Removal

Binary Search Trees 2

BST Removal

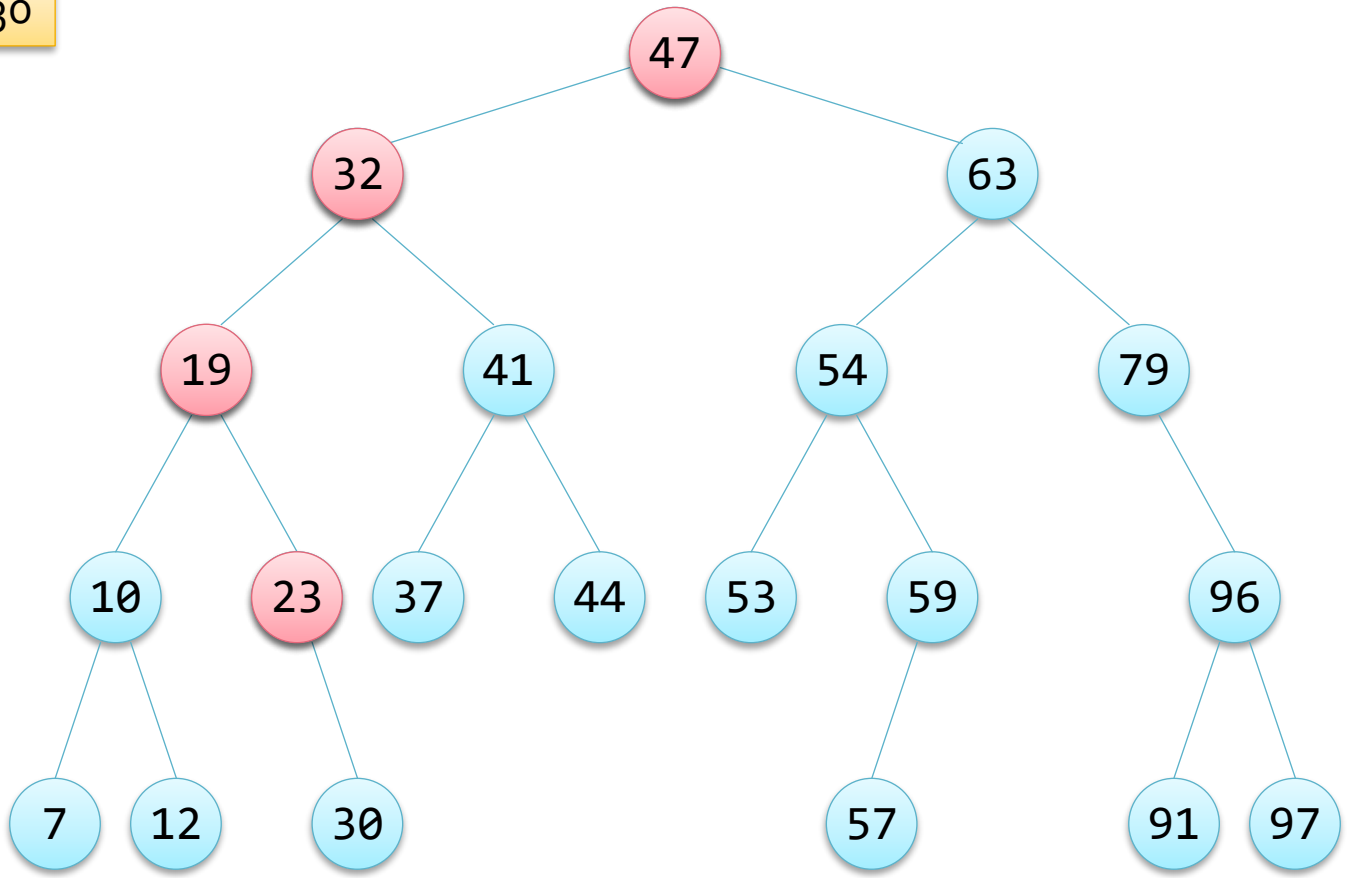
- After removal the BST property must hold
- Removal is not as straightforward as search or insertion
  - With insertion the strategy is to insert a new leaf
  - Which avoids changing the internal structure of the tree
  - This is not possible with removal
    - Since the removed node's position is not chosen by the algorithm
- There are a number of different cases to be considered

BST Removal Cases

- The node to be removed has no children
  - Remove it (assigning NULL to its parent's reference)
- The node to be removed has one child
  - Replace the node with its subtree
- The node to be removed has two children
  - …

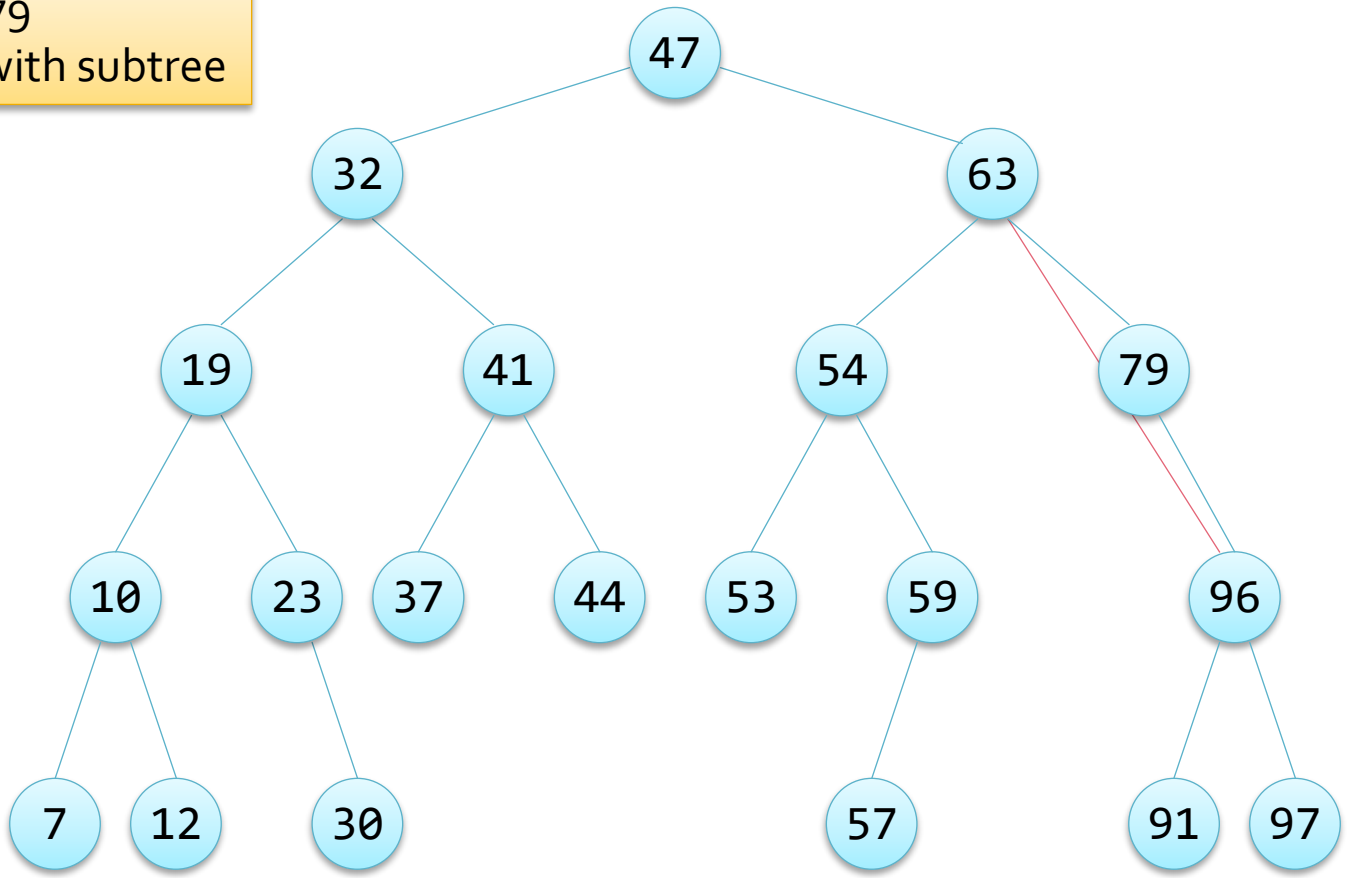
BST Removal – Target is a Leaf

remove 30



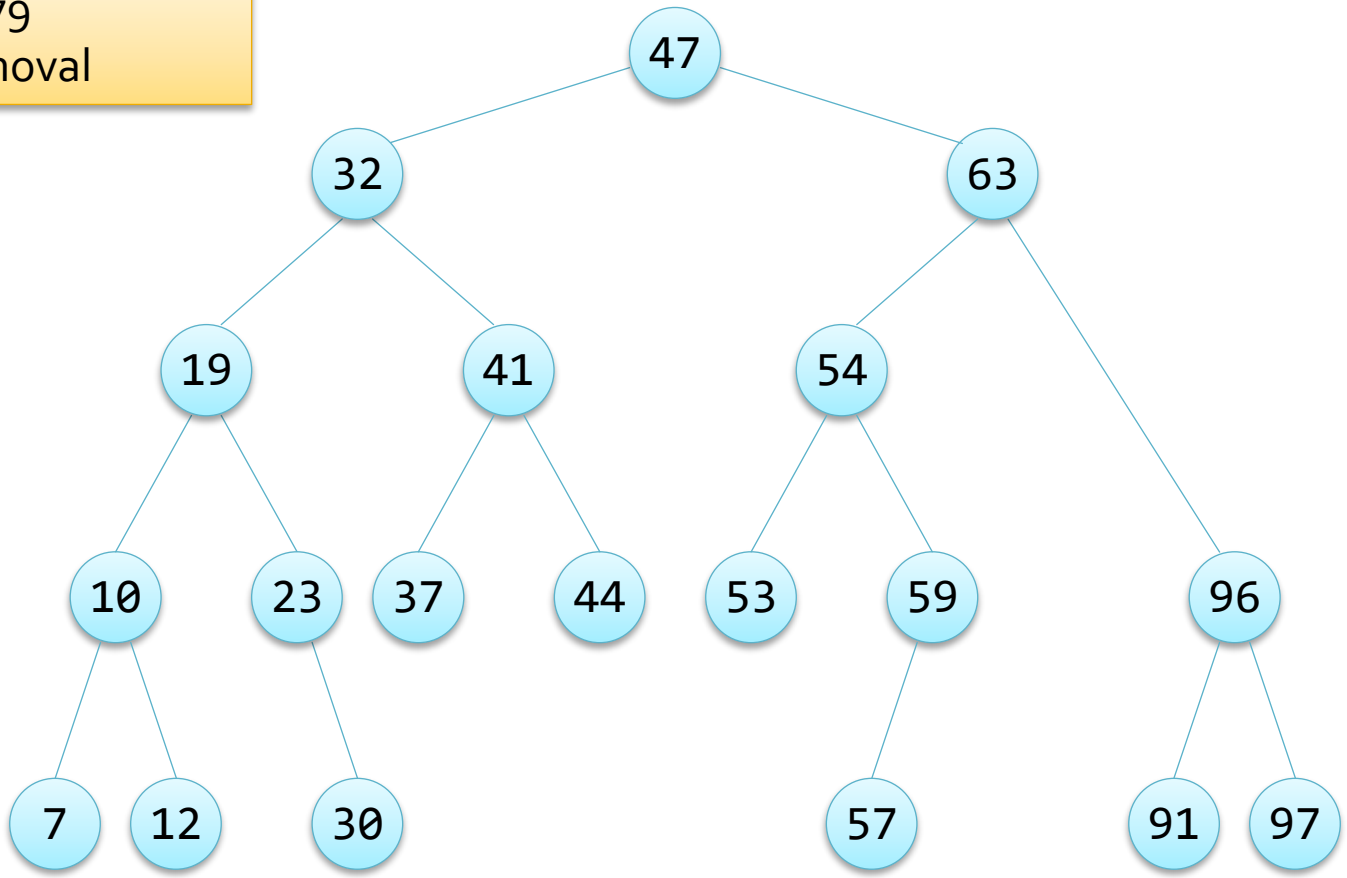
BST Removal – Target Has One Child

remove 79
replace with subtree



BST Removal – Target Has One Child

remove 79
after removal



Looking At the Next Node

- One of the issues with implementing a BST is the necessity to look at both children
  - And, just like a linked list, *look ahead* for insertion and removal
  - And check that a node is null before accessing its member variables
- Consider removing a node with one child in more detail

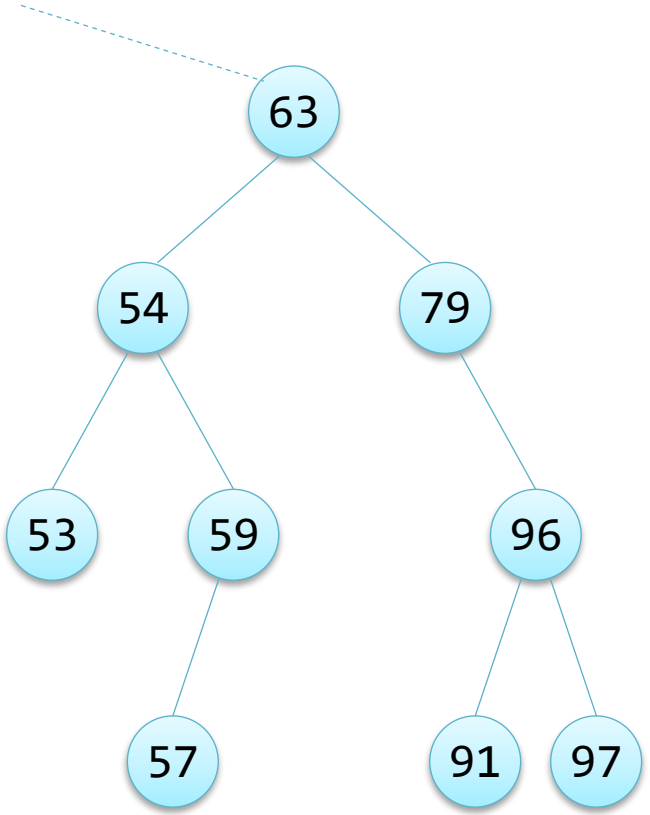
Looking Ahead

remove 59

Step 1 - we need to find the node to remove and its parent

it's useful to know if nd is a left or right child

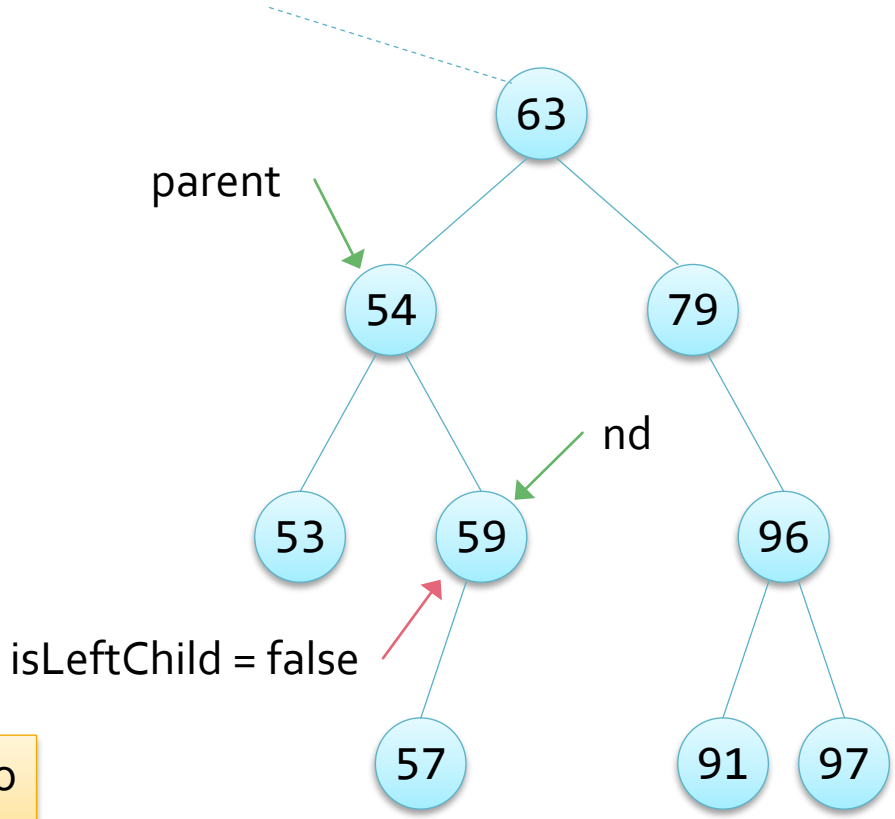
```
while (nd != target)
        if (nd == NULL)
                return
        if (target < nd->data)
                parent = nd
                nd = nd->left
                isLeftChild = true
        else
                parent = nd
                nd = nd->right
                isLeftChild = false
```



Left or Right?

remove 59

```
while (nd != target)
        if (nd == NULL)
                return
        if (target < nd->data)
                parent = nd
                nd = nd->left
                isLeftChild = true
        else
                parent = nd
                nd = nd->right
                isLeftChild = false
```



Now we have enough information to detach 59 and attach its child to 54

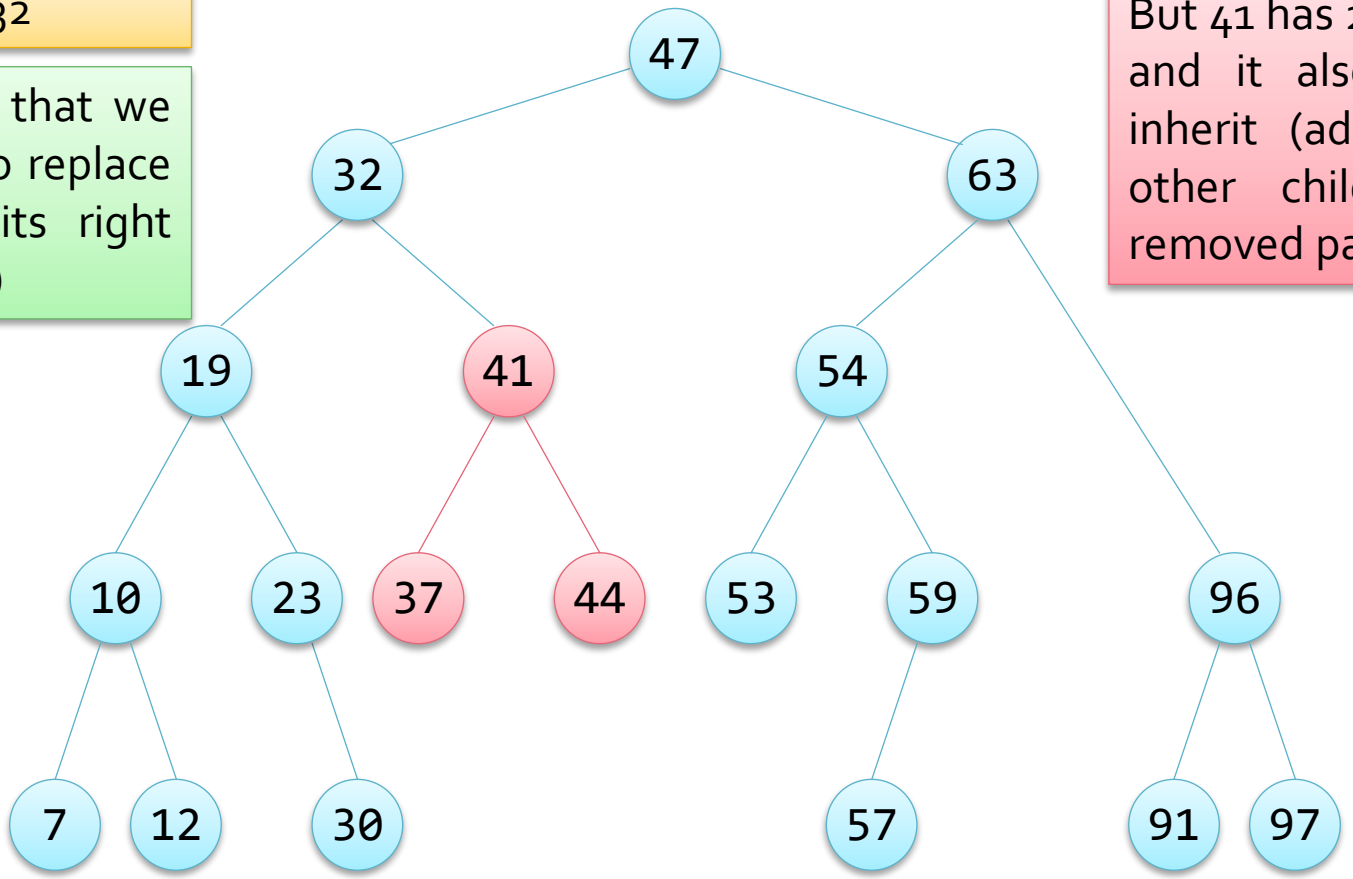
Removing a Node With 2 Children

- The most difficult case is when the node to be removed has two children
  - The strategy when the removed node had one child was to replace it with its child
  - But when the node has two children problems arise
- Which child should we replace the node with?
  - We could solve this by just picking one …
- But what if the node we replace it with also has two children?
  - This will cause a problem

Removed Node Has 2 Children

remove 32

let's say that we decide to replace it with its right child (41)



But 41 has 2 children, and it also has to inherit (adopt?) the other child of its removed parent

Find the Predecessor

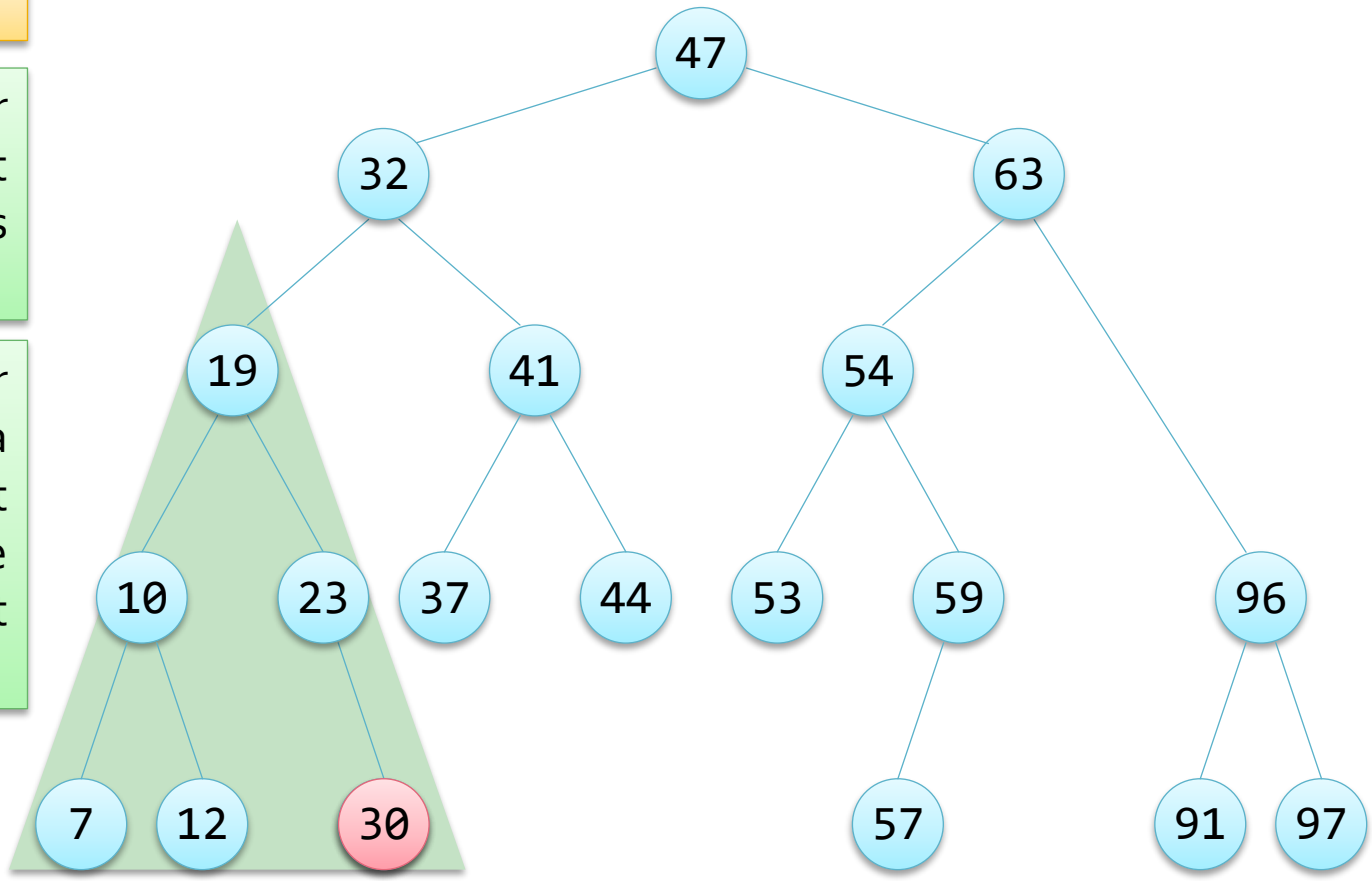
- When a node has two children, instead of replacing it with one of its children find its *predecesor*
  - A node's predecessor is the *right most* node of its *left subtree*
  - The predecessor is the node in the tree with the *largest* value *less* than the node's value
- The predecesor cannot have a right child and can therefore have at most one child
  - Why?

Predecessor Node

32's predecessor

the predecessor of 32 is the right most node in its left subtree

The predecessor *cannot* have a right child as it wouldn't then be the right most node



Why Use the Predecessor?

- The predecessor has some useful properties
  - Because of the BST property it must be the largest value less than its ancestor's value
    - It is to the right of all of the nodes in its ancestor's *left* subtree so must be greater than them
    - It is less than the nodes in its ancestor's *right* subtree
  - It can have only one child
- These properties make it a good candidate to replace its ancestor

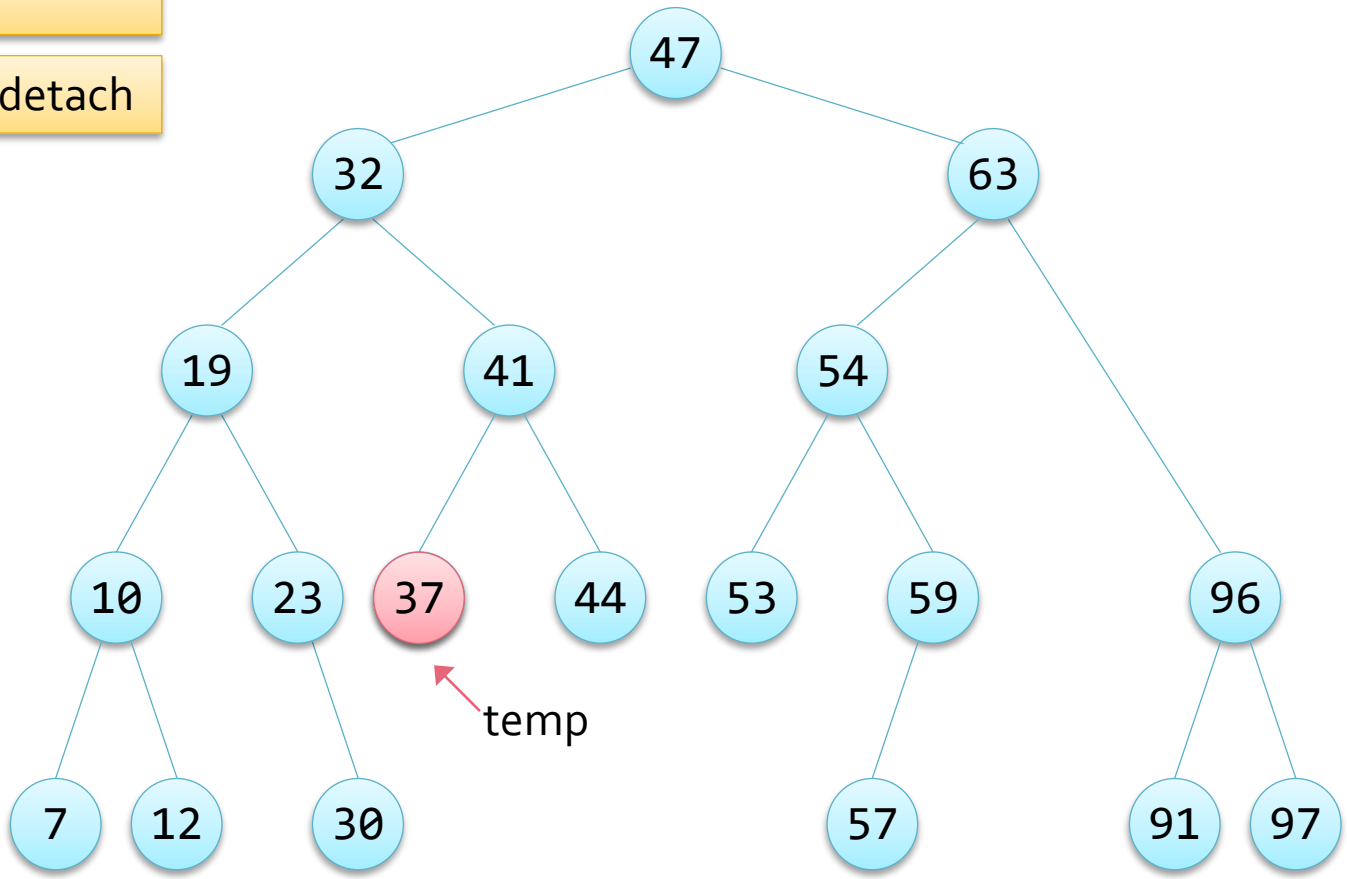
What About the Successor?

- The successor to a node is the *left* most child of its *right* subtree
  - It has the *smallest* value *greater* than its ancestor's value
  - And cannot have a left child
- The successor can also be used to replace a removed node
  - Pick either the precedessor or successor!

Removed Node Has 2 Children

remove 32

find successor and detach

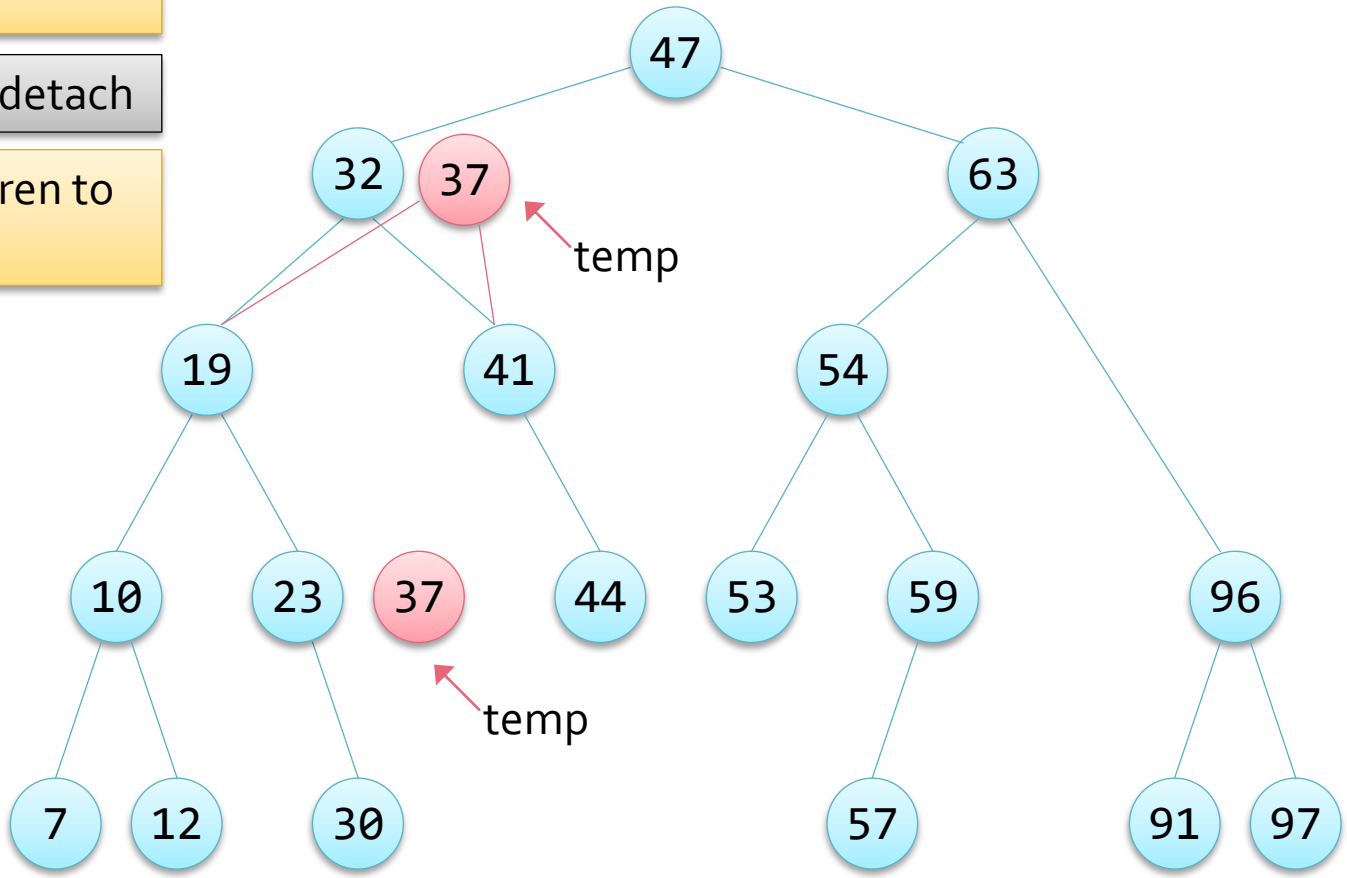


Removed Node Has 2 Children

remove 32

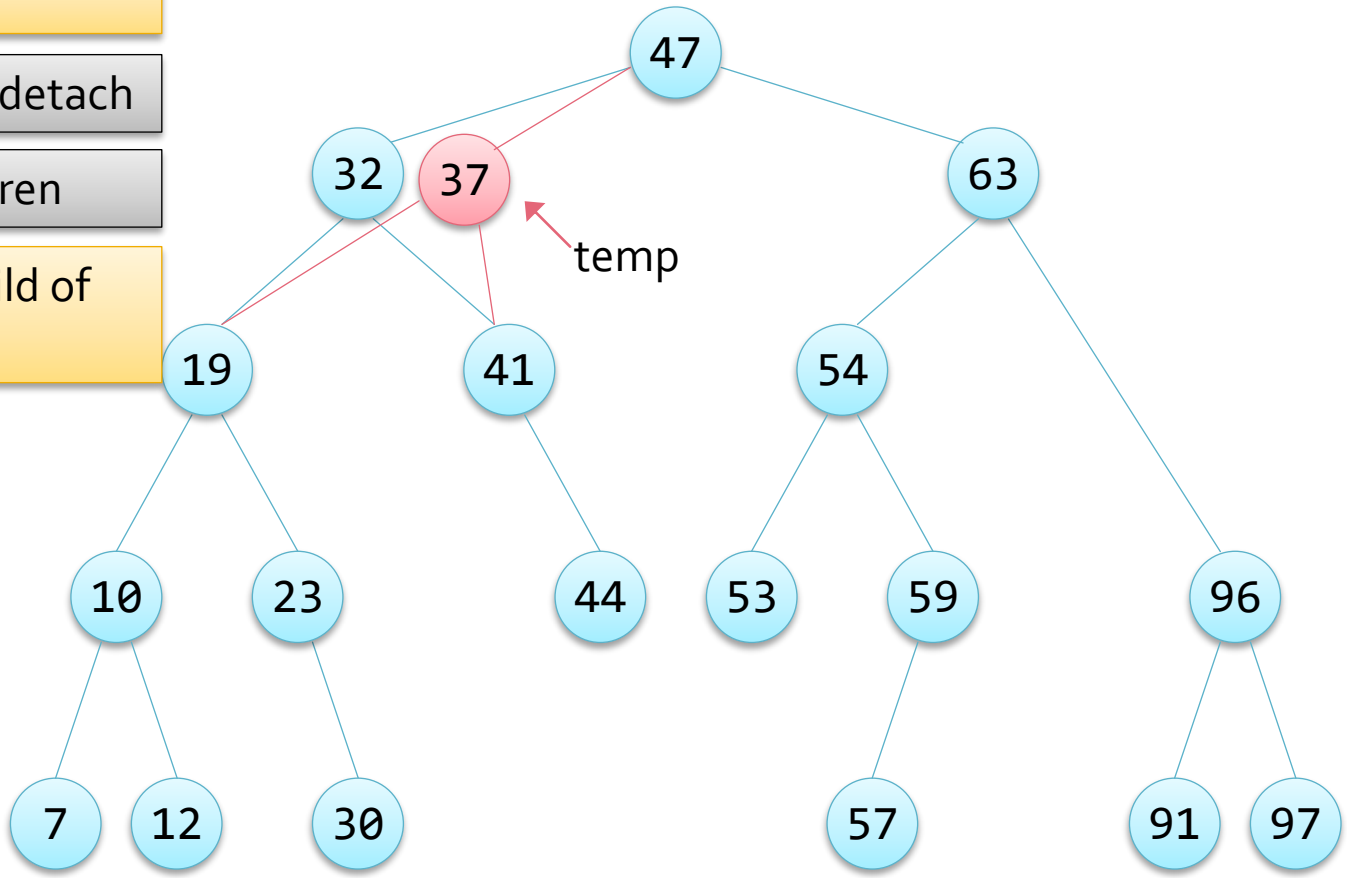
find successor and detach

attach node's children to its successor



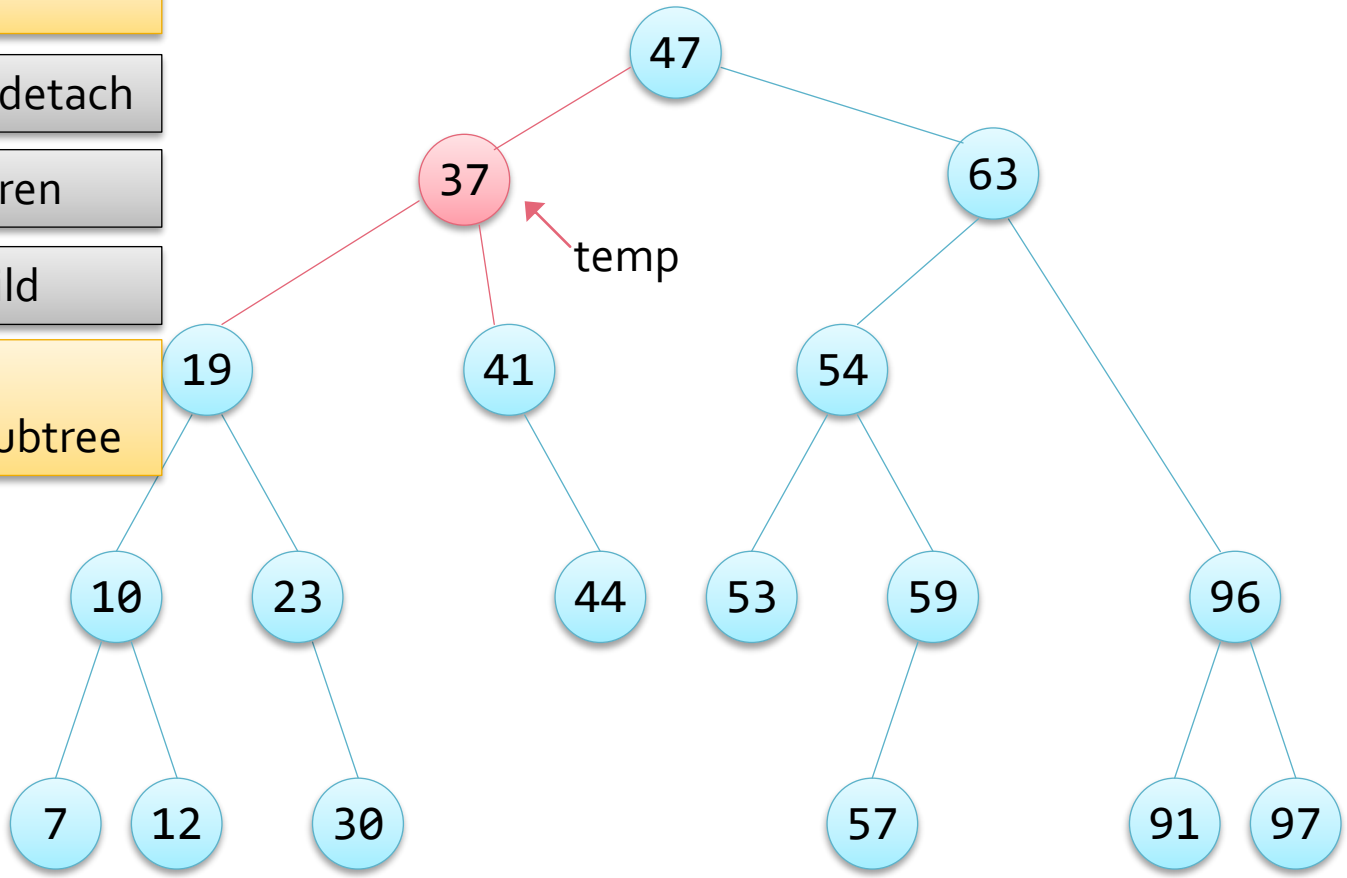
Removed Node Has 2 Children

- remove 32
- find successor and detach
- attach node's children
- make successor child of node's parent



Removed Node Has 2 Children

- remove 32
- find successor and detach
- attach node's children
- make successor child
- in this example the successor had no subtree

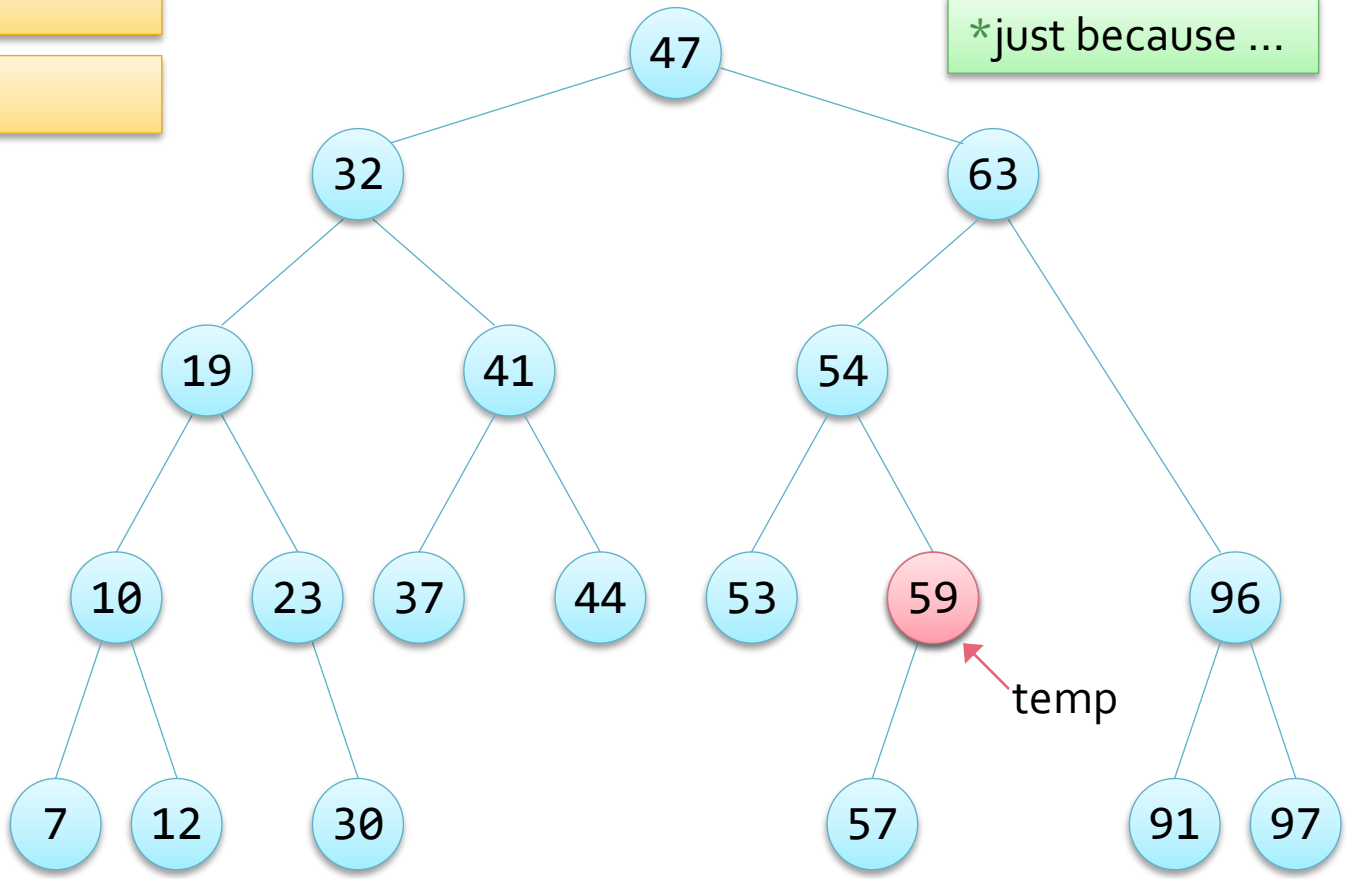


Removed Node Has 2 Children

remove 63

find predecessor*

*just because ...

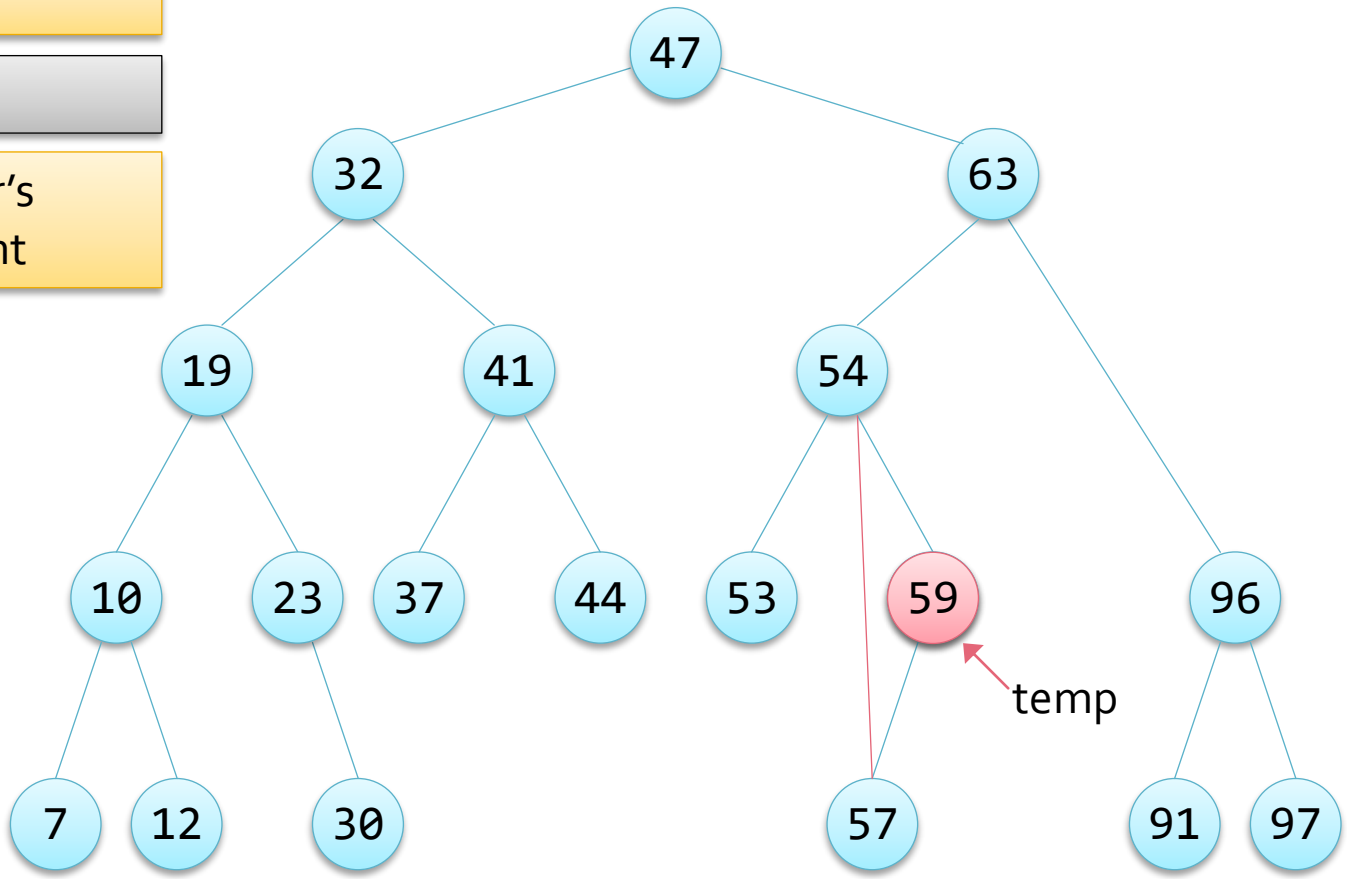


Removed Node Has 2 Children

remove 63

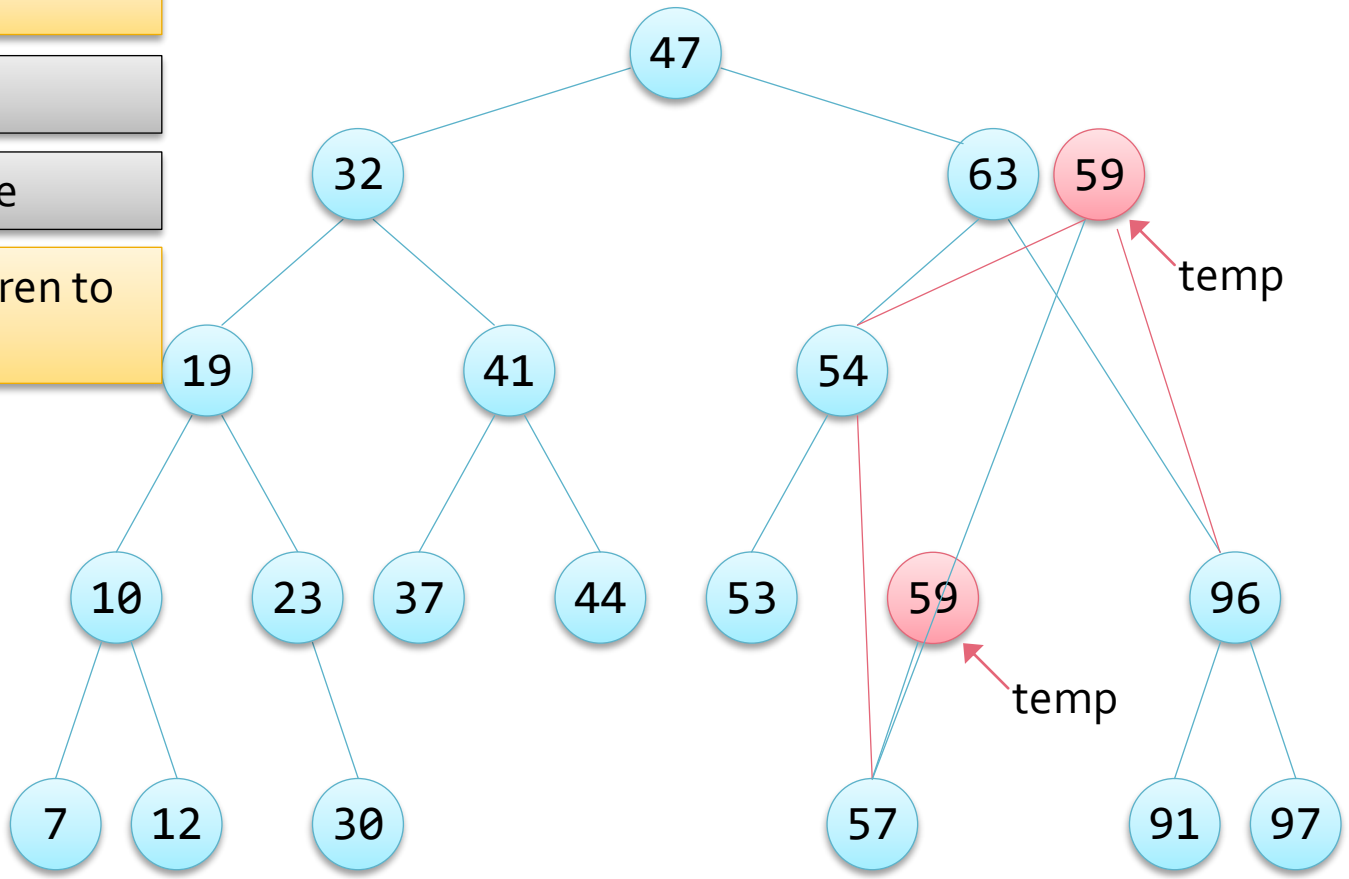
find predecessor

attach predecessor's subtree to its parent



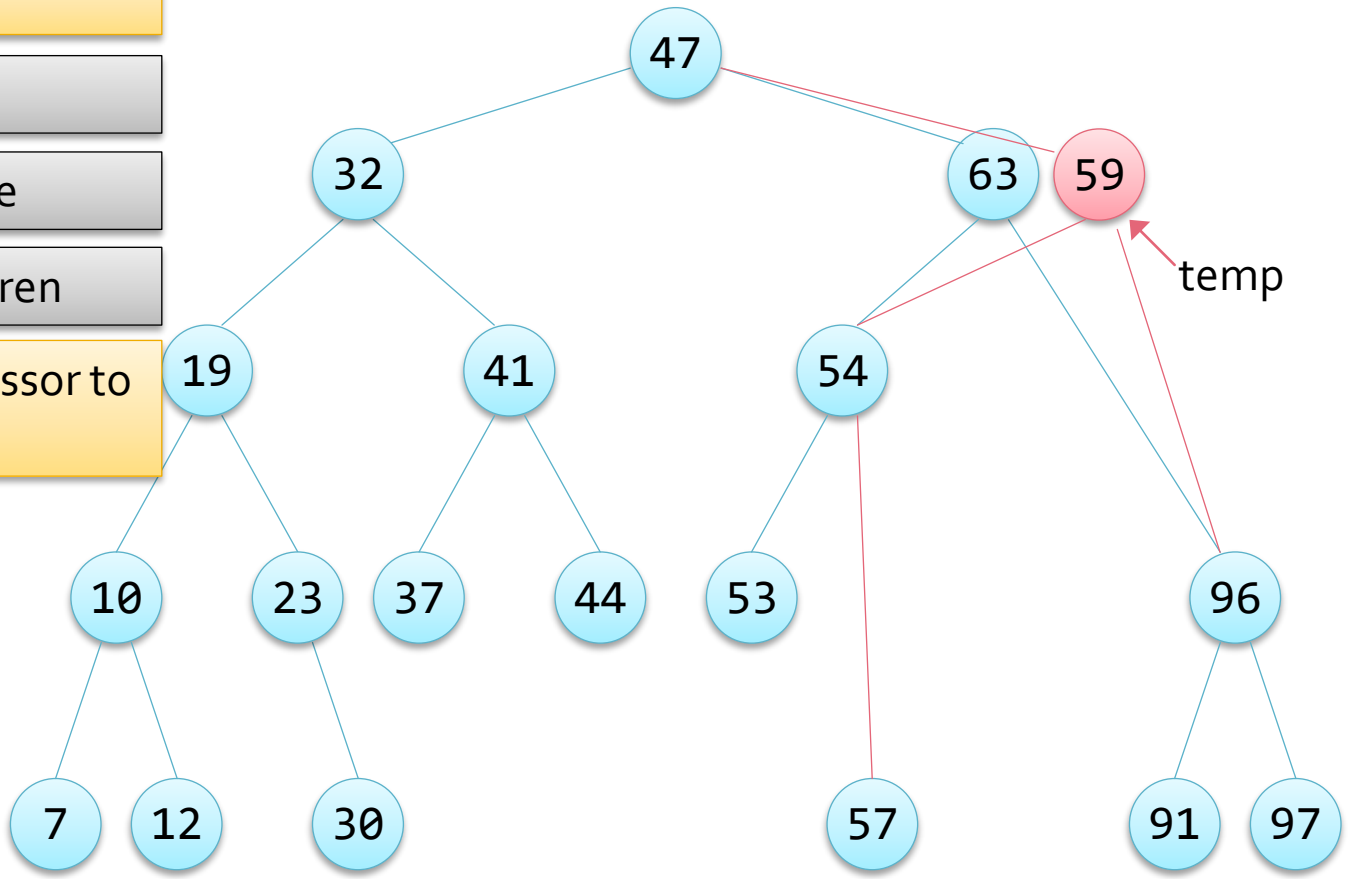
Removed Node Has 2 Children

- remove 63
- find predecessor
- attach pre's subtree
- attach node's children to predecessor



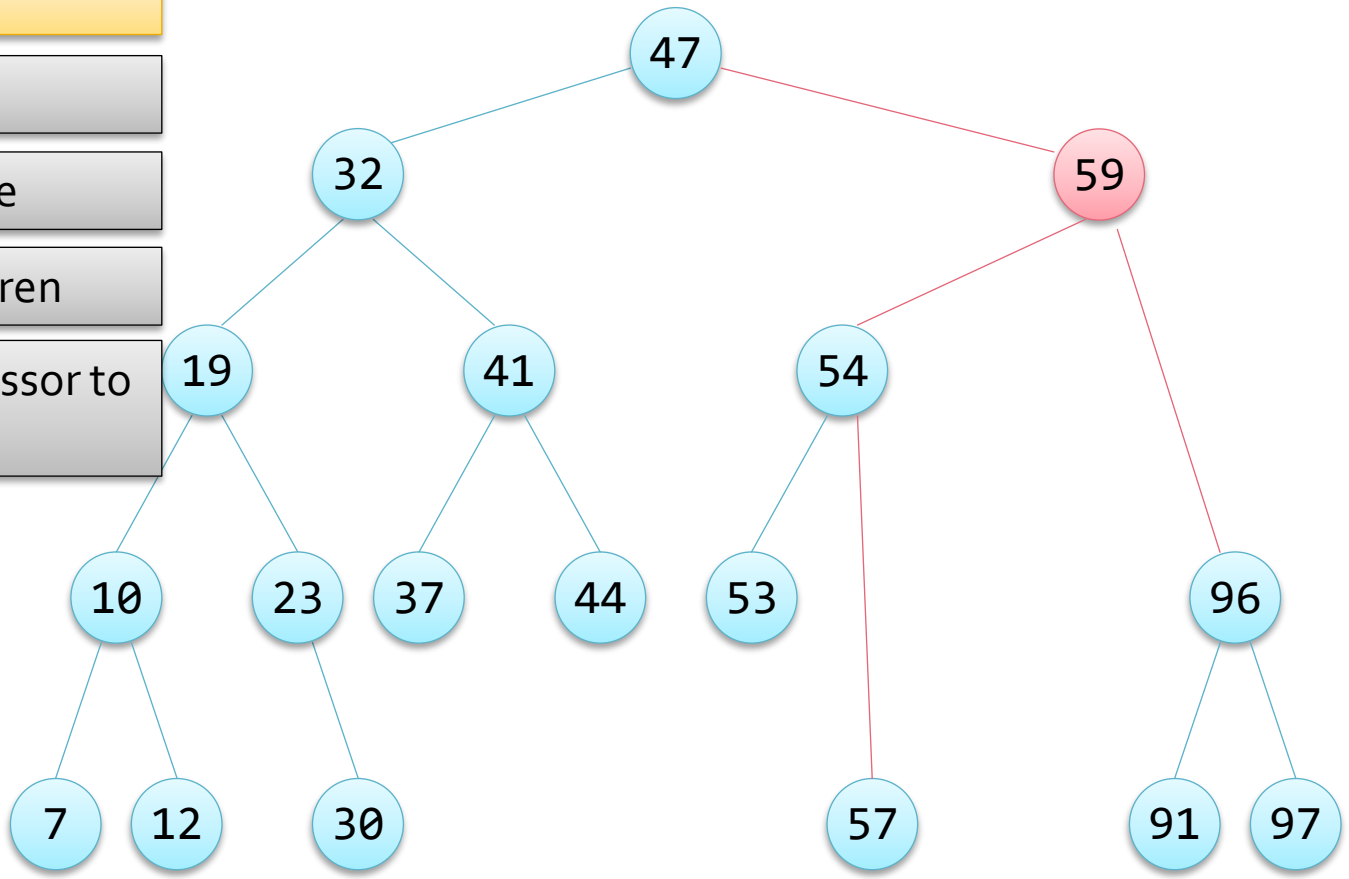
Removed Node Has 2 Children

- remove 63
- find predecessor
- attach pre's subtree
- attach node's children
- attach the predecessor to the node's parent



Removed Node Has 2 Children

- remove 63
- find predecessor
- attach pre's subtree
- attach node's children
- attach the predecessor to the node's parent



Removal Alternatives - 1

- Instead of removing a BST node mark it as removed in some way
  - Set the data object to *null*, for example
- And change the insertion algorithm to look for empty nodes
  - And insert the new item in an empty node that is found on the way down the tree

Removal Alternatives - 2

- An alternative to the removal approach for nodes with 2 children is to replace the data
  - The data from the predecessor node is copied into the node to be removed
  - And the predecessor node is then removed
    - Using the approach described for removing nodes with one or no children
- This avoids some of the complicated pointer assignments