found the item, there are three possibilities: It's in
  - A leaf,
  - A one-child node, or
  - A two-child node.

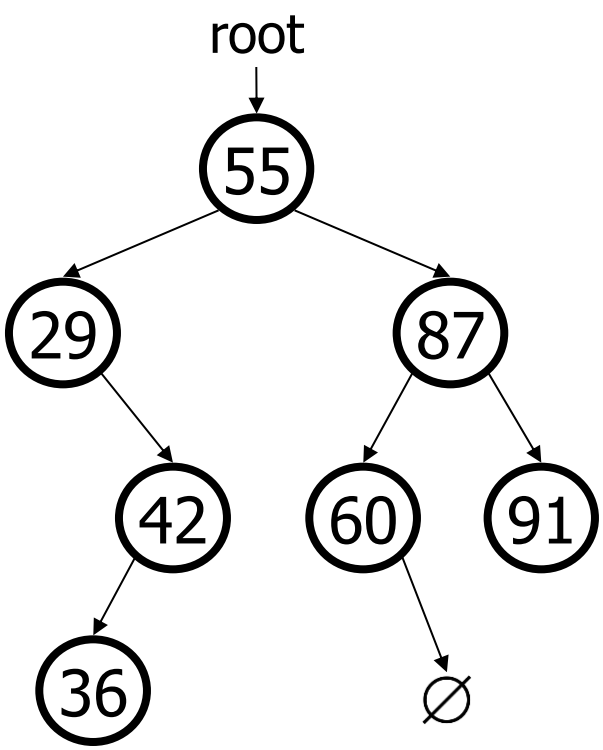


Deletion Case 1: Deleting a Leaf

- Easy! Simply set the parent's child pointer to null.



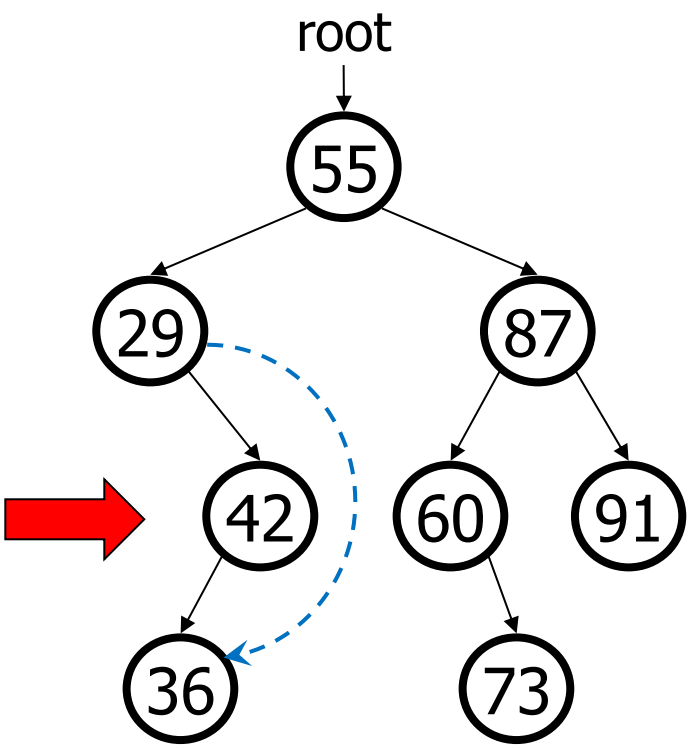
Example: To delete 73 from the following tree . . .



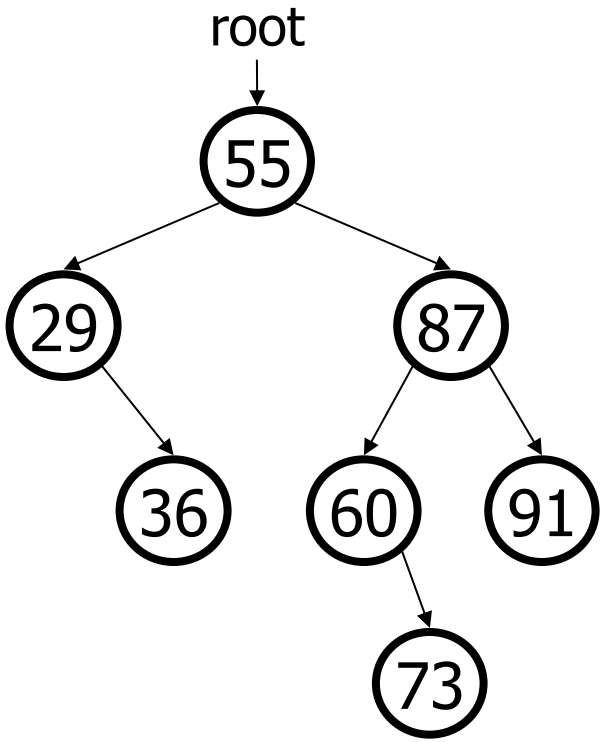
. . simply set the parent's right child pointer to null:

Deletion Case 2: Deleting a One-Child Node

- Easy! Simply set the parent's child pointer to null.



Example: To delete 42 from the following tree . . .



. . . simply make the right child pointer of 29 (42's parent) point at 36 (29's sole child):

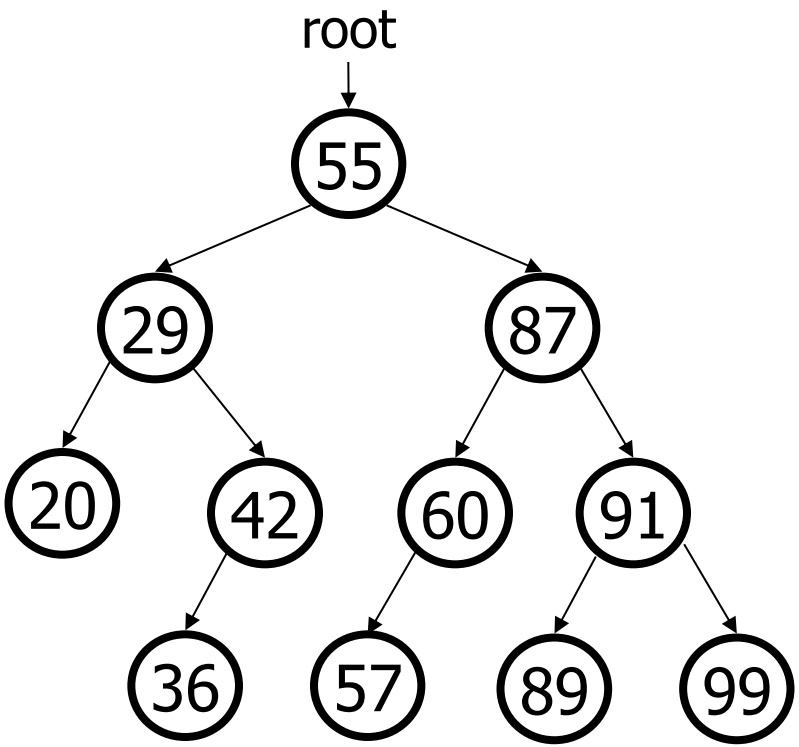
Deletion Case 3: Deleting a Two-Child Node

- To delete a node with two children: A little more difficult!

- The inorder successor of a node *n* is the node n_{is} containing the next higher value in the tree.

Deletion Case 3: Deleting a Two-Child Node

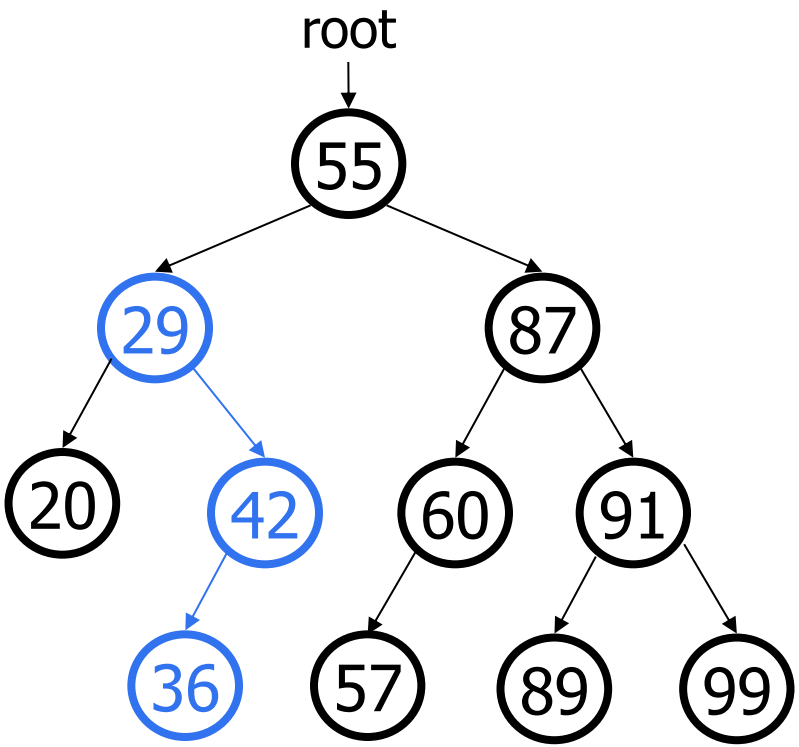
- Example: In the following tree, give the inorder successors below:



Node	29	55	87	91
Inorder Successor	36	57	89	99

Deletion Case 3: Deleting a Two-Child Node

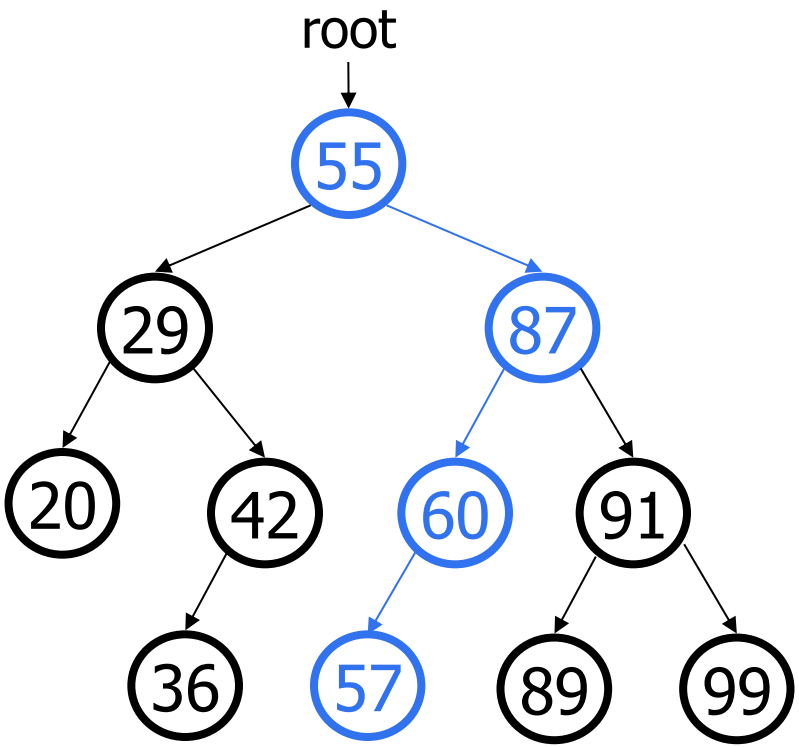
- Example: In the following tree, give the inorder successors below:



Node	29	55	87	91
Inorder Successor	36	57	89	99

Deletion Case 3: Deleting a Two-Child Node

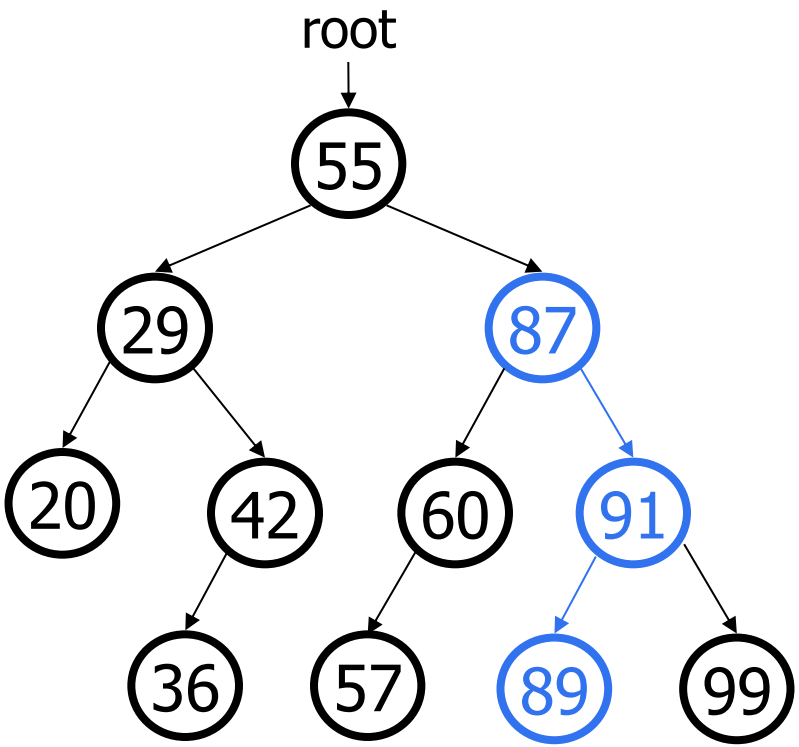
- Example: In the following tree, give the inorder successors below:



Node	29	55	87	91
Inorder Successor	36	57	89	99

Deletion Case 3: Deleting a Two-Child Node

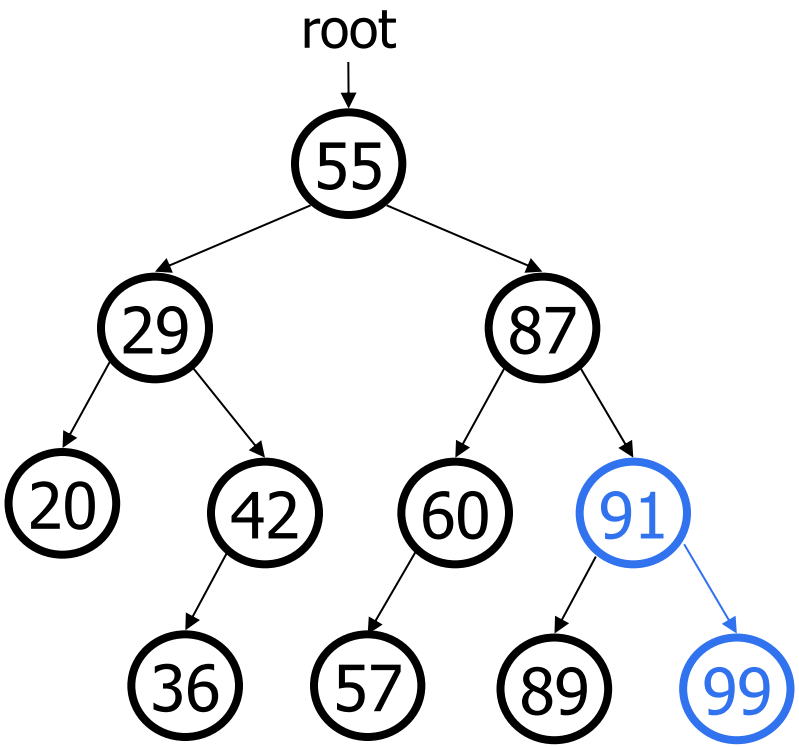
- Example: In the following tree, give the inorder successors below:



Node	29	55	87	91
Inorder Successor	36	57	89	99

Deletion Case 3: Deleting a Two-Child Node

- Example: In the following tree, give the inorder successors below:



Node	29	55	87	91
Inorder Successor	36	57	89	99

Deletion in BSTs

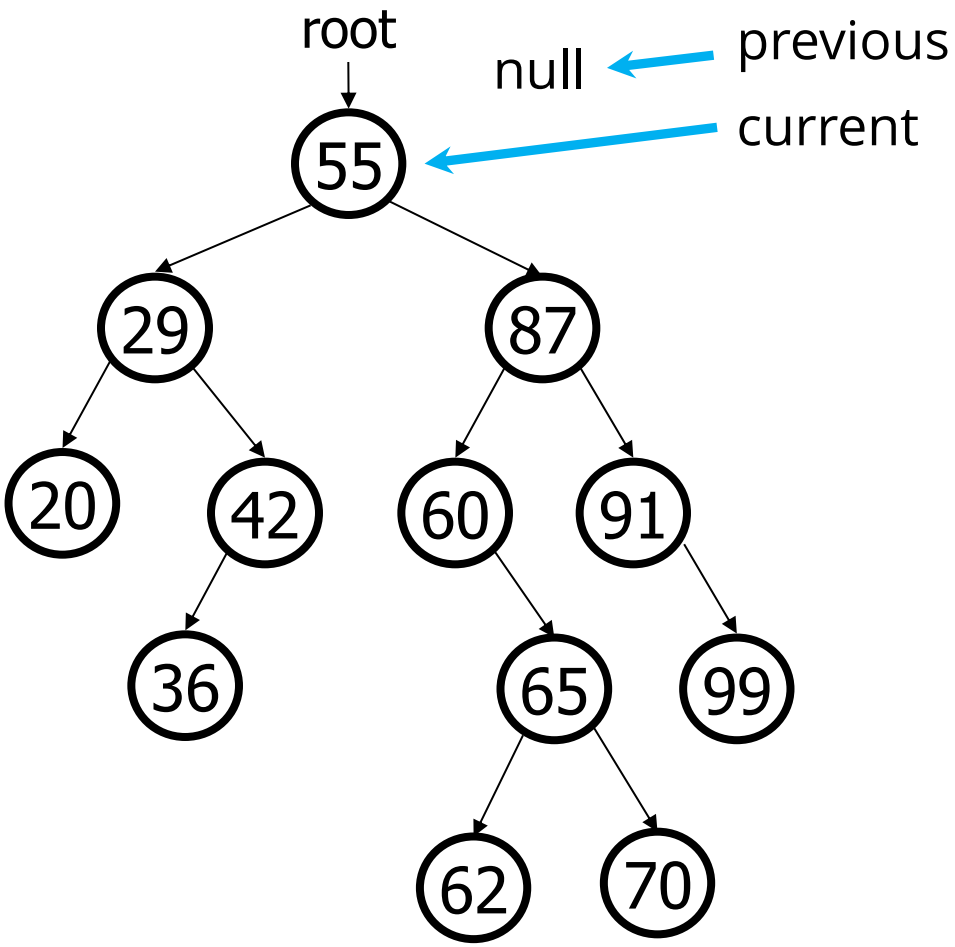
- If n is a node with two children, how can we find *n*'s inorder successor, n_{is}?
  - Go to *n*'s **right** child.
  - Then follow **left pointers** until the left pointer is **null**.
- The last node visited is n_{is} , the inorder successor of *n*.

Deletion Case 3: Deleting a Two-Child Node

- To delete a node *n* with two children:
- Find the inorder successor n_{is} of *n*.
  - To find the inorder successor: go to the right child of n and then go left as far as possible.
- Replace the contents of the node n to be deleted with the contents of its inorder successor n_{is} .
- Delete the inorder successor node n_{is} . (A guaranteed easy case because nis has no left child.)

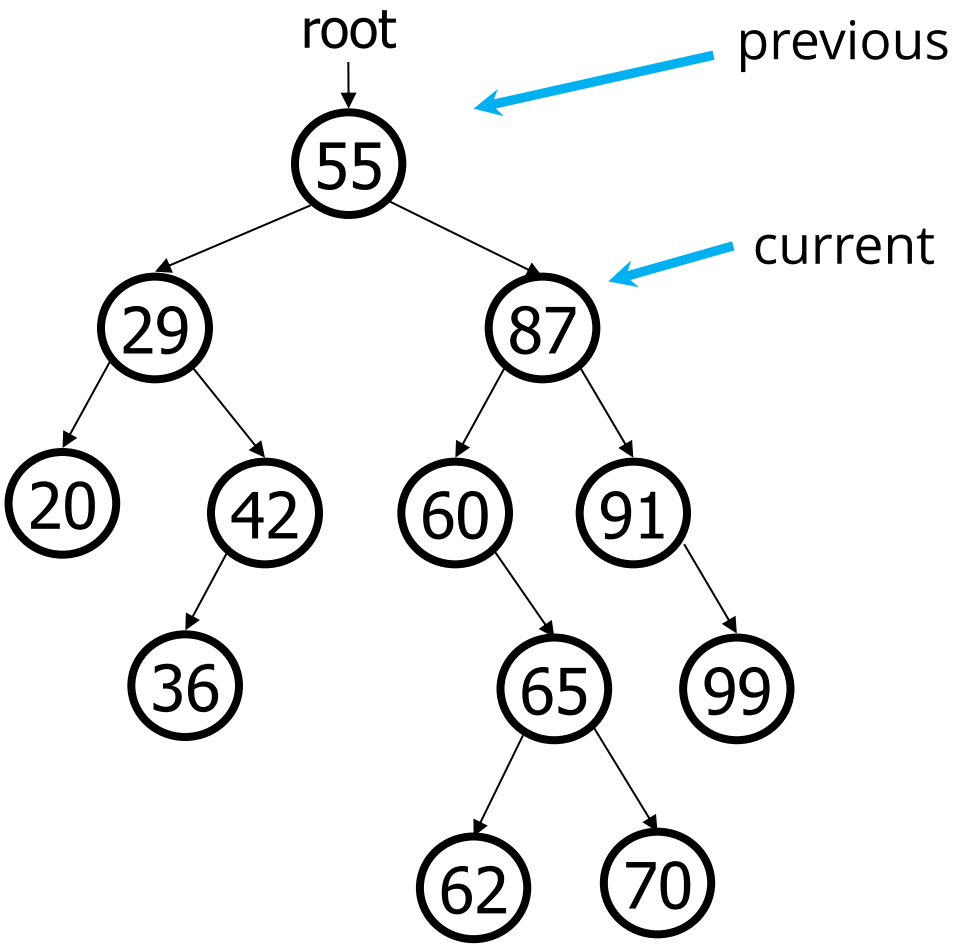
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



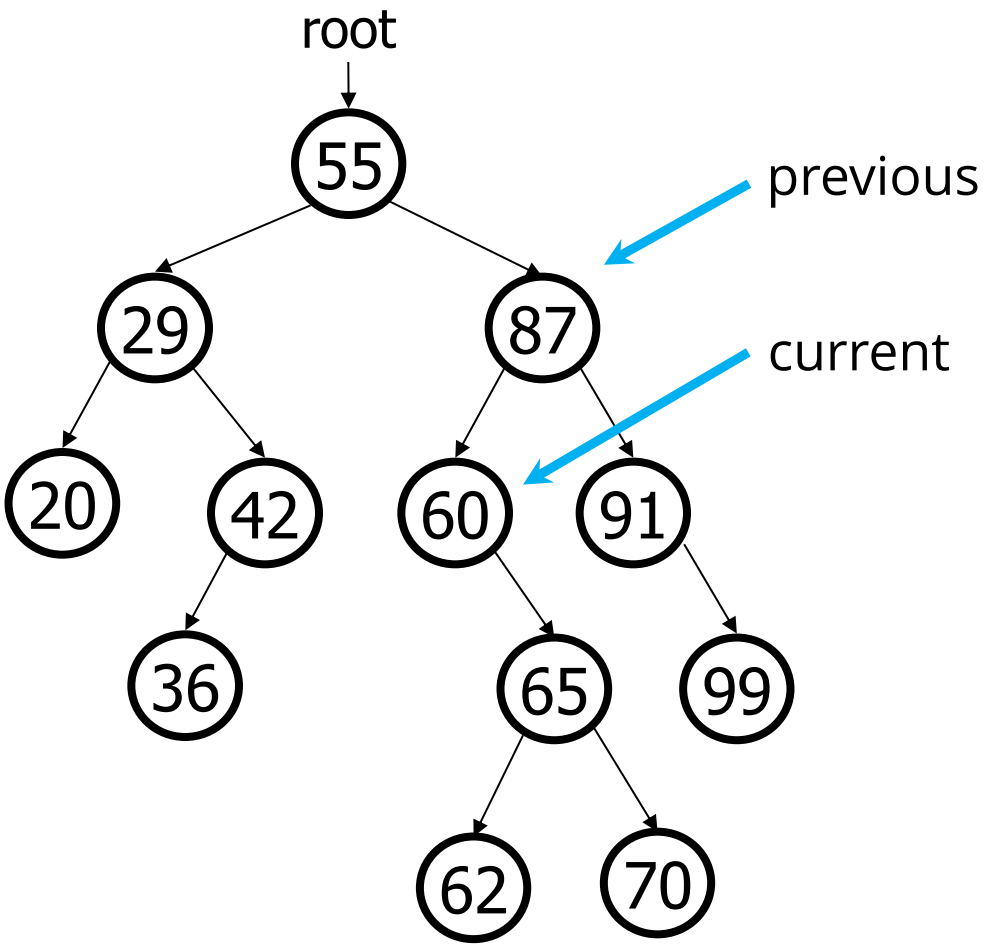
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



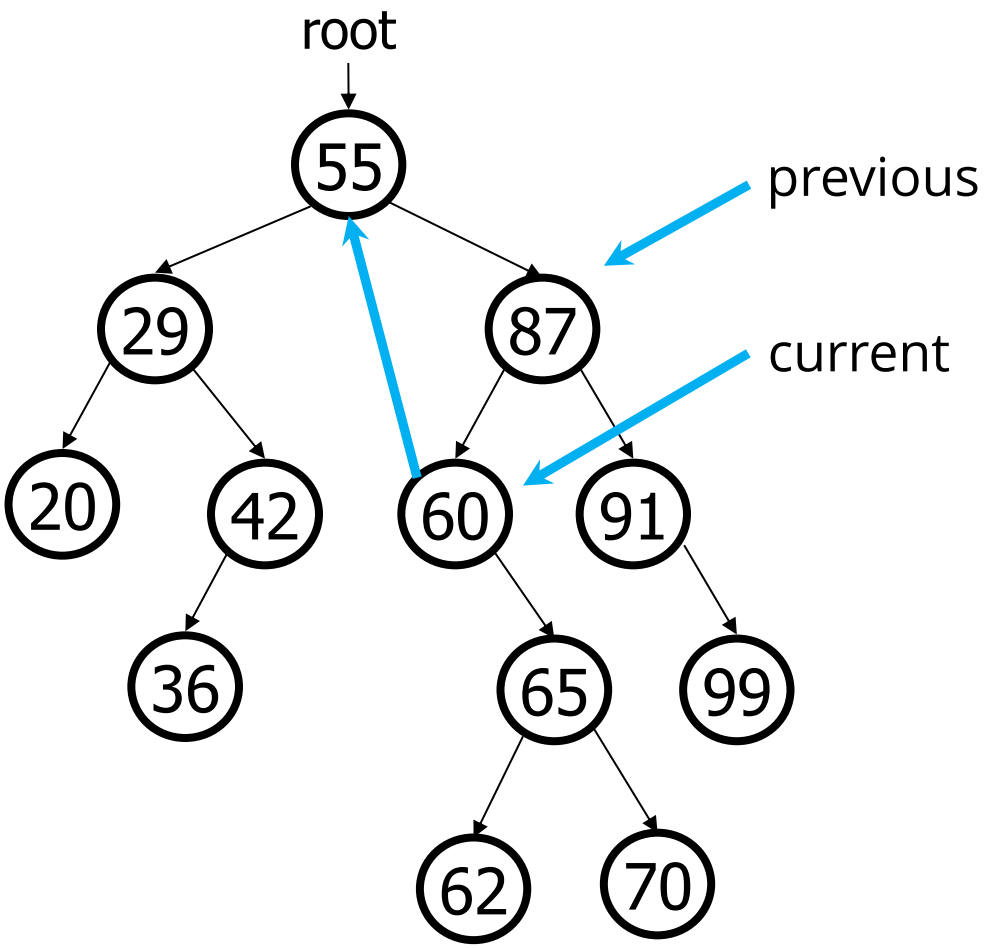
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



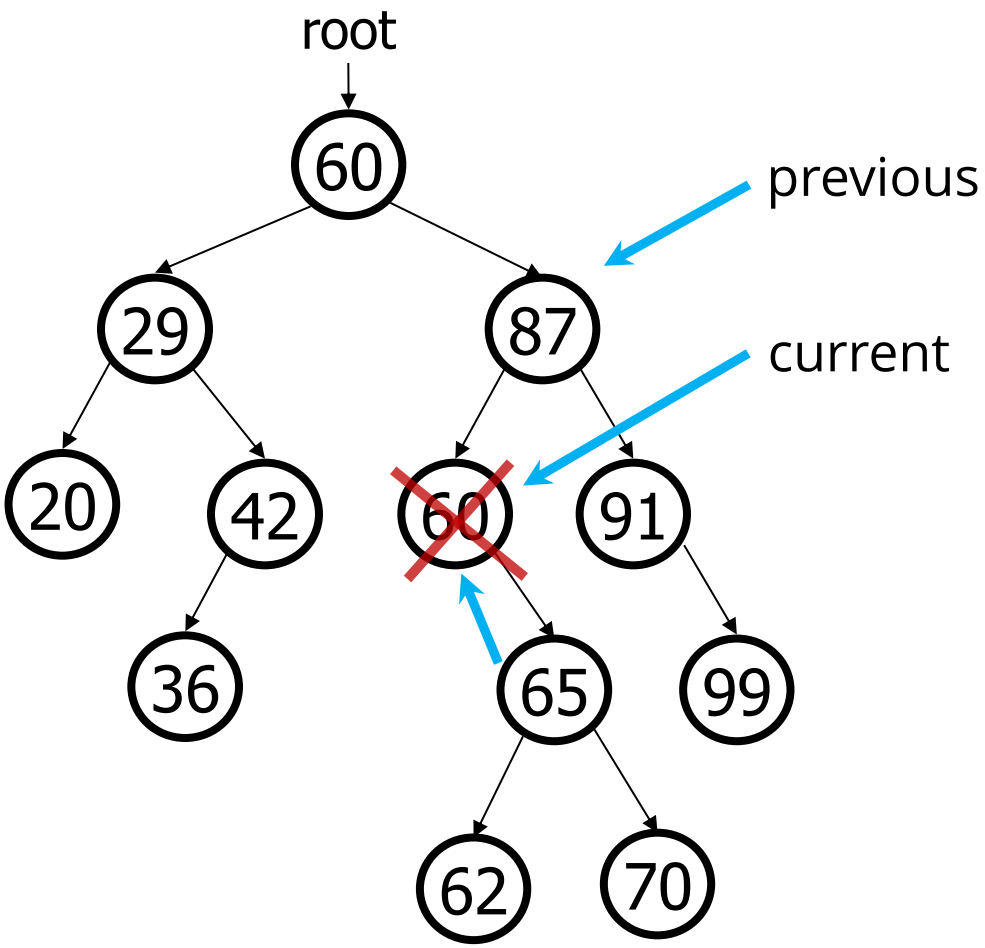
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



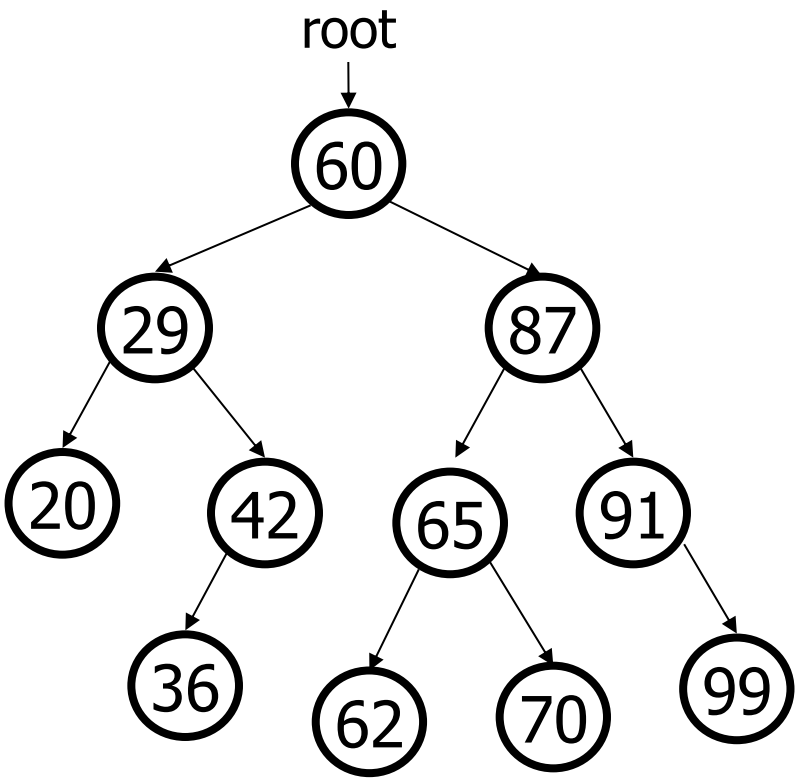
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



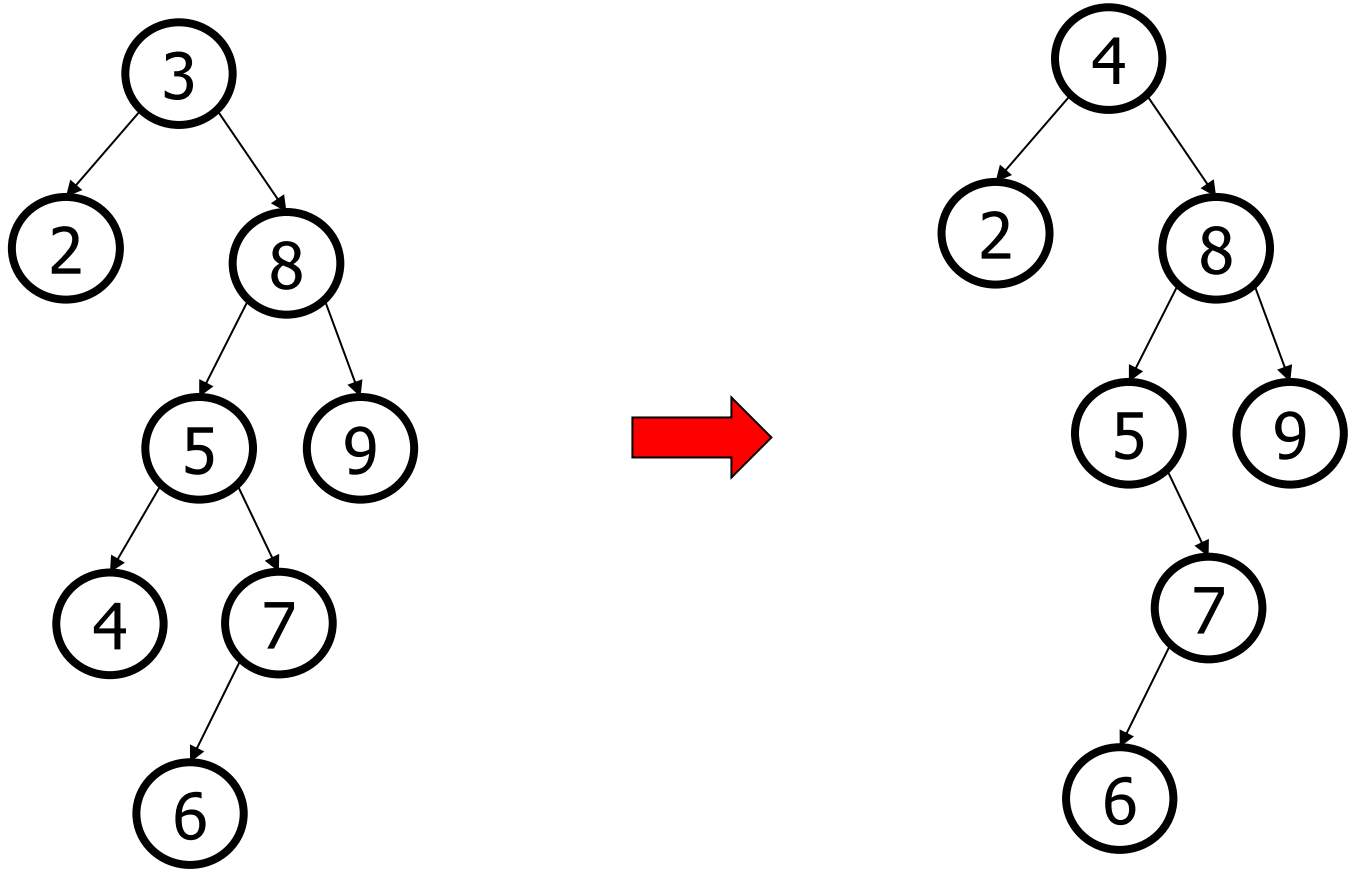
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



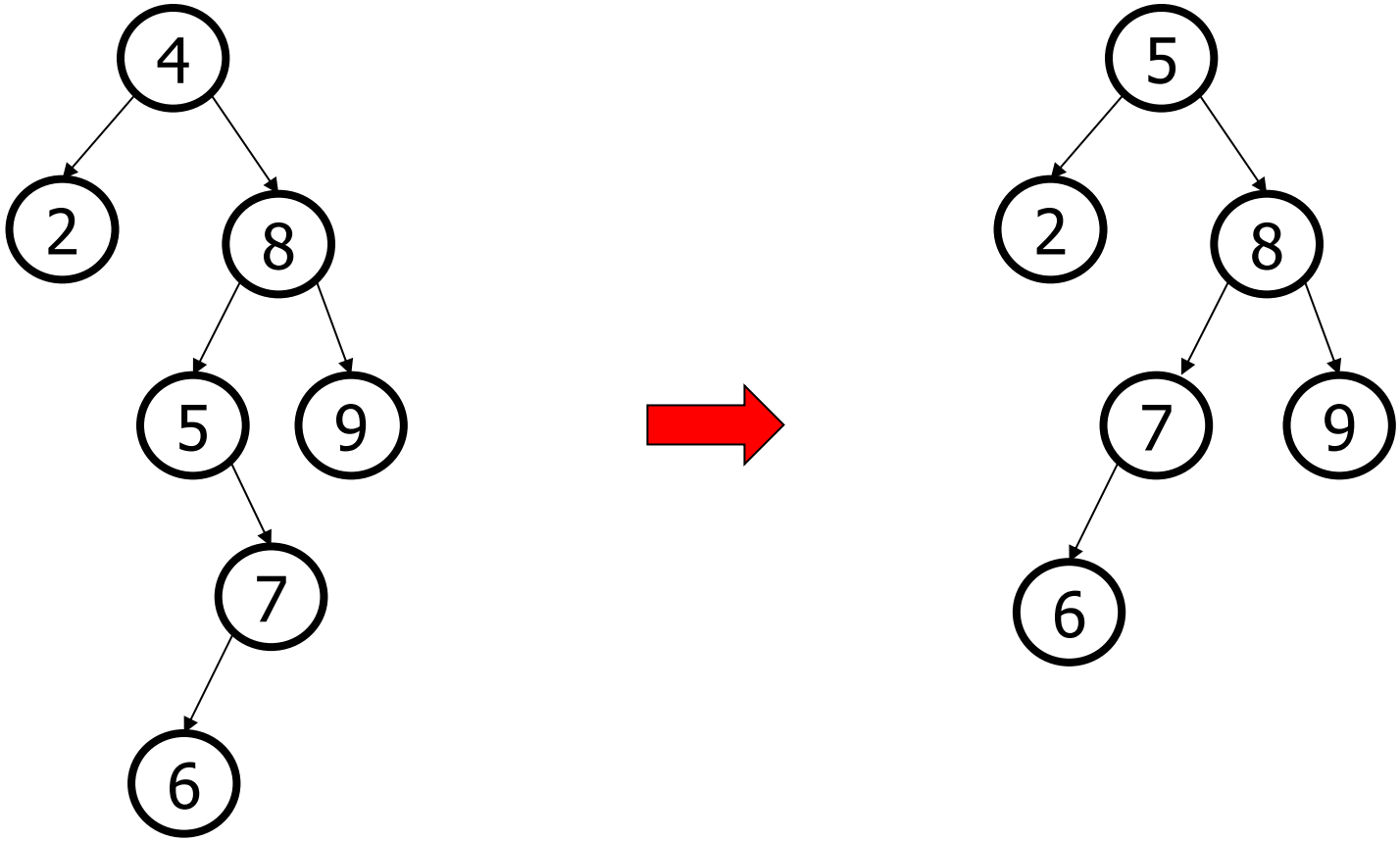
Deletion in BSTs

- Example: Delete the value in the root two times in succession
- (draw the tree after each deletion): **delete 3**



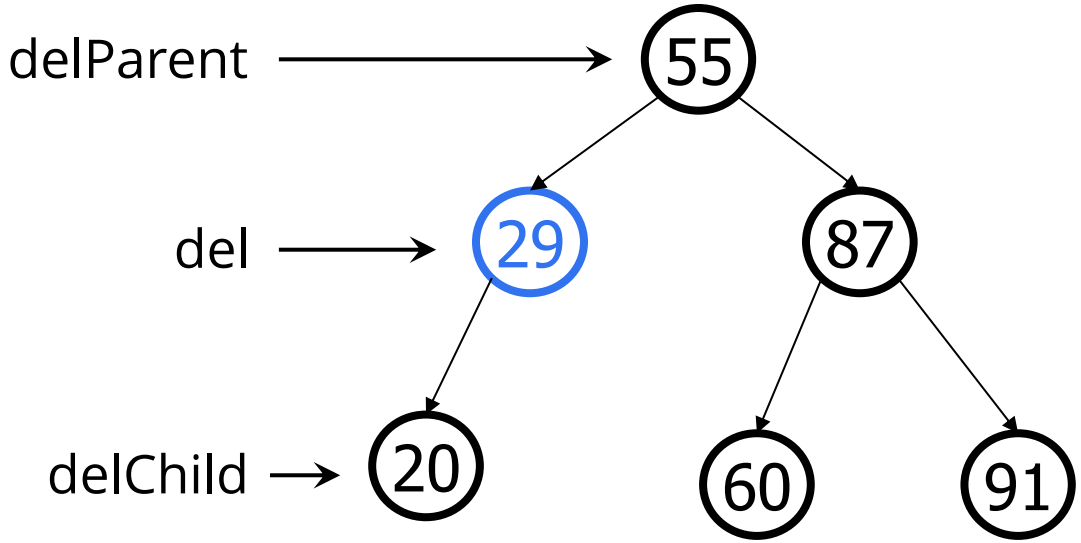
Deletion in BSTs

- Example: Delete the value in the root two times in succession
- (draw the tree after each deletion): **delete 4**



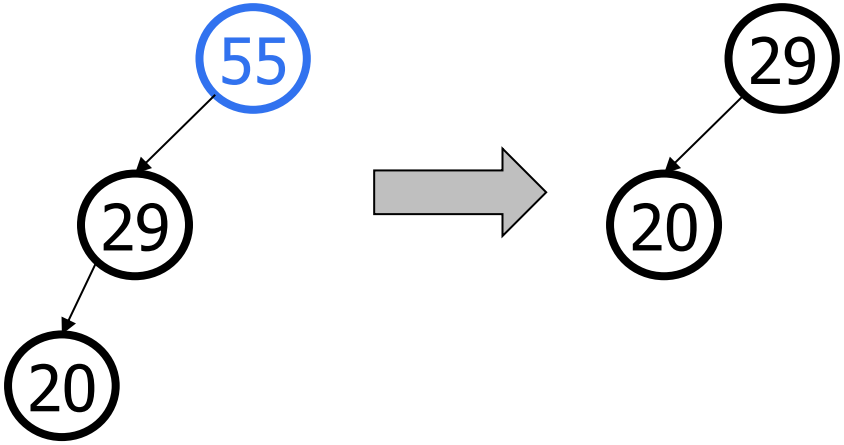
Deletion Implementation

- A helper method for the **easy cases**: Delete a node, given
  - *del*: the node to be deleted (which must have at most one child)
  - *delParent*: its parent and
  - *delChild*: its sole child (or null if it has no children)



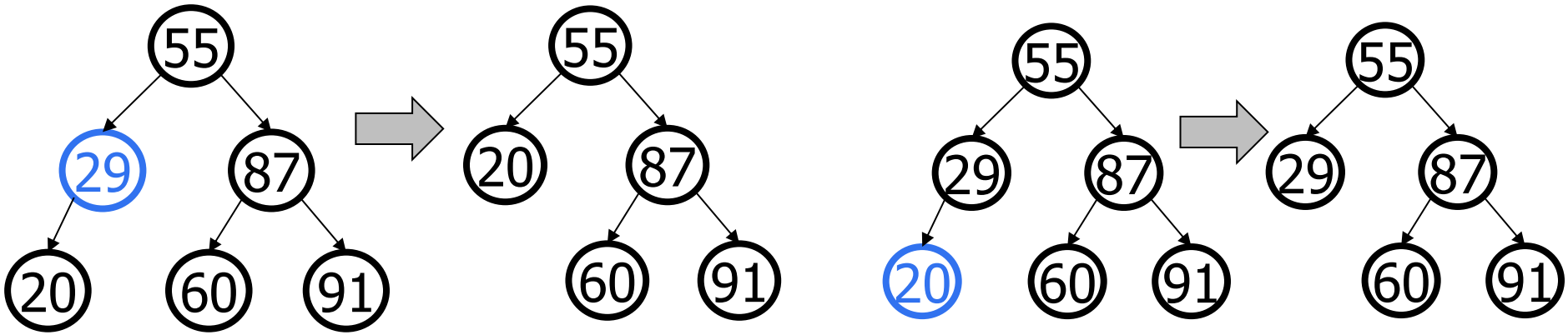
Deletion Implementation

```
public void easyDelete(Node del, Node delParent, Node
delChild){
    if (delParent == null){
        root = delChild;
    } else {
        if (del == delParent.left)
            delParent.left = delChild;
        else
            delParent.right = delChild;
    }
}
```



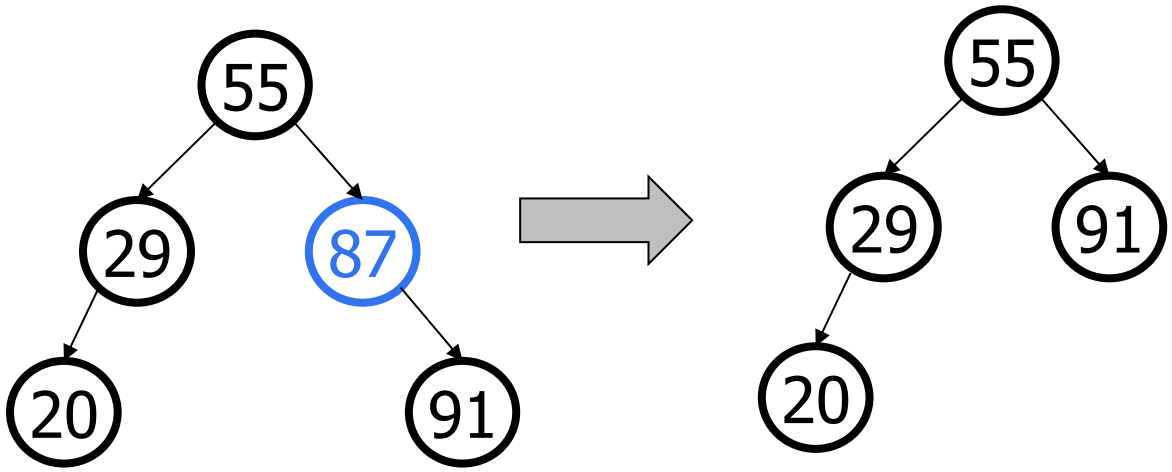
Deletion Implementation

```
public void easyDelete(Node del, Node delParent, Node
delChild){
    if (delParent == null){
        root = delChild;
    } else {
        if (del == delParent.left)
            delParent.left = delChild;
        else
            delParent.right = delChild;
    }
}
```



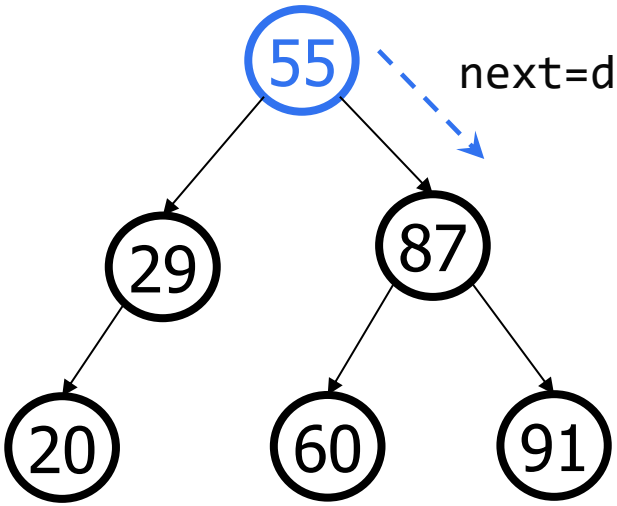
Deletion Implementation

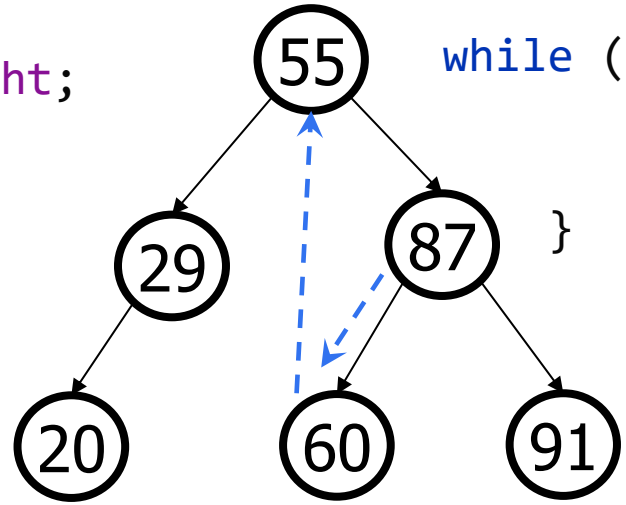
```
public void easyDelete(Node del, Node delParent, Node
delChild){
    if (delParent == null){
        root = delChild;
    } else {
        if (del == delParent.left)
            delParent.left = delChild;
        else
            delParent.right = delChild;
    }
}
```



Deletion Implementation: Two-children Case

```
public void twoChildrenDelete(Node del){
    Node previous, next;
    previous = del;
    next=del.right;
    while (next.left != null) {
        previous = next;
        next= next.left;
    }
    del.key = next.key;
    easyDelete(next,previous,next.right);
}
```






```
while (next.left != null){
    previous = next;
    next= next.left;
}
```

Deletion Implementation: Iteratively

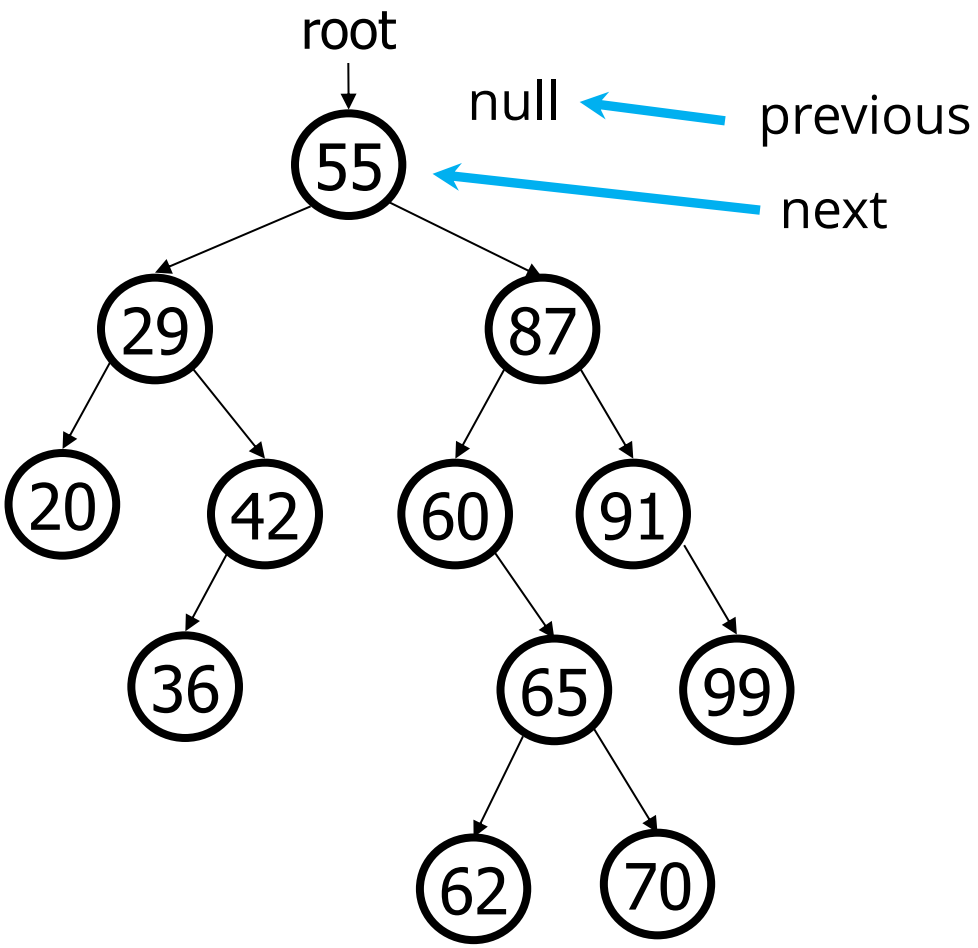
```
public void delete(int key){
    Node previous = null;
    Node current = root;

    while(current!=null && current.key != key){
        previous = current;
        if (key < current.key)
            current = current.left;
        else
            current = current.right;
    }

    if (current!=null){
        if (current.left == null)
            easyDelete(current,previous,current.right);
        else if (current.right == null)
            easyDelete(current,previous,current.left);
        else
            twoChildrenDelete(current);
    }
}
```

Search for the key in the tree

Delete Conditions

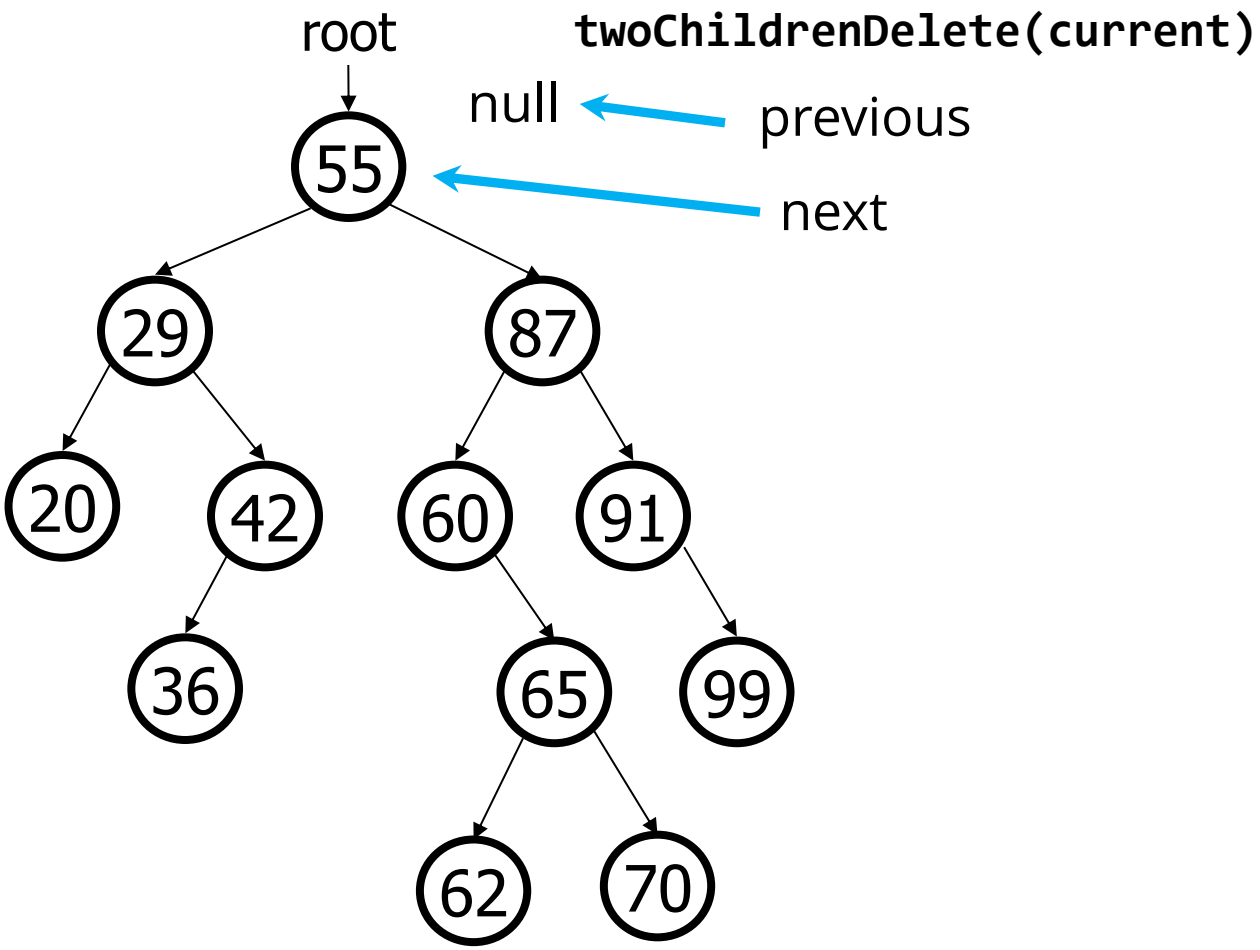
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



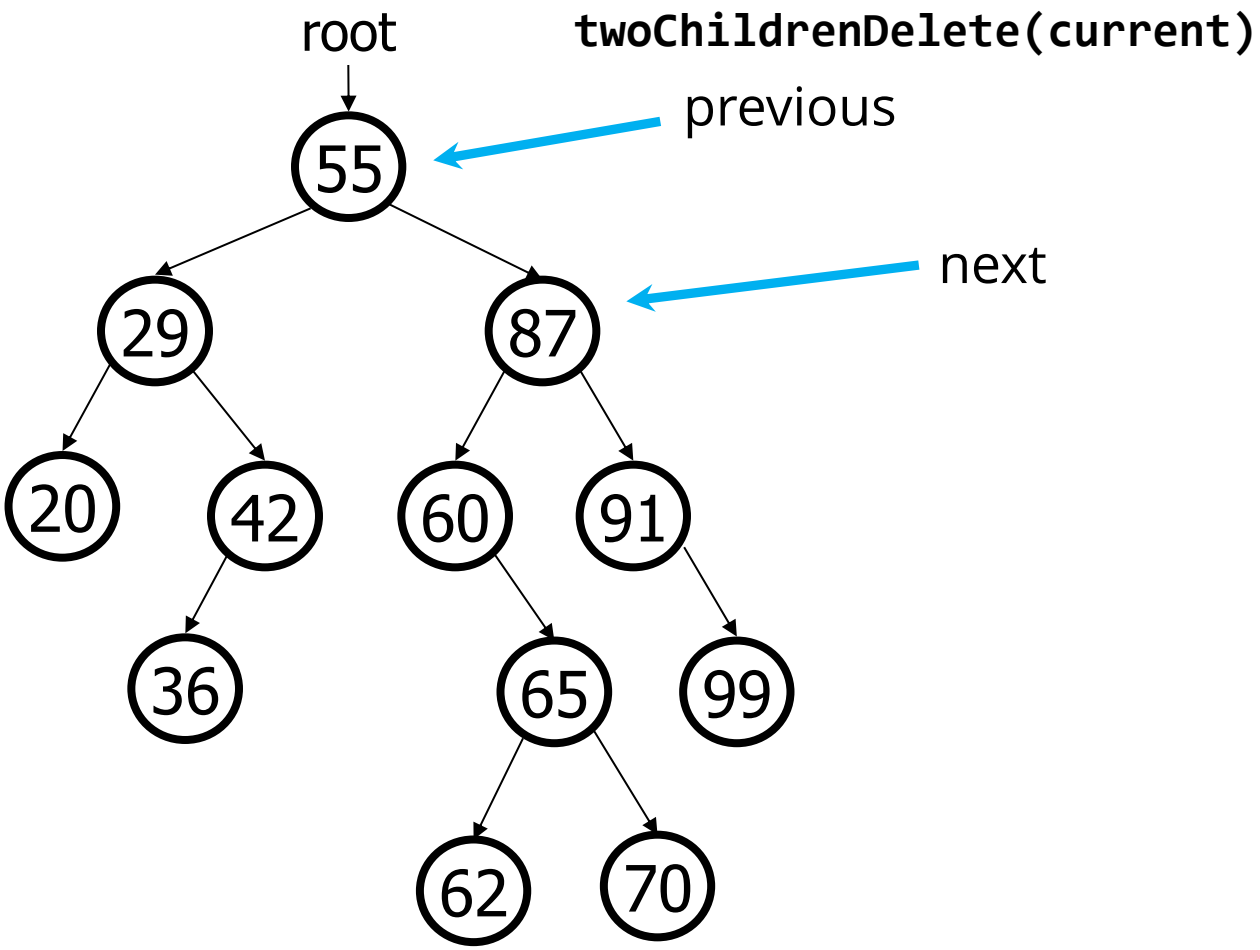
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



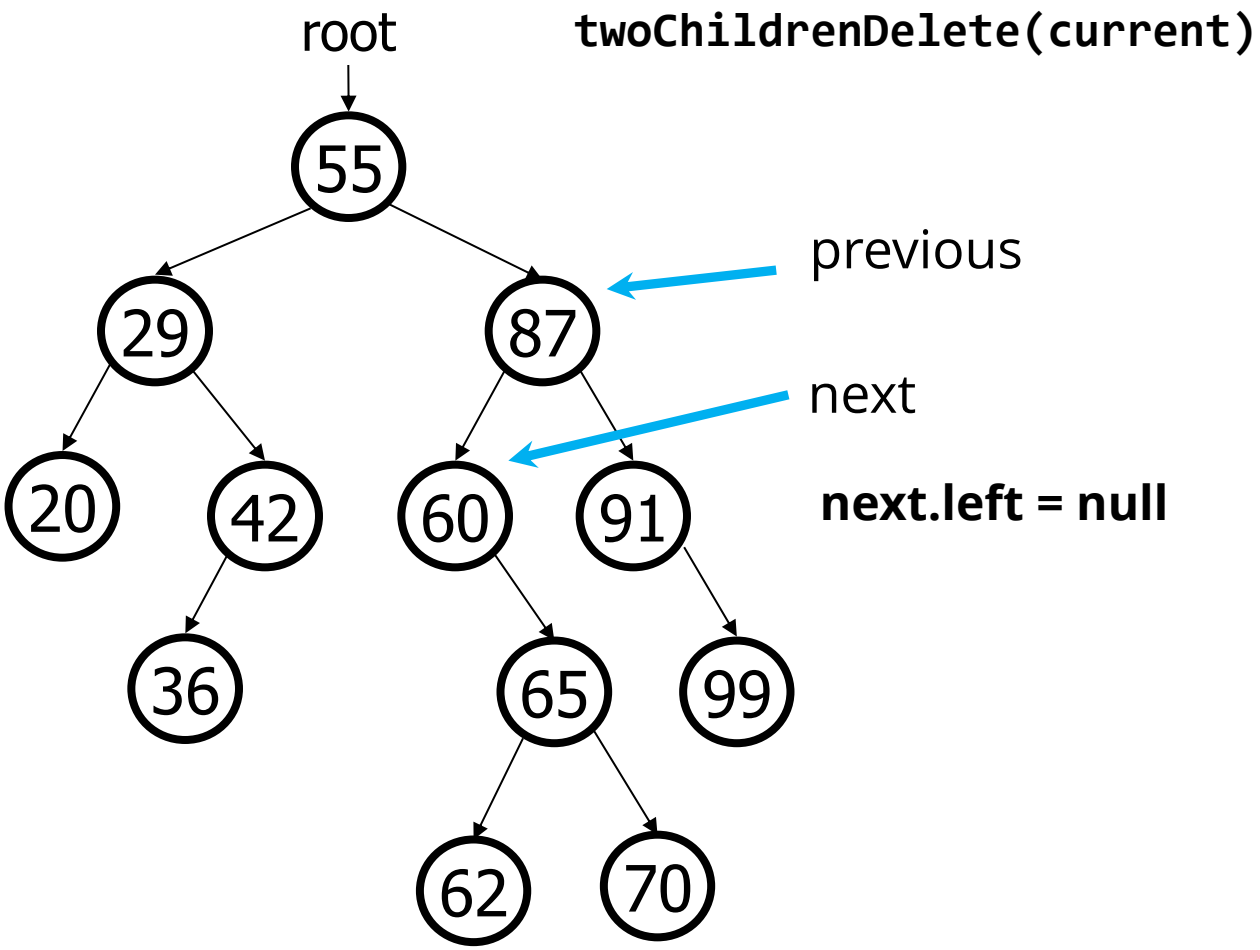
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



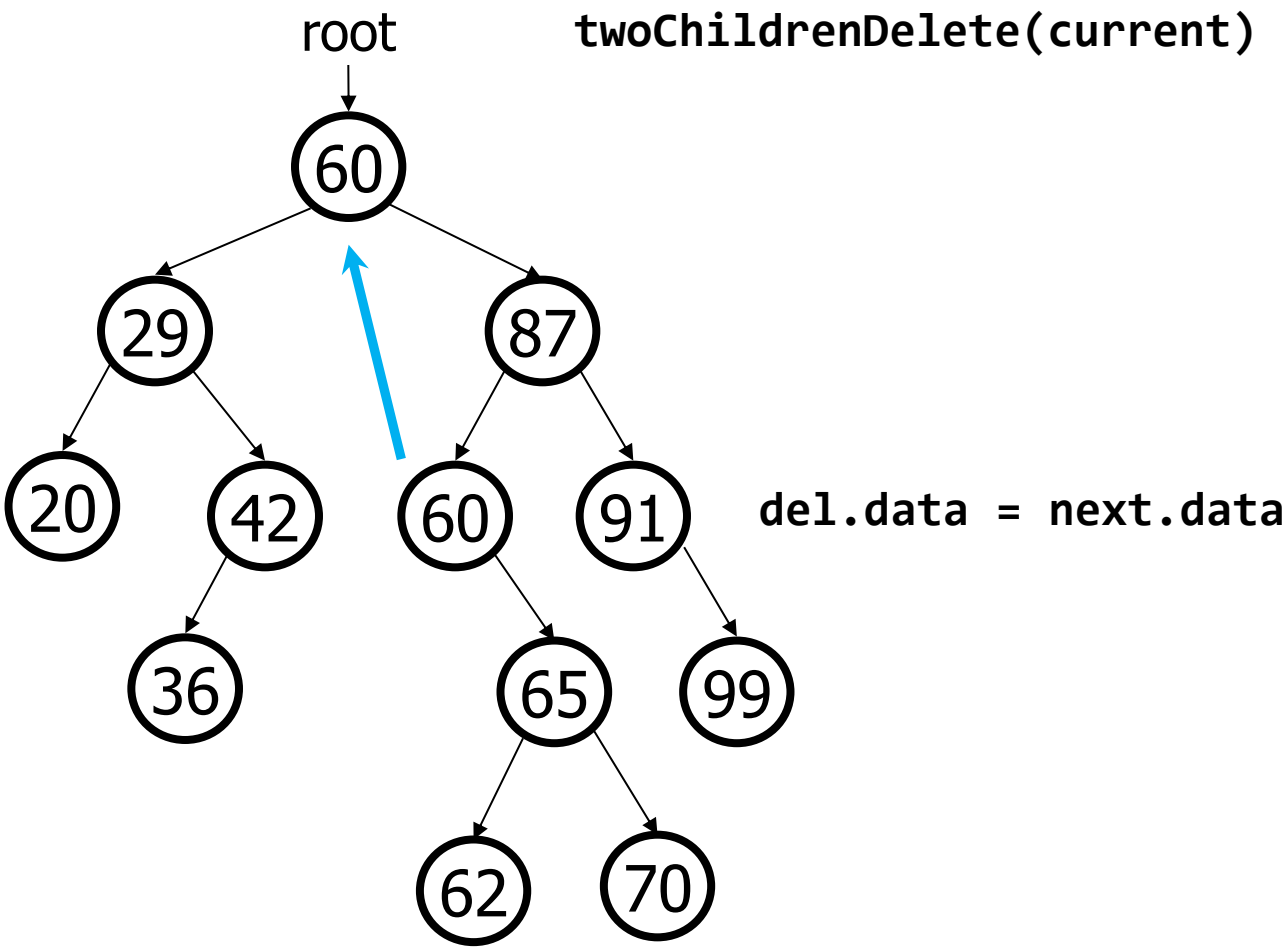
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



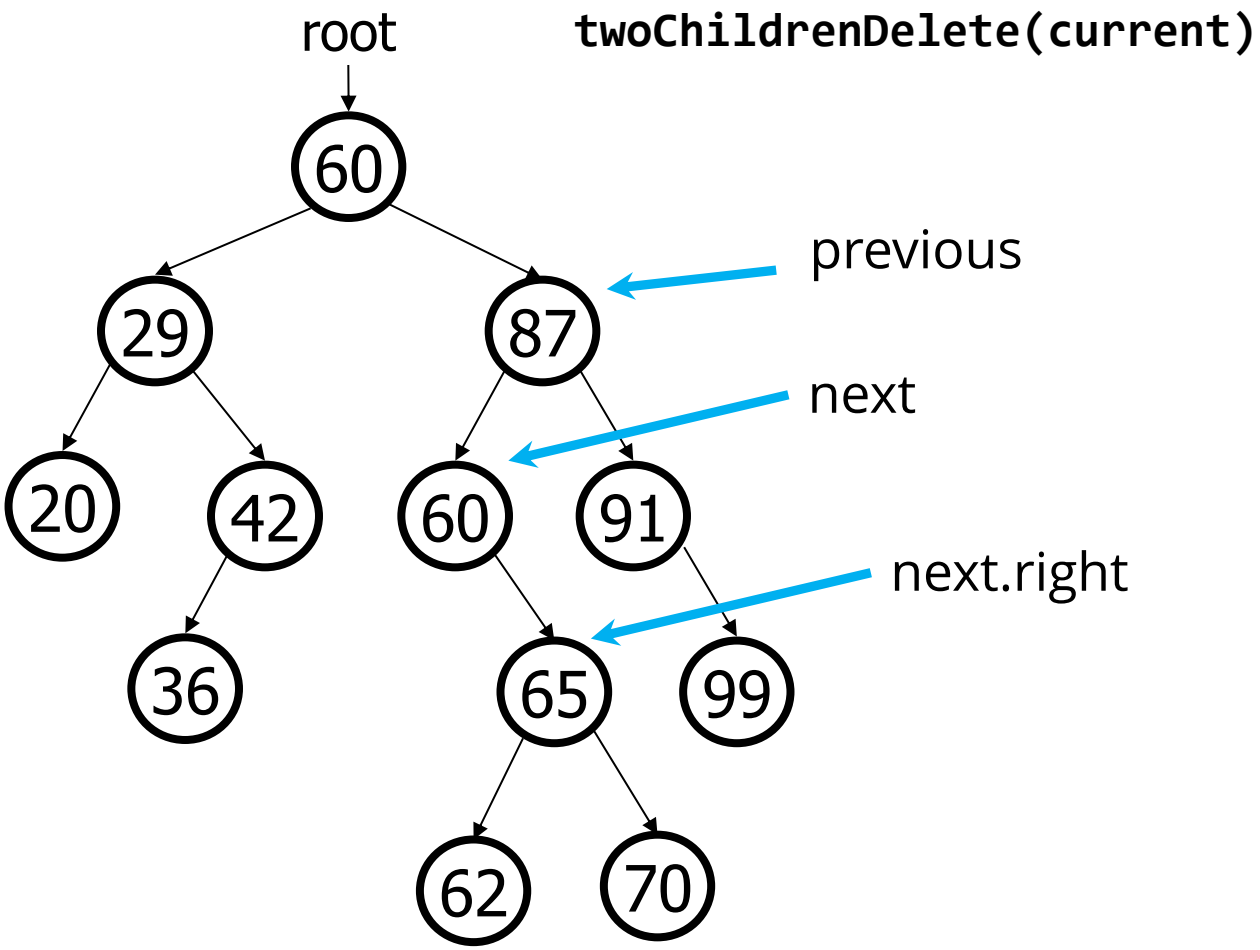
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



Deletion Case 3: Deleting a Two-Child Node

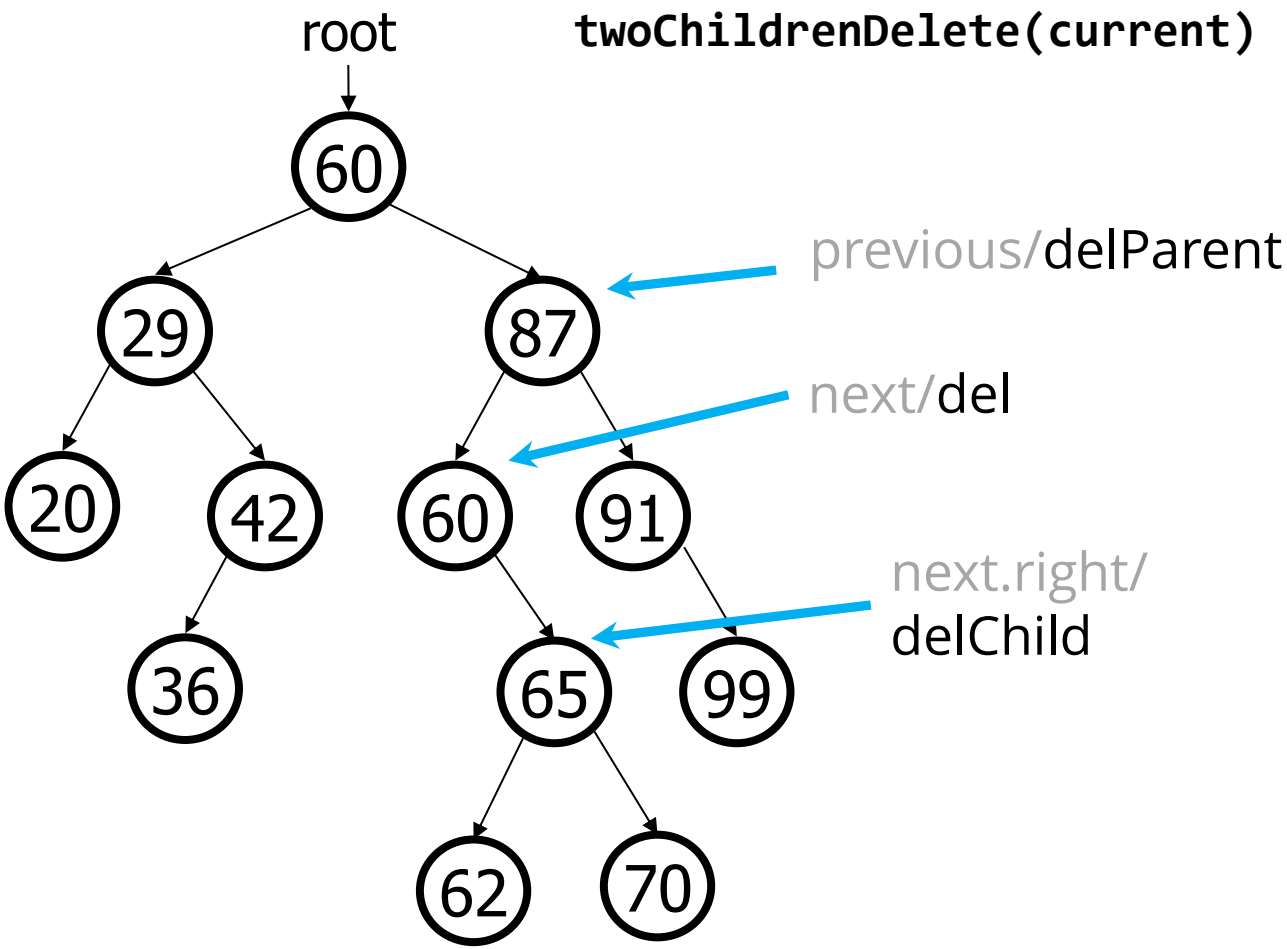
- Example: Delete 55 in the following tree:




```
easyDelete(next,previous,next.right);
```

Deletion Case 3: Deleting a Two-Child Node

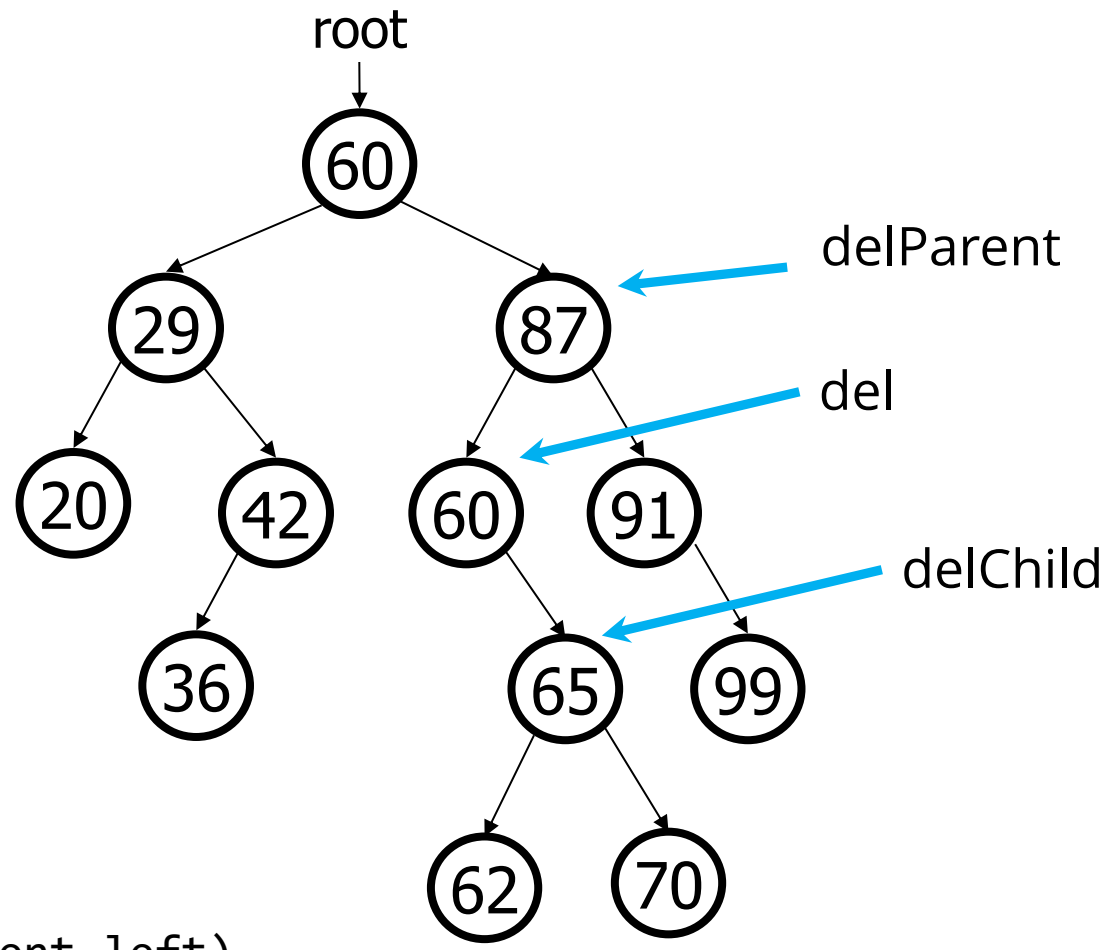
- Example: Delete 55 in the following tree:




```
easyDelete(Node del, Node delParent, Node delChild)
easyDelete(next,previous,next.right);
```

Deletion Case 3: Deleting a Two-Child Node

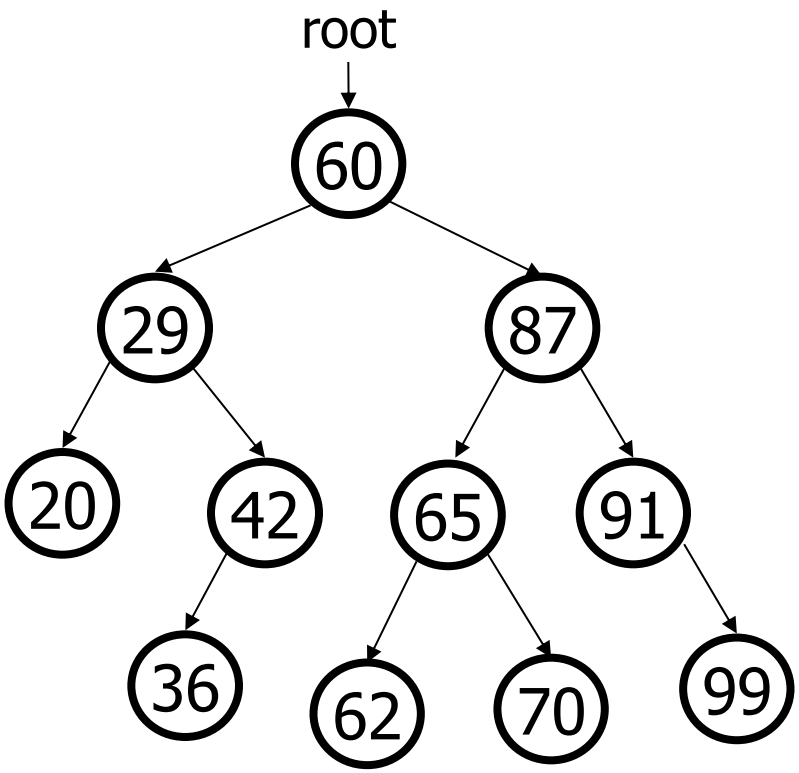
- Example: Delete 55 in the following tree:




```
if (del == delParent.left)
        delParent.left = delChild;
```

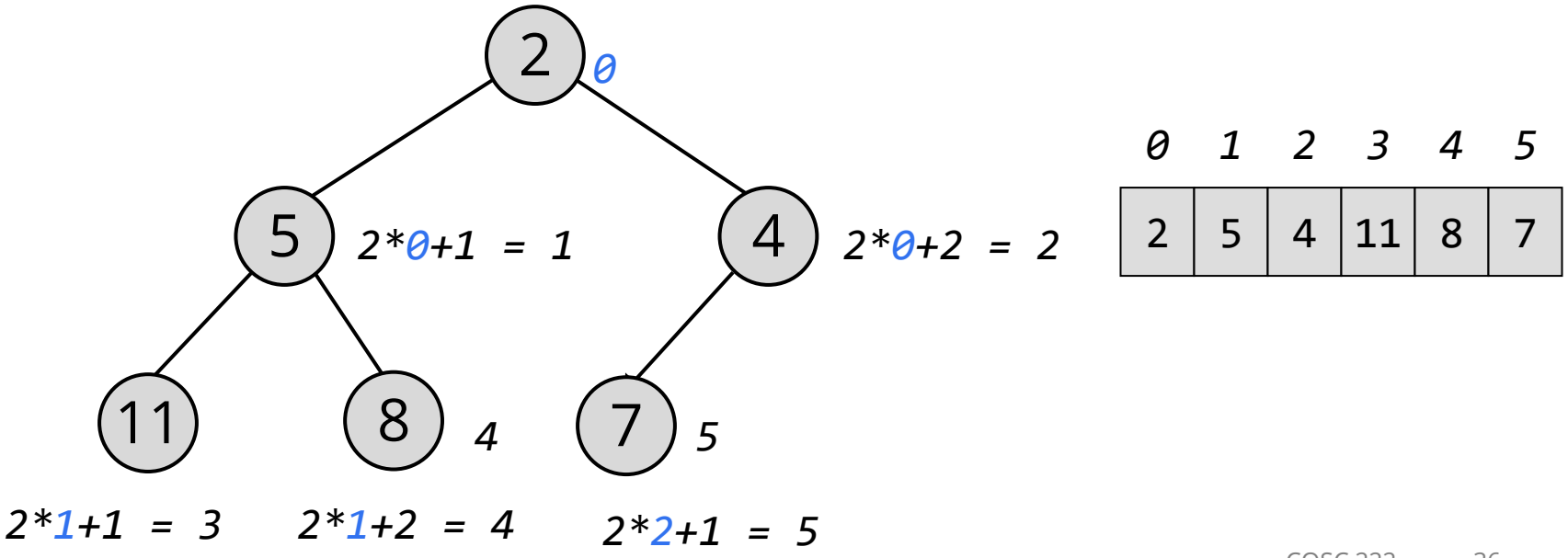
Deletion Case 3: Deleting a Two-Child Node

- Example: Delete 55 in the following tree:



Another Way to Store Trees

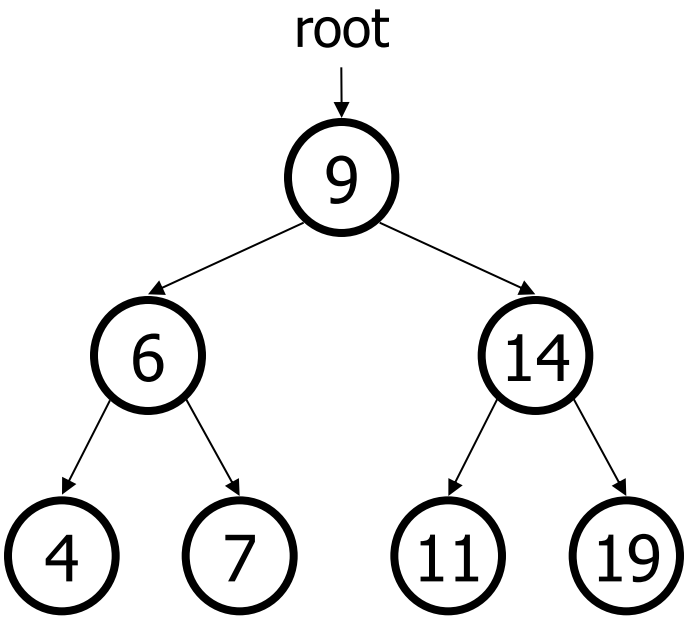
- We've stored binary trees by creating TreeNodes who each have a left and right pointer and forming the tree that way.
- We can also store a binary tree using an array.
  - The root is at index 0
  - Children of node at index i are at indices 2i+1, 2i+2



Searching in BSTs

- In a binary search tree, how many of the items do we eliminate at each step?

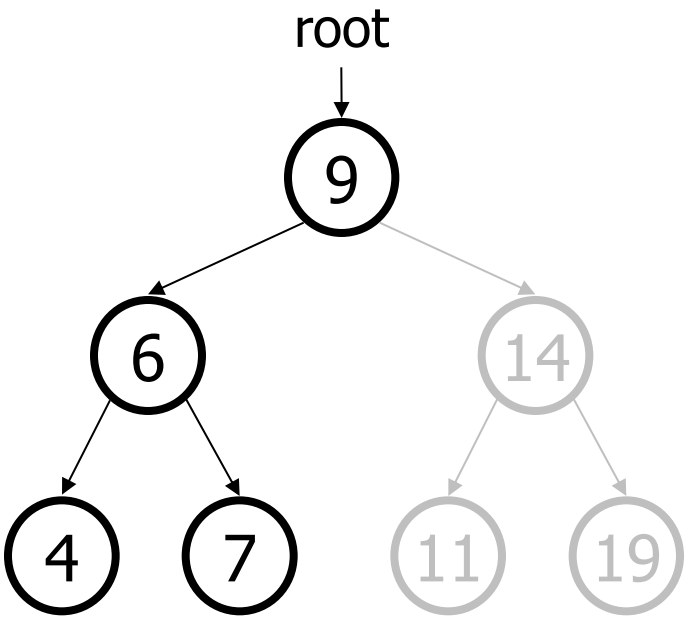
- **Searching for 7**



Searching in BSTs

- In a binary search tree, how many of the items do we eliminate at each step?

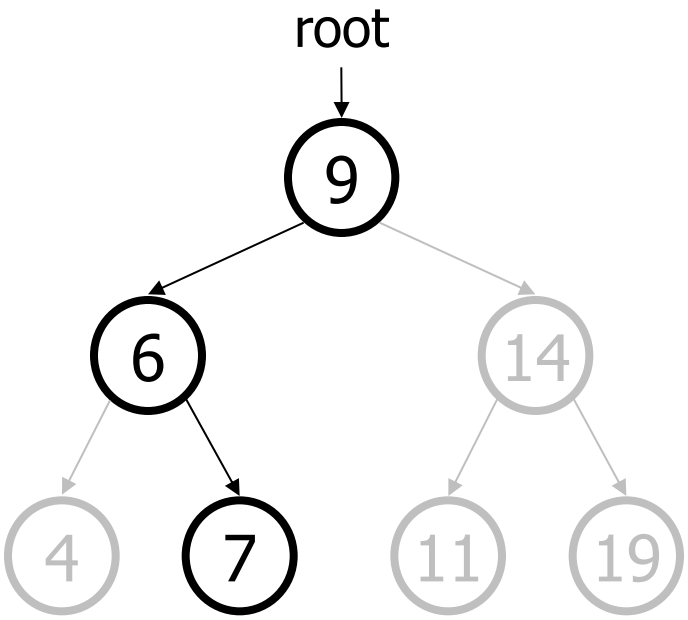
- **Searching for 7**



Searching in BSTs

- In a binary search tree, how many of the items do we eliminate at each step?

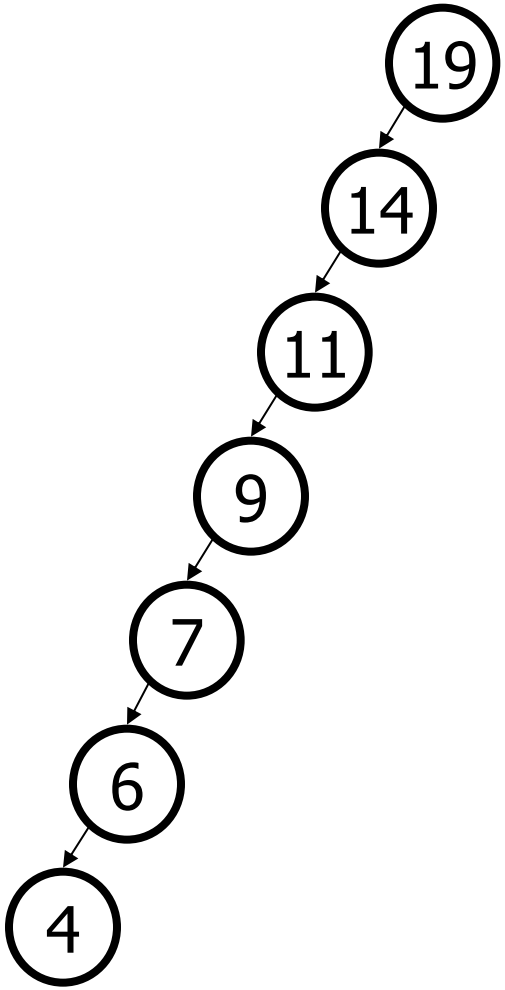
- **Searching for 7**
- Half the tree is eliminated at each step.



Searching in BSTs

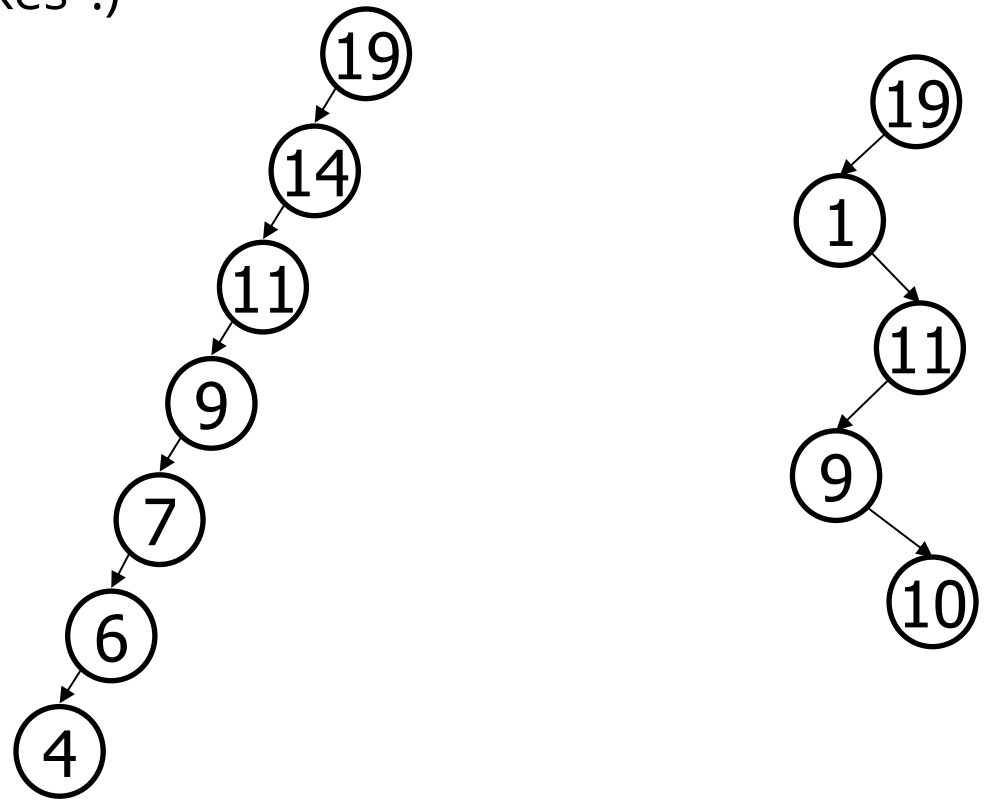
- In a binary search tree, how many of the items do we eliminate at each step?

- **Searching for 6**
- We eliminate almost nothing at each step.



Worst Tree Shapes for Searching

- At worst, we would have to examine every node in the tree, so the worst-case search time is **O(n)**.
- These **degenerate trees** are essentially linked lists, so we get the running times of linked list operations. (Another name for these trees is “snakes".)



Tree Efficiency

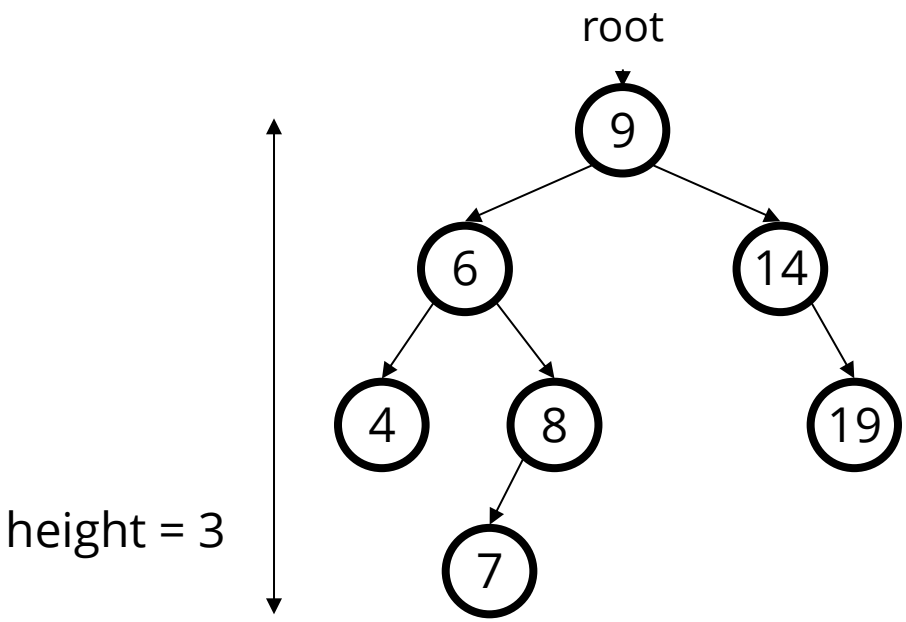
- With 1,000,000 keys:
  - A worst-case search in a snakey linked-list tree takes **1,000,000**
  - **steps** and
  - A worst-case search in a complete tree takes around **20 steps**.

- That's a big difference

- **There's a big pay-off for keeping the tree shorter.**

A good tree

- Balanced tree: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - The runtime of adding/searching a BST is related to height
  - A balanced tree of N nodes has a height of ~ $\log_2$ N.



Questions?