Binary Search Trees Update operations

Lecture 19

by Marina Barsky

BST: update operations

- ***Add*** (k): creates a new node with key k and inserts it into the appropriate position of BST

- ***Remove*** (k): deletes the node with key k in such a way that the BST property of the tree is preserved

We already have all the helper algorithms to implement these

Algorithm ***Add***

Input: Key k

Output: Updated BST containing a new node N with key k

Algorithm *Search* (*k*, *R*)

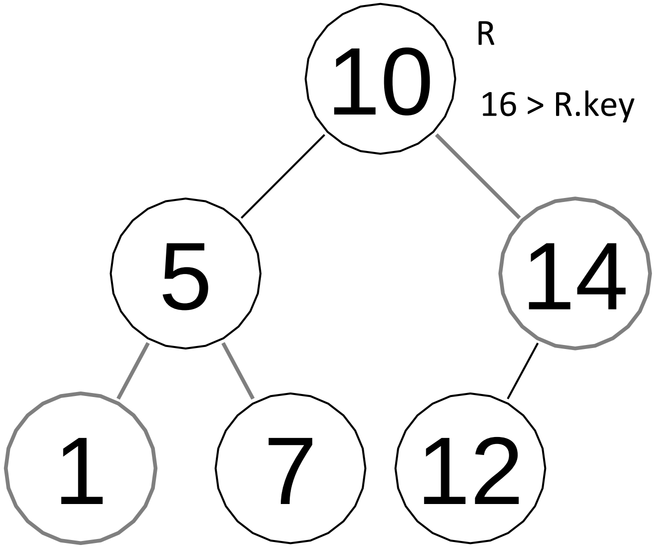
```
if R is Null or R.Key = k:
  return R
if R.Key > k:
  return Search(k, R.Left)
else if R.Key < k:
  return Search(k, R.Right)
```

We need to slightly modify ***Search***

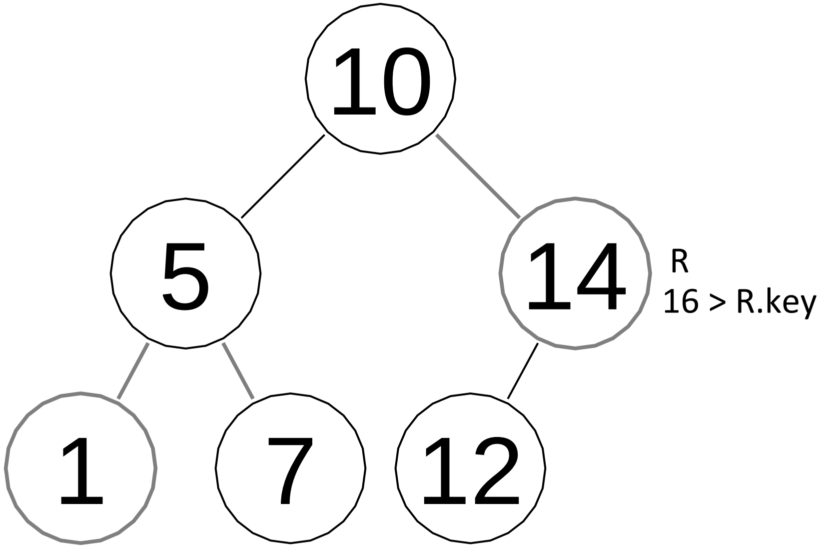
Algorithm ***Add*** (*k*, *R*)

```
if R != Null and R.Key = k:
      ERROR: already in the tree
if k < R.Key:
      if R.left == Null:
            R.left = new Node(k)
      else:
            Add( k, R.left)
if k > R.Key:
      if R.right == Null:
            R.right = new Node(k)
      else:
            Add( k, R.right)
```

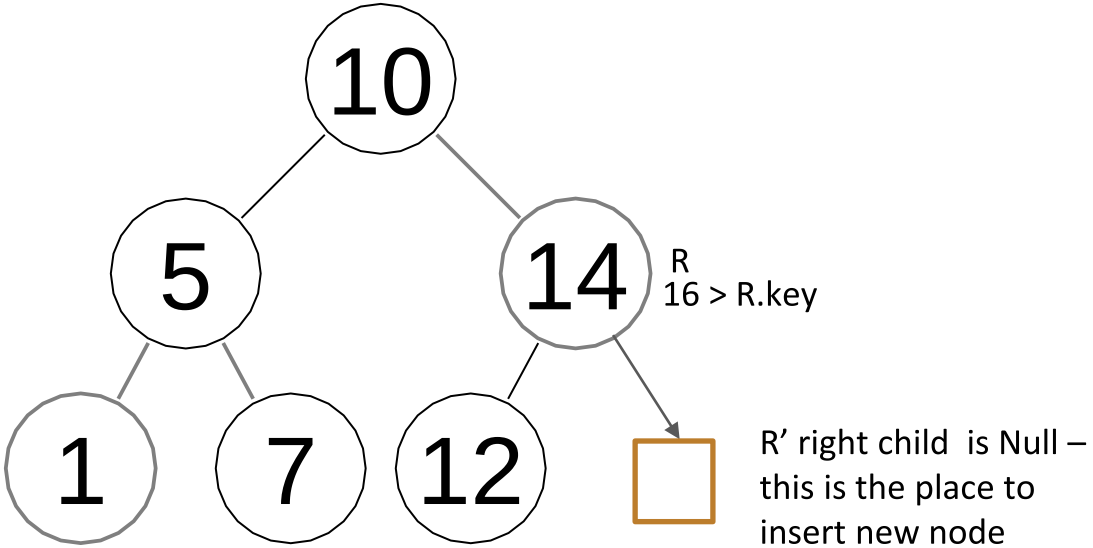
Example: add (16, R)



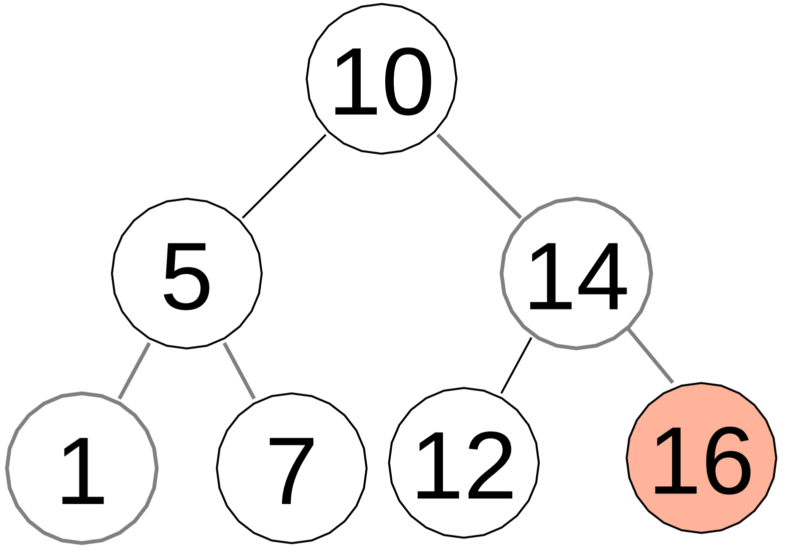
Example: add (16, R)



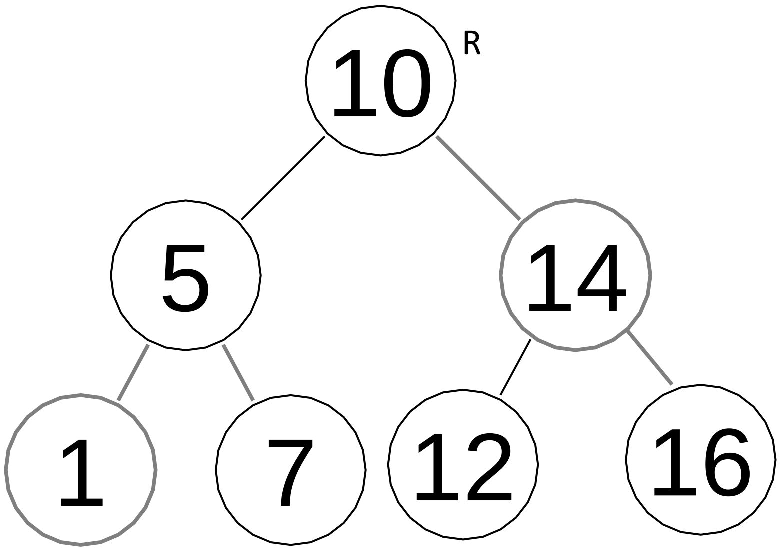
Example: add (16, R)



Example: add (16, R)

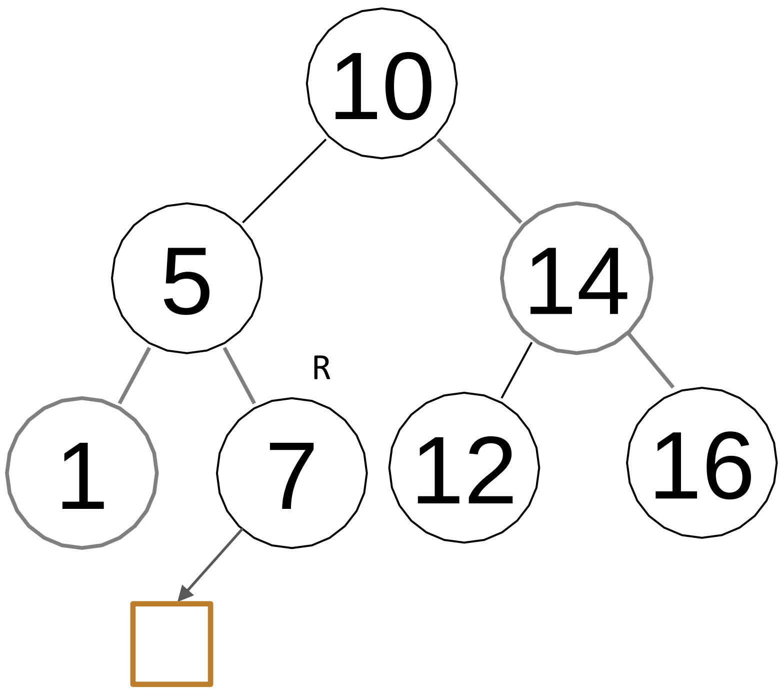


Where will be the new node *N* created after we call add (6, R) on the root R of the following tree?

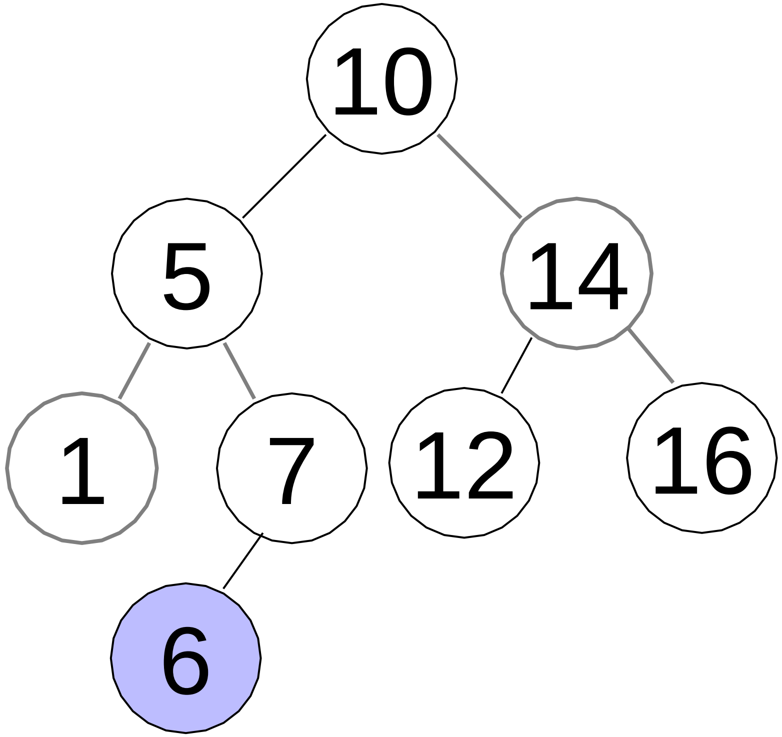


A. *N* will become the right child of 5
B. The left child of 7
C. The right child of 1
D. None of the above (somewhere else)

Solution: add (6, R)



Solution: add (6, R)



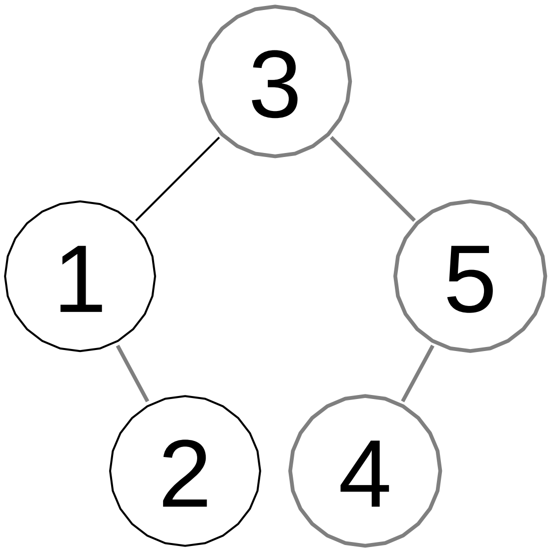
Algorithm Remove

Input: Key k

Output: BST without node N *with key* k

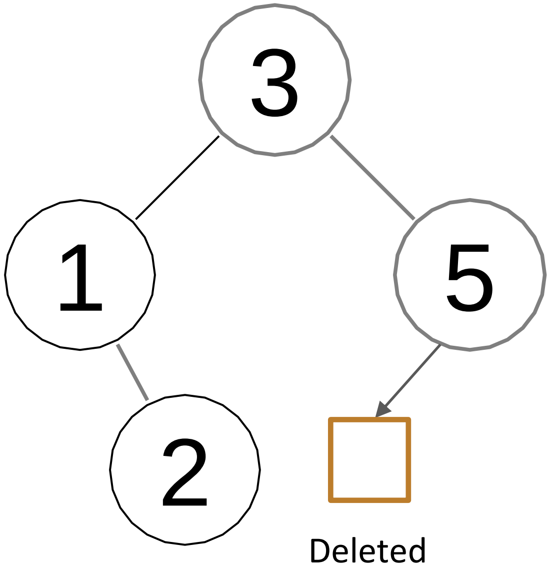
The most challenging algorithm in this module

Remove node *N* with key *k*



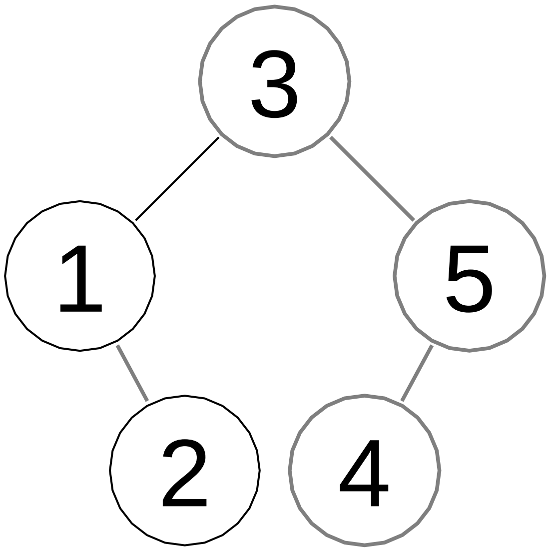
- First, find *N*
- Easy case (both *N*'s children are nulls)
  - Replace *N* with a Null Node

Remove node *N* with key *k*



- ➢First, find *N*
- ➢Easy case (both *N*'s children are nulls)
  - Replace *N* with Null Node

Remove node *N* with key *k*

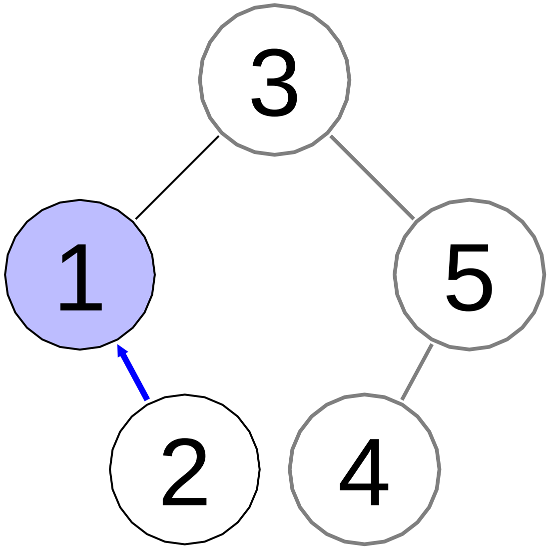


➢Medium case (N has one real child):

Just “splice out” node *N*

- Its unique real child assumes the position previously occupied by *N* – gets ***promoted*** to its place

Example: *remove*(1)

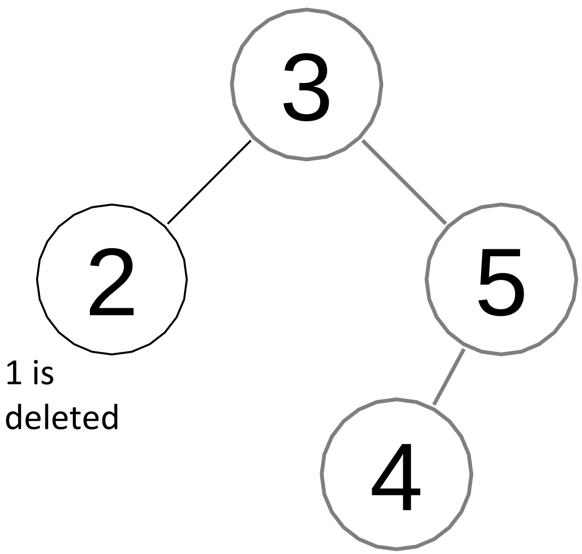


➢ Medium case (N has one real child):

Just “splice out” node *N*

- Its unique real child assumes the position previously occupied by *N* – gets ***promoted*** to its place

Example: *remove*(1)

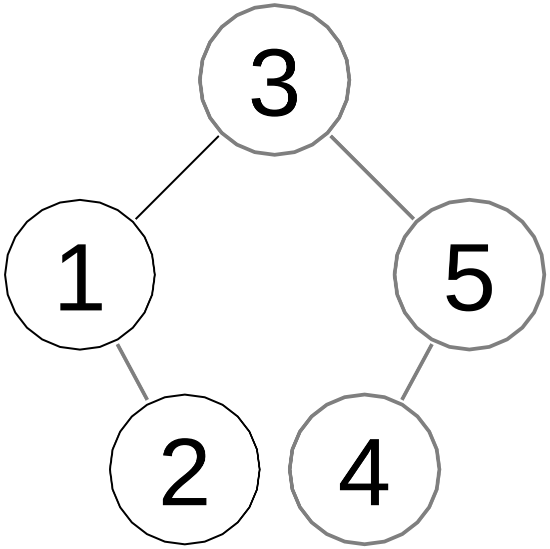


- Medium case (N has one real child):

  Just “splice out” node *N*

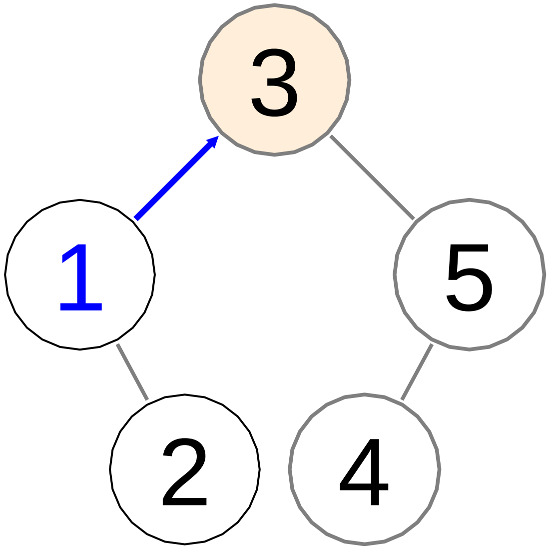
  - Its unique real child assumes the position previously occupied by *N* – gets ***promoted*** to its place

Remove node *N* with key *k*



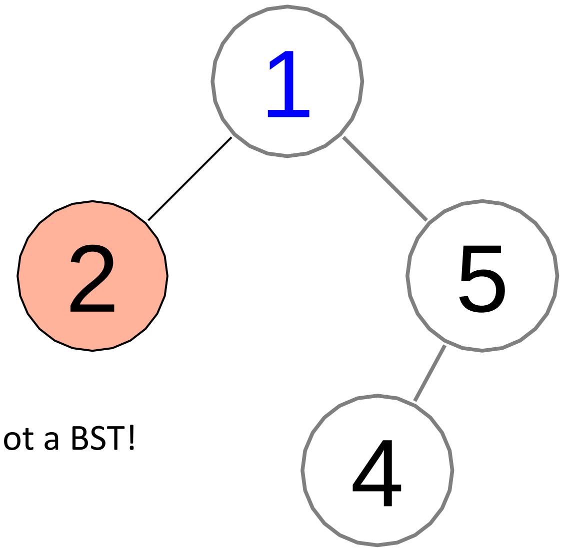
➢ Difficult case (both N's children are real nodes):

Example: *remove*(3)



- Difficult case (both N's children are real nodes):
  - Promote 1?

Example: *remove*(3)

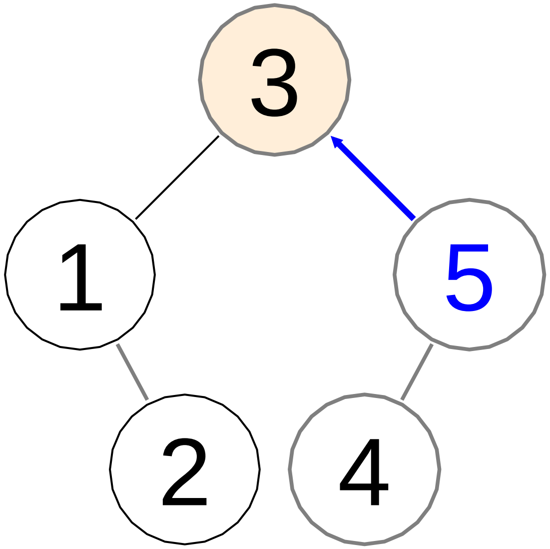


Not a BST!

➢ Difficult case (both N's children are real nodes):

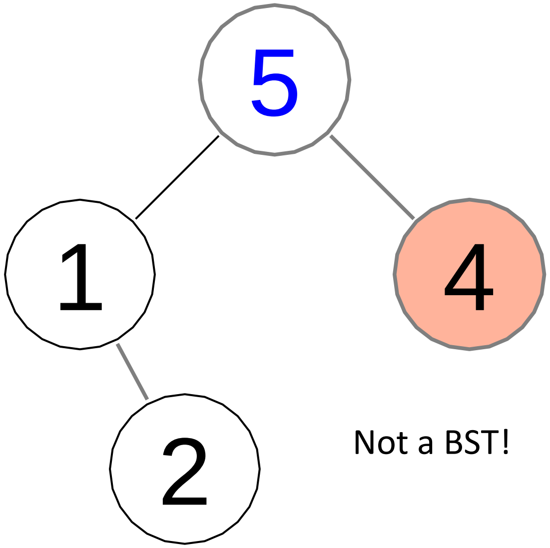
- Promote 1?

Example: *remove*(3)



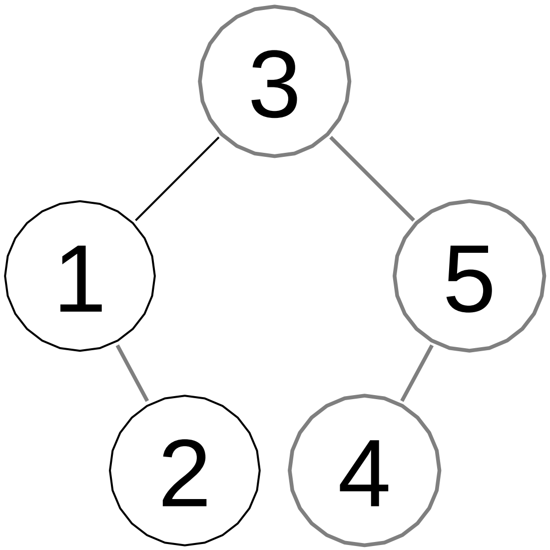
- Difficult case (both N's children are real nodes):
  - Promote 5?

Example: *remove*(3)



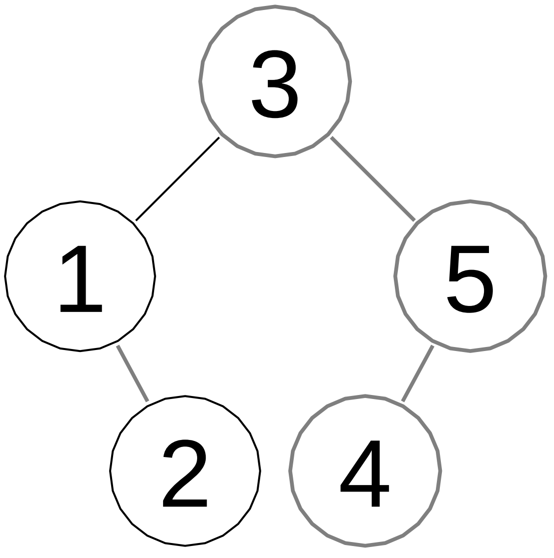
- Difficult case (both N’s children are real nodes):
  - Promote 5?

Remove node *N* with key *k:* difficult case



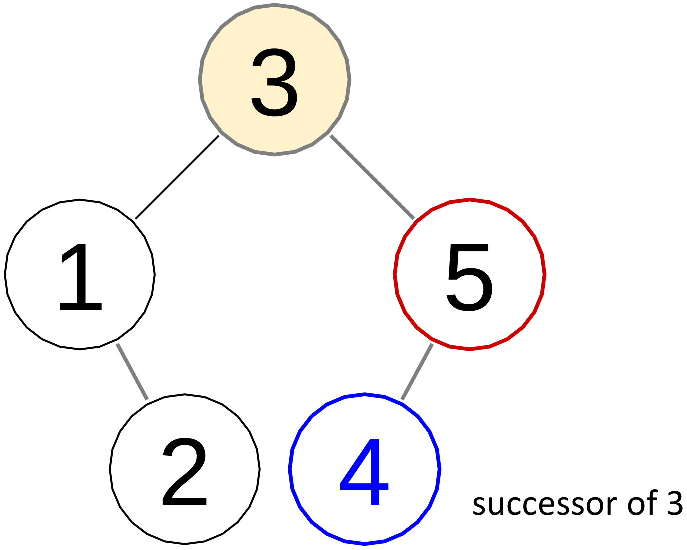
- Difficult case (N has 2 real children):
  - We want to make as little changes to the tree structure as possible:
  - Replace node N with its successor (with the next larger key)

Remove node *N* with key *k:* difficult case



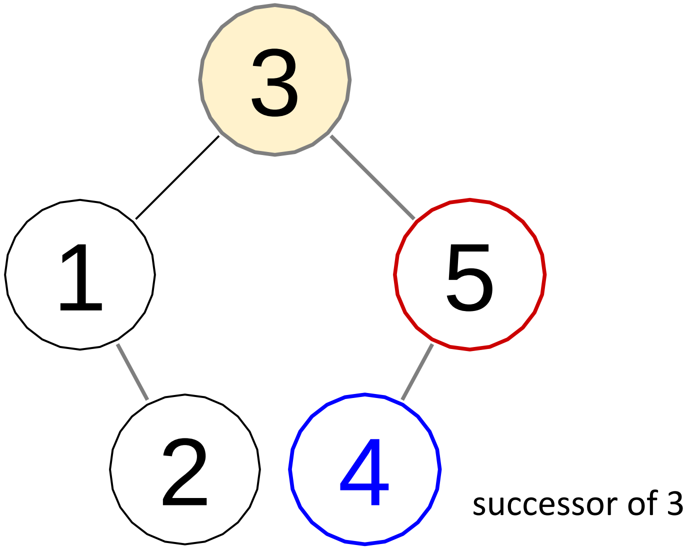
- Difficult case (N has 2 real children):
  - Replace node N with its successor (with the next larger key)
  - Luckily we know that N has the right child
  - To find successor - look for a min in its right subtree

Example: *remove*(3)



- Difficult case (N has 2 children):
  - Replace node N with its successor (with the next largest key)
  - To find successor - look for a min in its right subtree

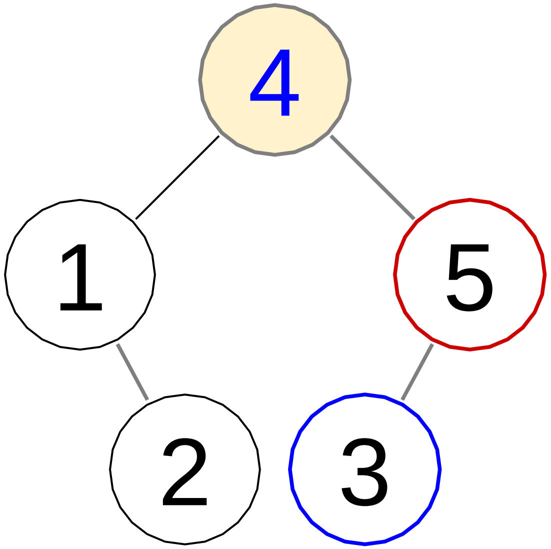
Example: *remove*(3)



➢ Difficult case (N has 2 children):

- Replace node N with its successor (with the next largest key)
- To find successor - look for a min in its right subtree
- Swap data in N and its successor

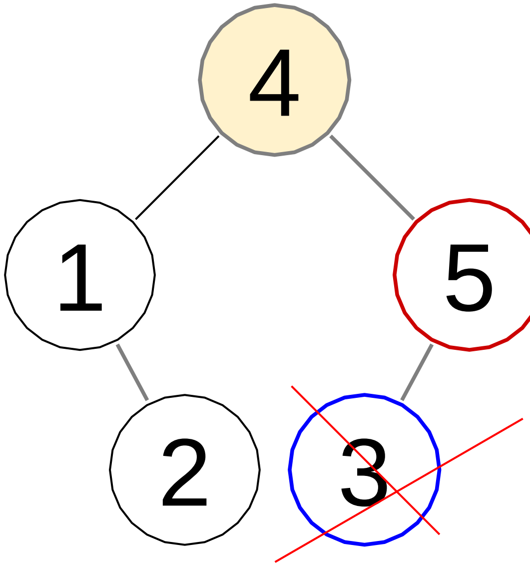
Example: *remove*(3)



➢ Difficult case (N has 2 children):

- Replace node N with its successor (with the next largest key)
- To find successor - look for a min in its right subtree
- Swap values in N and its successor

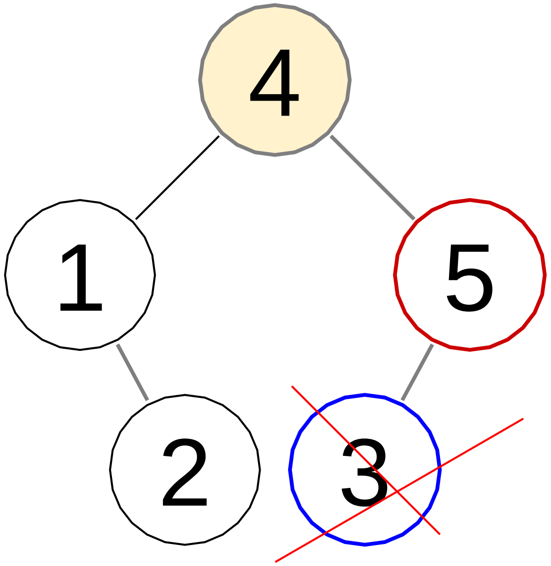
Example: *remove*(3)



➢ Difficult case (N has 2 children):

- Replace node N with its successor (with the next largest key)
- To find successor - look for a min in its right subtree
- Swap values in N and its successor
- Remove successor: this would be easy - why?

Example: *remove*(3)



➢ Difficult case (N has 2 children):

- Replace node N with its successor (with the next largest key)
- To find successor - look for a min in its right subtree
- Swap values in N and its successor
- Remove successor: this would be easy - why?

The successor **does not have a left child!**

(it was a min in the right subtree - which was the last possible left node)

Algorithm ***Remove***(*k*, *R*)

```
if k < R.Key:
    if R.left == Null:
        ERROR: key k is not in the tree
    else if R.left.key == k:
        removeLeftChild(R, R.left)
    else:
        Remove( k, R.left)
if k > R.Key:
    if R.right == NullTree:
        ERROR: key k is not in the tree
    else if R.right.key == k:
        removeRightChild(R, R.right)
    else:
        Remove( k, R.right)
```

Algorithm ***removeRightChild*** *(parent P, child C)*

```
//at least one child of C is Null
//promote the other child in place of C
if C.left == Null:
        Set P.____ = C.____     //promote other child

else if C.right == Null:
        Set P.____ = C.____    //promote other child

//both children of C are real nodes
else:
        ...
```

Fill in missing code:

A. **left right right left**
B. **right left right right**
C. **right right right left**
D. None of the above (something else)