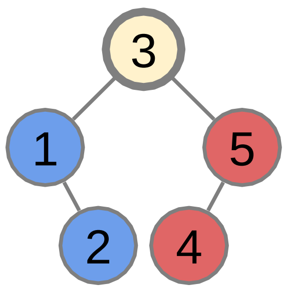
ADT and Data structures.
Binary Search Tree

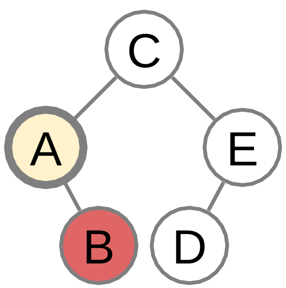
[Review 02.06]

by Marina Barsky

Definition

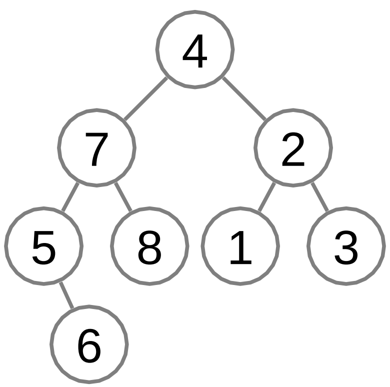
Binary search tree is a binary tree with the following property:
for each node with key $\boldsymbol{x}$, all the nodes in its **left subtree** have keys **smaller than $\boldsymbol{x}$**, and all the keys in its **right subtree** are **greater or equal* to $\boldsymbol{x}$**.



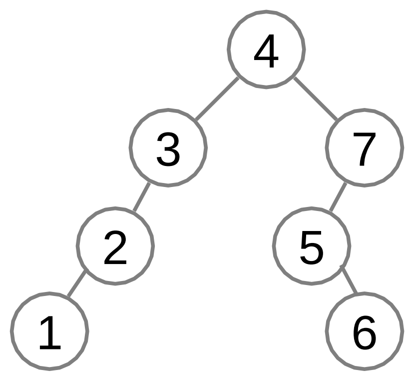


*To simplify the discussion we will assume that all keys are unique, so the keys in the right subtree are strictly greater than *x*

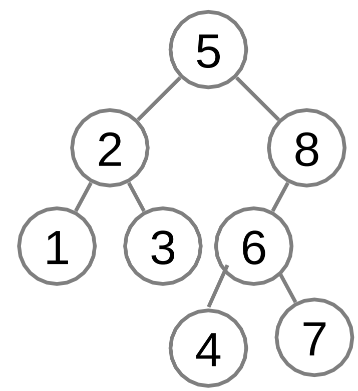
Which one is a Binary Search Tree?



A



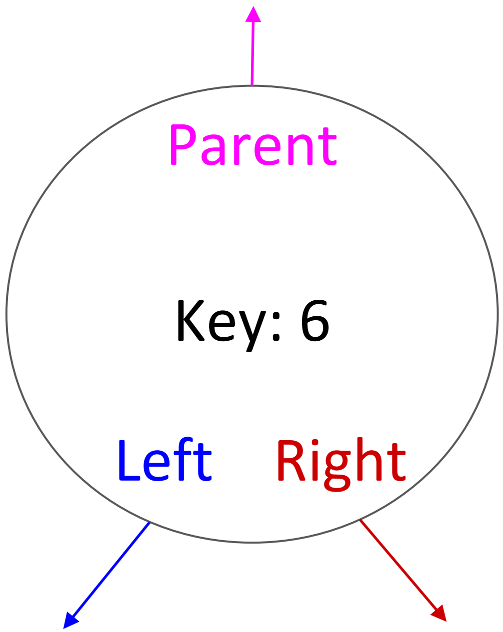
B



C

BST Node

```
BST Node:
     Key
     Left
     Right
     Optional: Parent
```



BST: read operations

- **Find (*k*)**: returns tree node with key *k*
- **Successor (*k*)**: finds and returns the node in the tree with the smallest key among all keys greater than *k* - i.e. finds the node with the next to *k* key in the list of sorted keys
- **Predecessor (*k*)**: same as successor, but from the left of *k* - finds and returns the node with the key immediately preceding *k* in the sorted list of all keys
- **Range (*lo*, *hi*)**: returns the list of all tree nodes with keys between *lo* and *hi* (inclusive)

All these operations do not modify the tree

Operation ***Find***

Input: Key k, Root R of BST

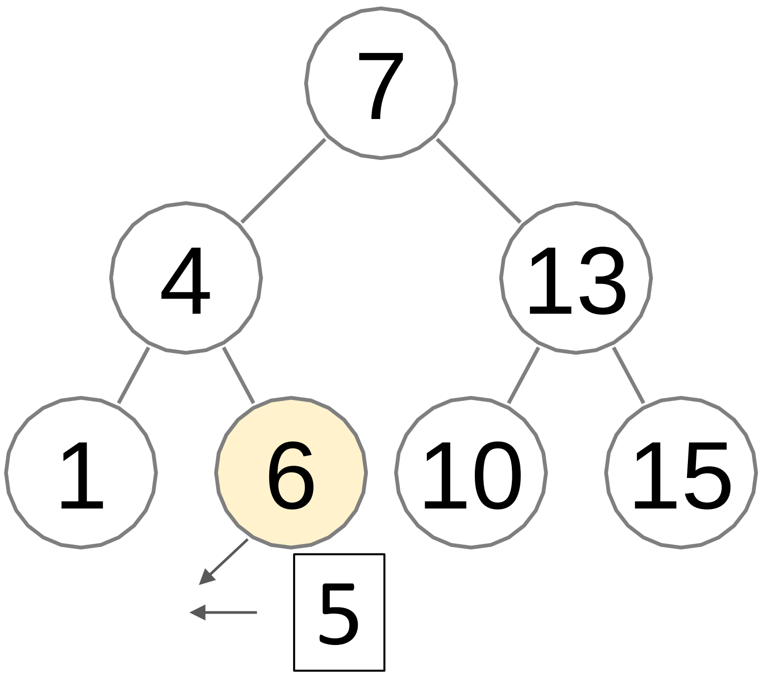
Output: The node with key k

Algorithm *Find* (k, R)

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

Also works for the case of missing key

Missing key: find(5, R)



Note: If you stop before reaching null pointer, you find the place in the tree where *k* would fit.

Given a node N in a Binary Search Tree
- find nodes with adjacent keys

Operation ***Successor***

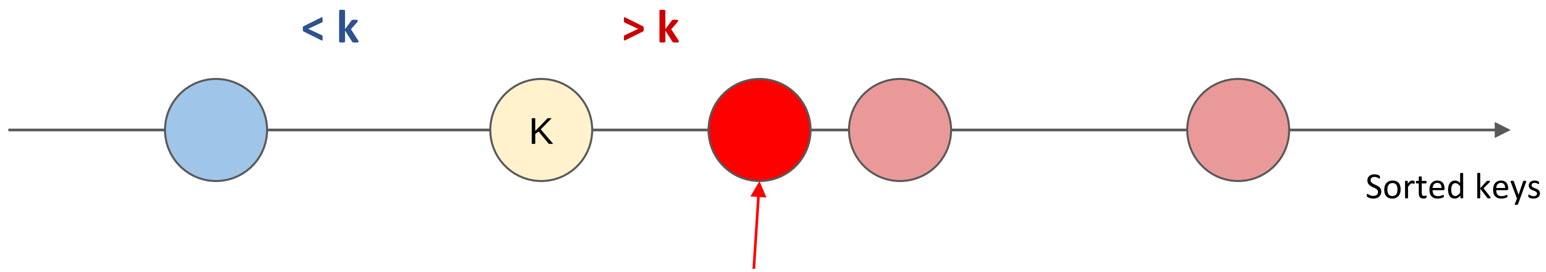
Input: key k

Output: The node in the tree with the next larger key.

Operation ***Predecessor***

Input: key k

Output: The node in the tree with the next smaller key.



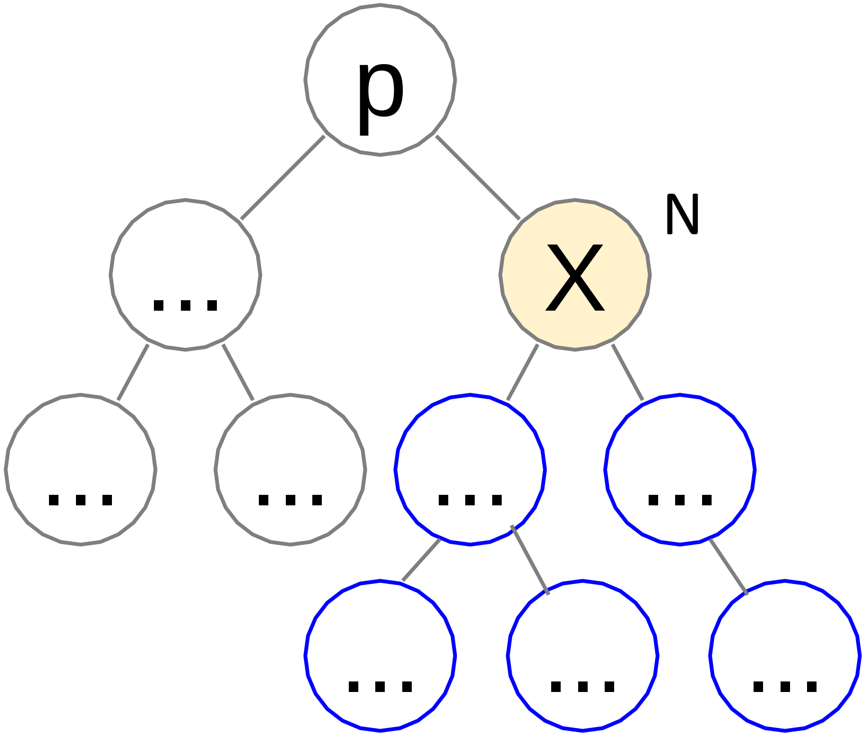
Operation ***Successor***

Input: key *k*

Output: The node in the tree with the next larger key.

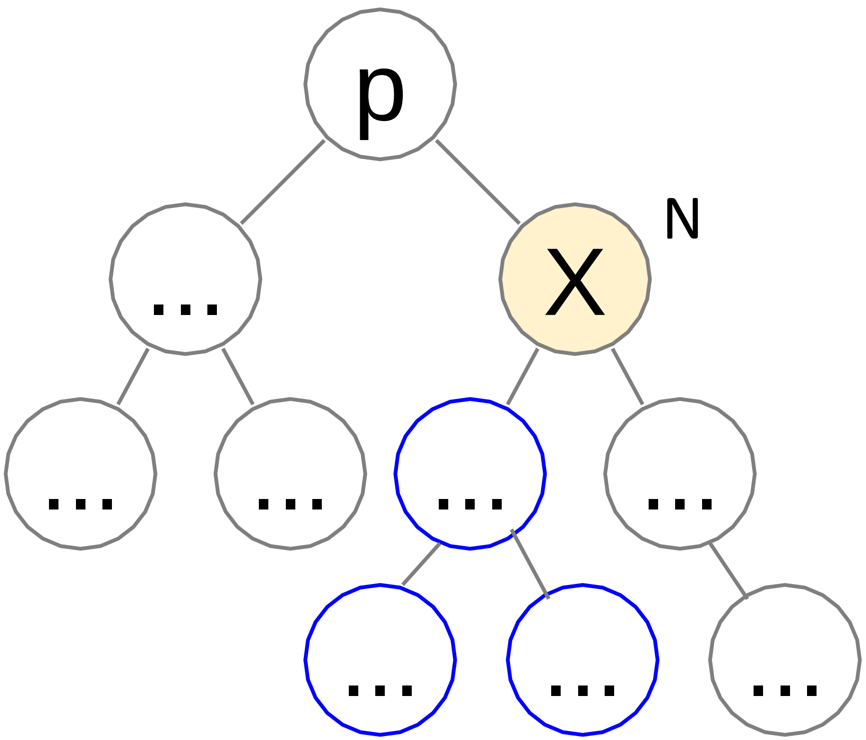
- We want to find the node with the key which is closest to *k* from above
- We would need a sub-operation ***get_min*** to solve this problem

Sub-operation: *get_min* (node *N*)



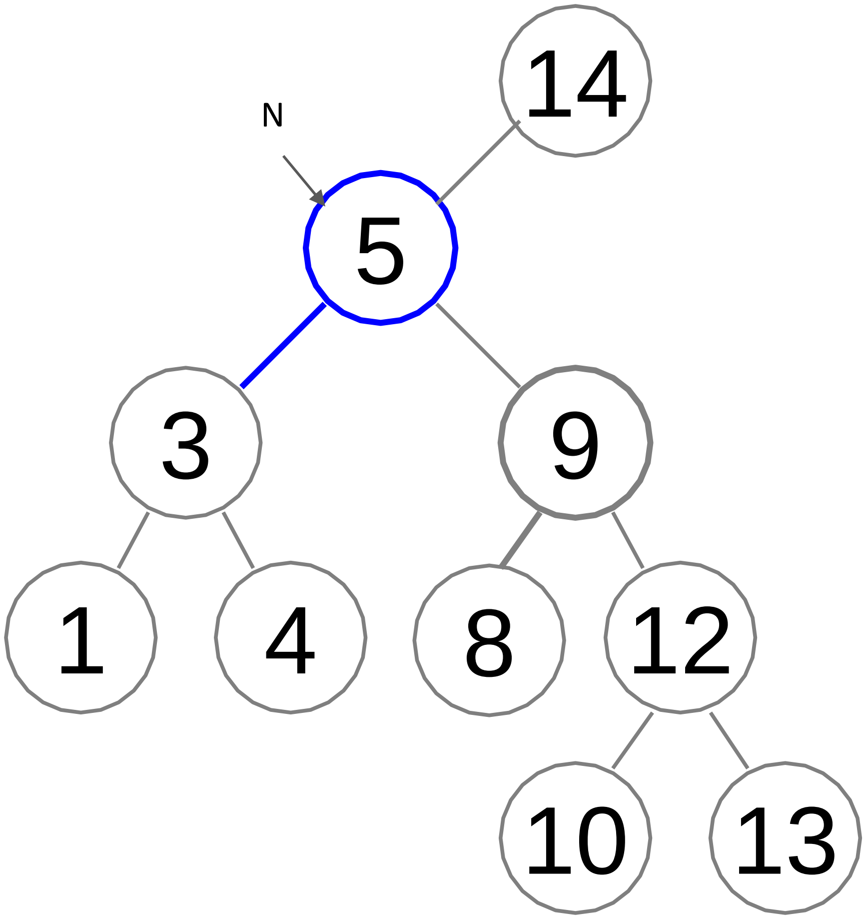
➢ We want to find min key in a subtree rooted at *N*

Sub-operation: *get_min* (node *N*)



- We want to find min key in a subtree rooted at *N*
- Among all descendants of *N* the only keys that are < *X* are in the left subtree of N

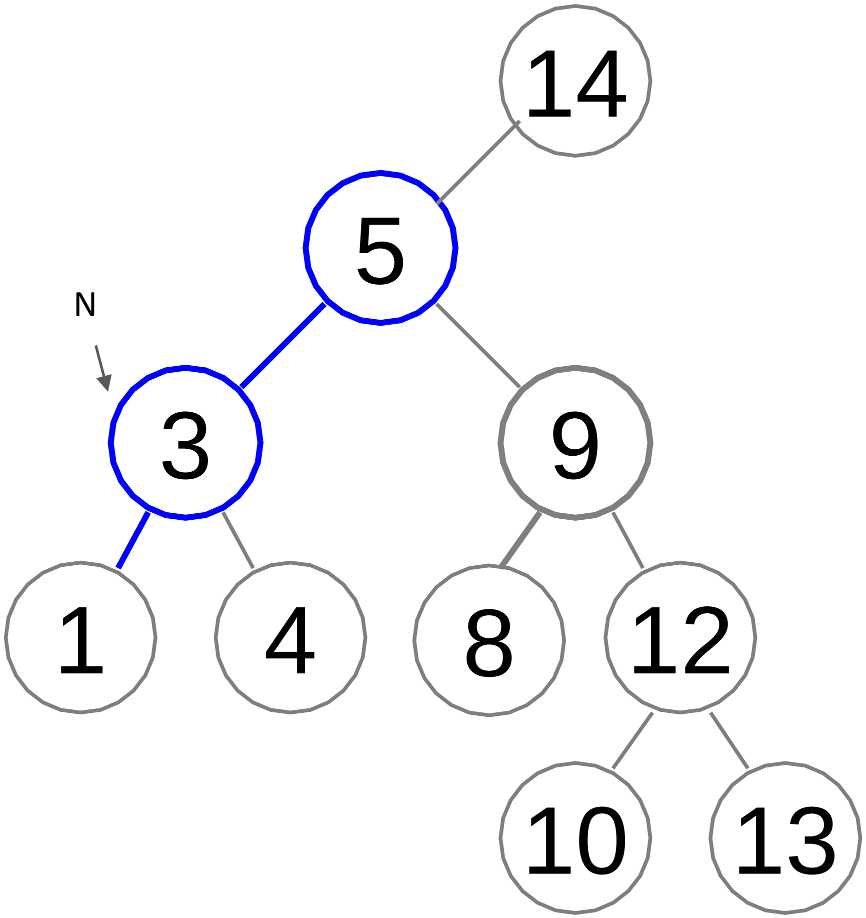
Example: *get_min* (N)



➔ Does node N have left child?
Yes → there is a key smaller than 5

➔ Set N to be the left child and ask the same question

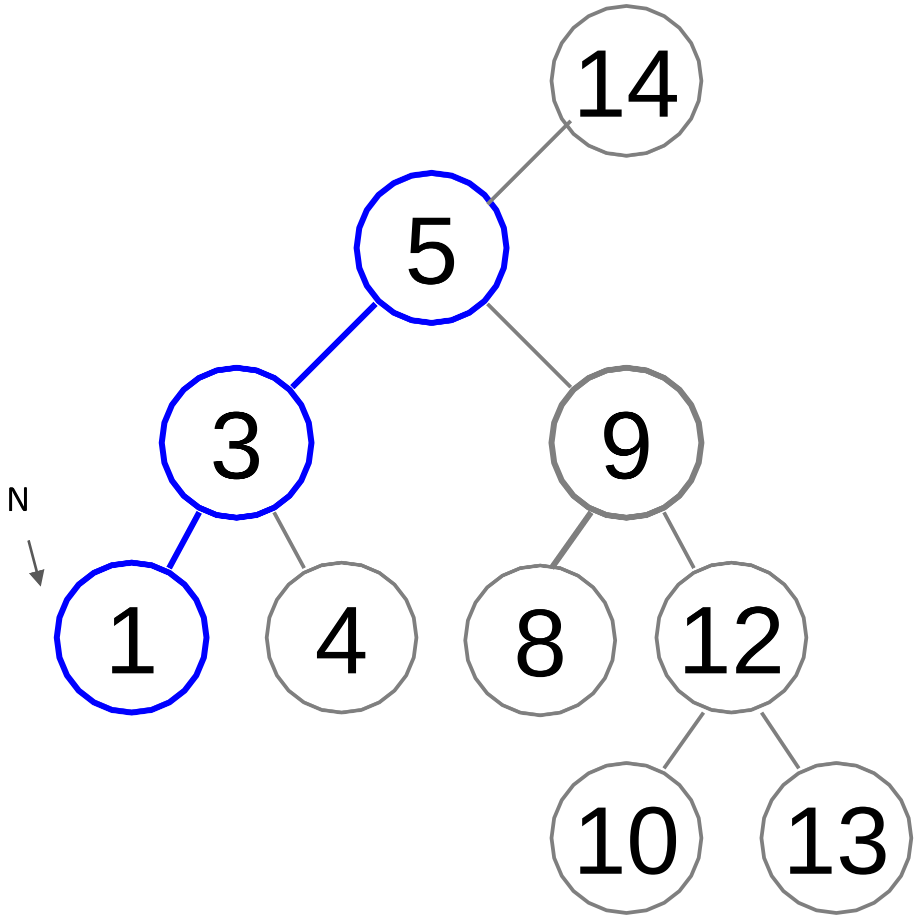
Example: *get_min* (N)



➔ Does node N have left child?
Yes → there is a key smaller than 3

➔ Set N to be the left child and ask the same question

Example: *get_min* (N)



➔ Does node N have left child?
No → there is no key smaller than N

Follow the leftmost path in the tree - until no more left child

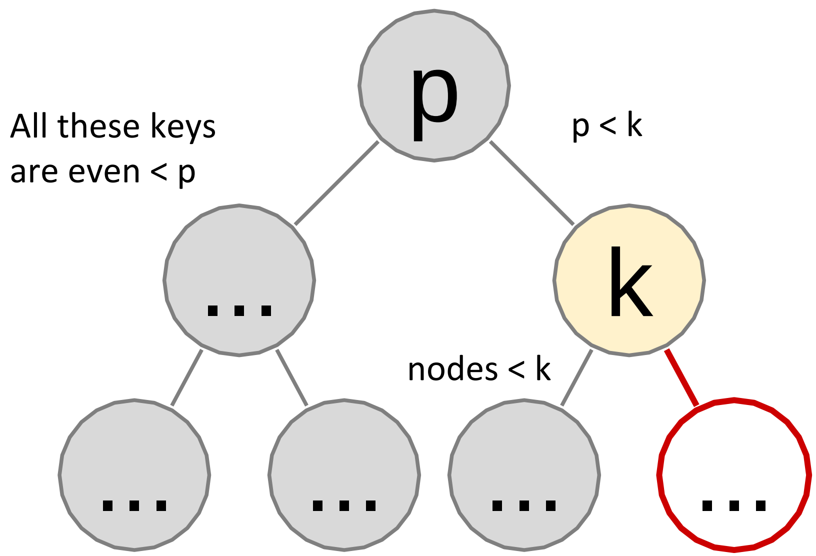
Algorithm *Get_min* (*N*)

```
if N.Left = null:
  return N
else:
  return Get_min (N.Left)
```

Successor (k)

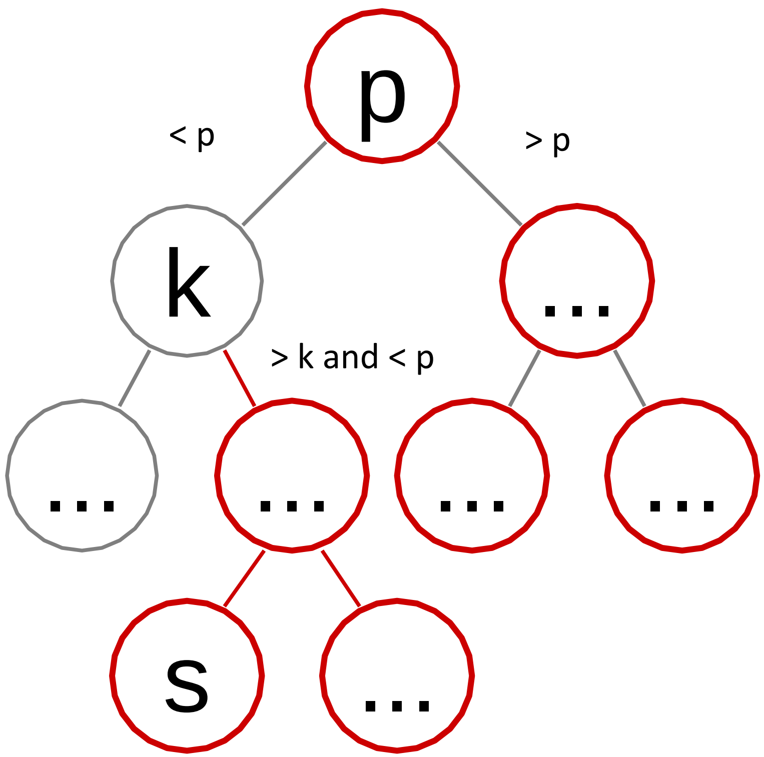
First, find node *N* with key *k*

Case 1: *N* has right child



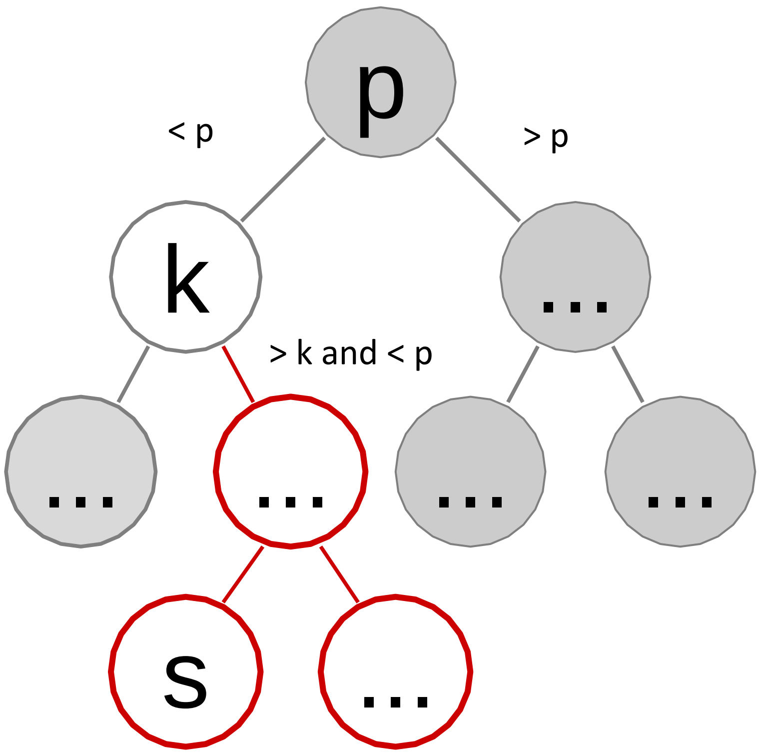
- In this situation all keys > *k* are in the right subtree of *N*

Case 1: Node N has the right child, but also has a parent with p > k



- In this situation there are also keys > *k* in the parent of *N* and in the right subtree of the parent
- However we are looking for the **smallest** among these keys
- The min among all keys > *k* is again in the right subtree of *N* - because the keys in this subtree are precisely between *k* and *p*

Case 1: Node N has the right child, but also has a parent with p > k

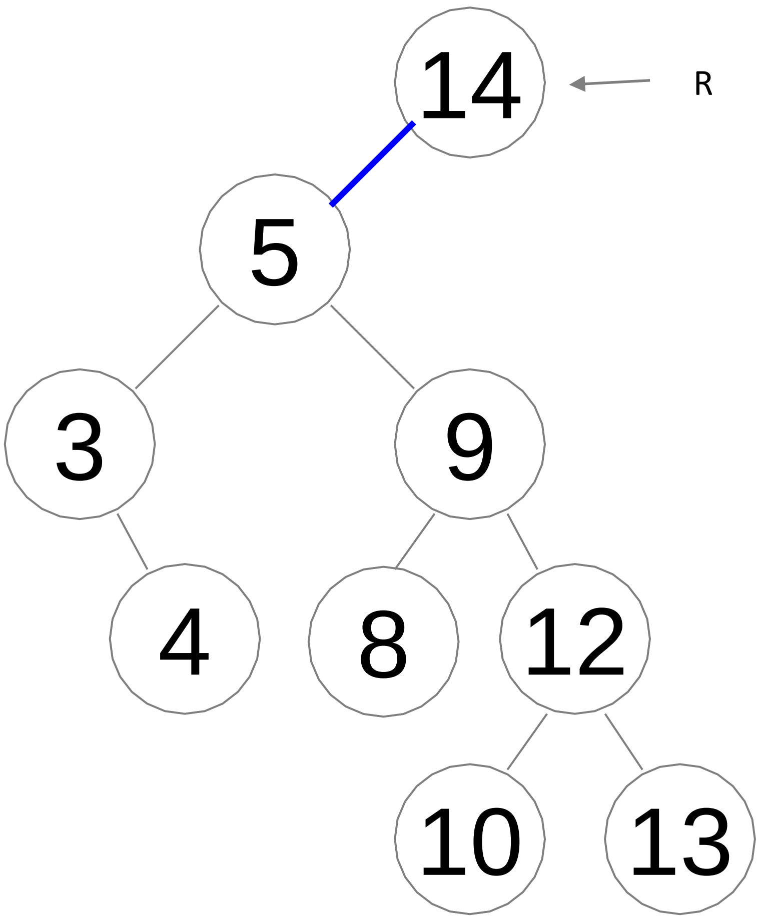


- ➢ The goal then becomes to find the smallest among all the keys in the right subtree of *N*
- ➢ Use *get_min* (*N*.right)

Algorithm *Successor* (*k*, *R*)

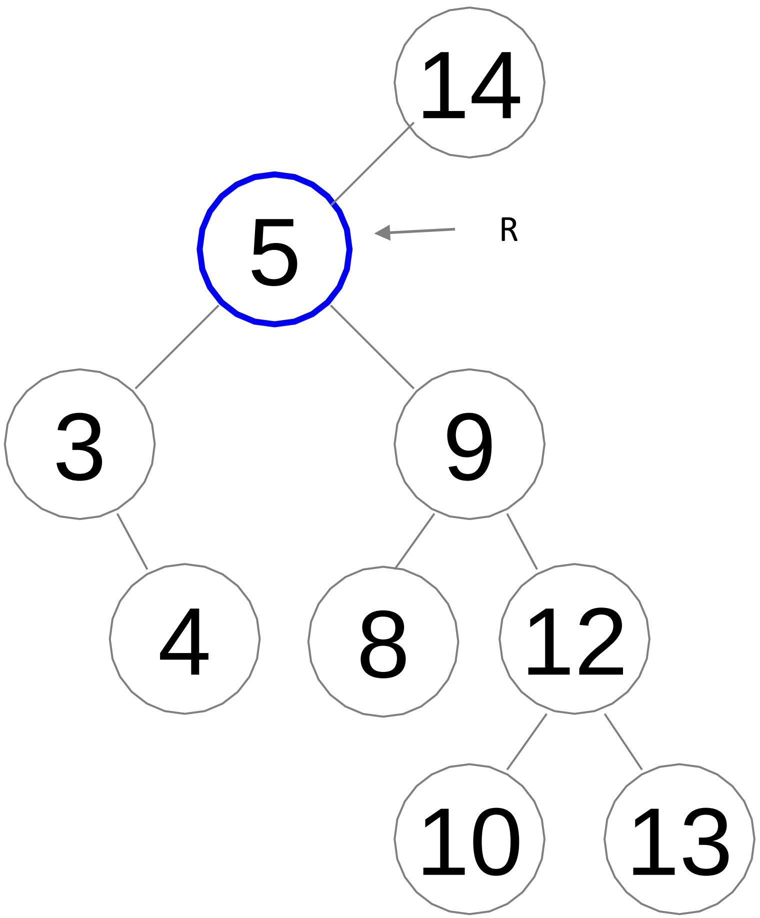
```
if R.Key = k : # found N
        if R.Right != null :
        return Get_min (R.Right)
        ...
if k < R.Key: # continue searching for N
    return Successor (k, R.Left)
        ...
if k > R.Key : # continue searching for N
    return Successor (k, R.Right)
       ...
```

Example: successor (5, R)



➔ Follow the left subtree: 5 < 14

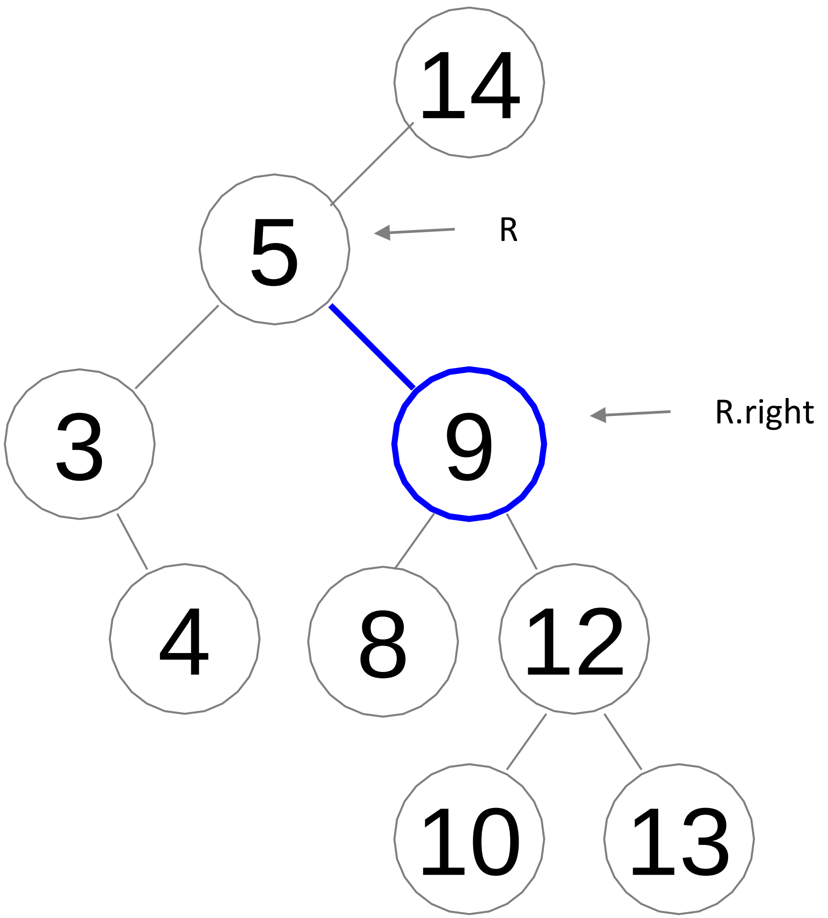
Example: successor (5, R)



➔ Follow the left subtree: 5 < 14

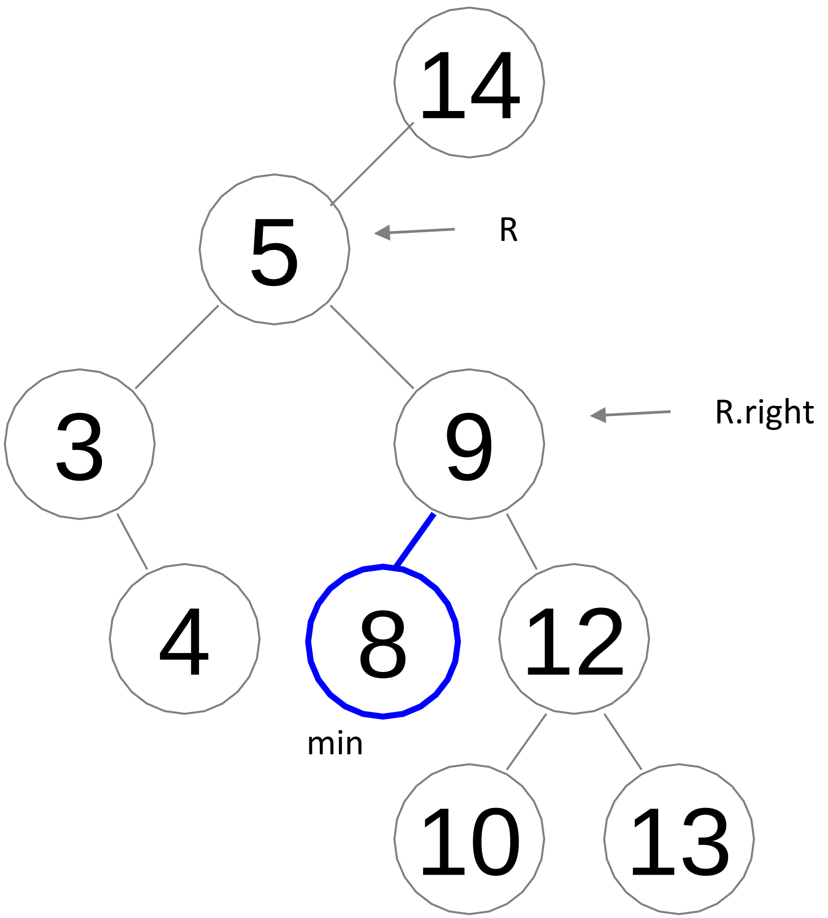
➔ Found 5

Example: successor (5, R)



- ➔ Follow the left subtree: 5 < 14
- ➔ Found 5
- ➔ *N* has right child

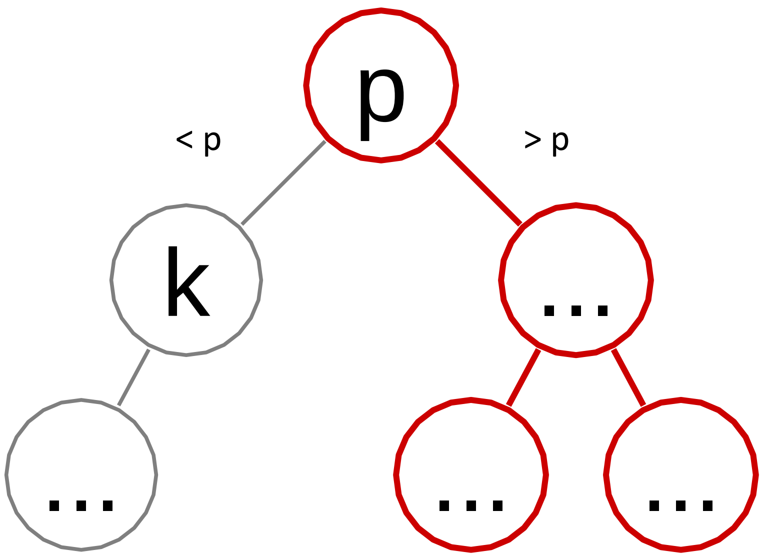
Example: successor (5, R)



- ➔ Follow the left subtree: 5 < 14
- ➔ Found 5
- ➔ *N* has right child
- ➔ *Min* in the subtree rooted at 9 is the successor of 5

successor (5, R)→8

Case 2: Node *N* with key *k* does not have the right child



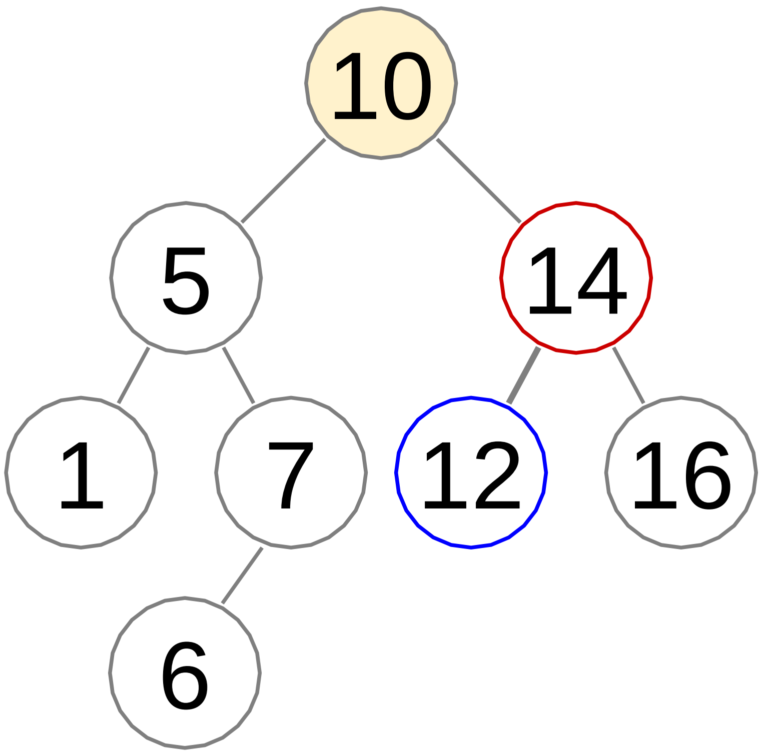
- In this case the successor of *N* is among *N*'s ancestors
- Namely the last time we took the turn to left subtree - the key at the root of this subtree is the successor of *N*
- If we do not have a parent field in our Node, then we cannot recover this parent
- Instead, we will keep track of the last parent when we took the left turn in the search for *N*

Algorithm *Successor* (*k*, *R*, *S*)

```
if R.Key = k : # found N
    if R.Right != null :
        return get_min (R.Right)
    else:
        return S
if k < R.Key : # left turn
    S ← R # remember the parent
    return Successor (k, R.Left, S)
if k > R.Key:
    return Successor (k, R.Right, S)
```

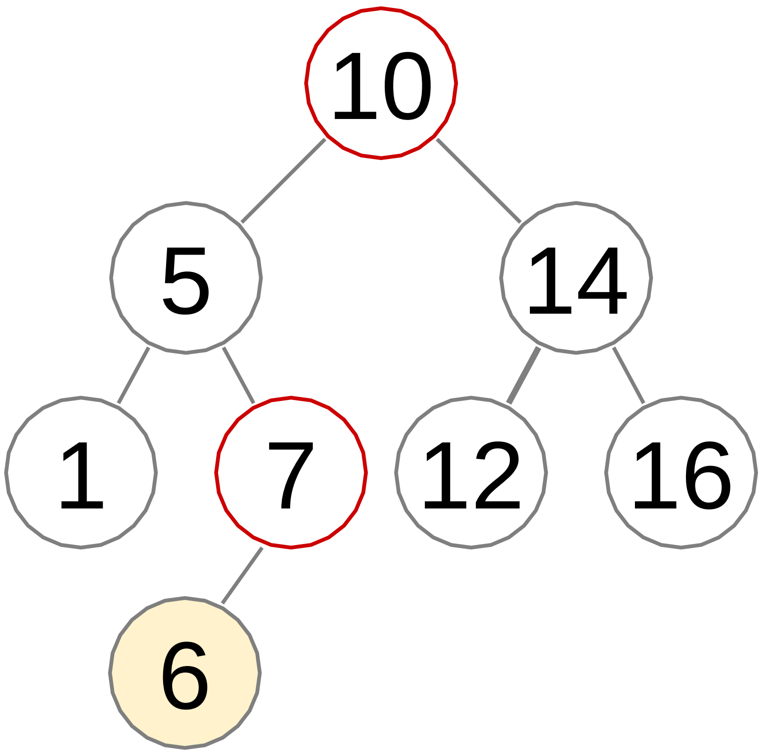
You start this algorithm with *R* = root of BST and *S* (successor) set to null

Example: *Successor* (10, R)



- ➔ 10 has right subtree
- ➔ Successor is the min in this right subtree: *Successor* (10) → 12

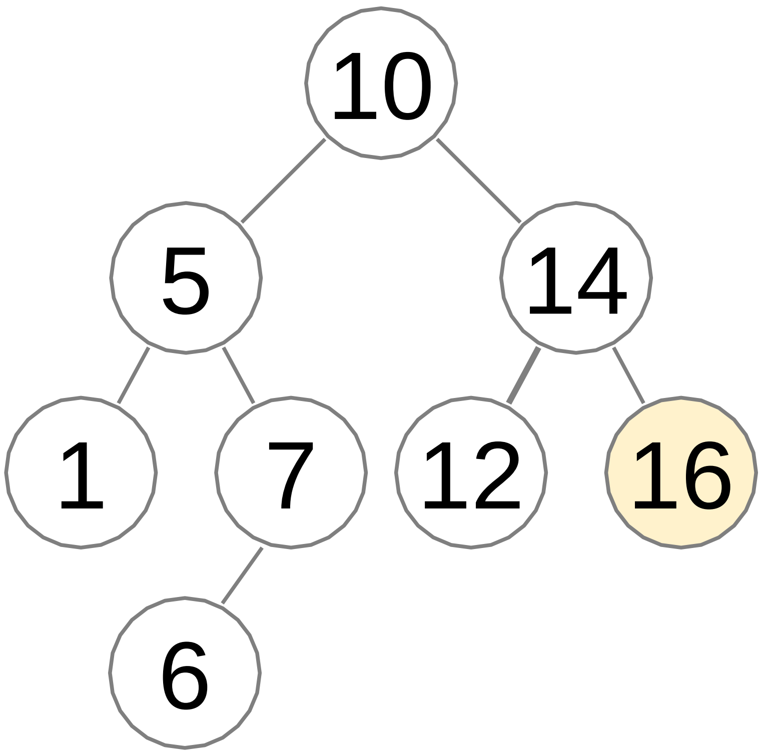
Example: *Successor* (6, R)



- ➔ While searching for 6: we update a possible candidate for successor (first 10, then 7) - because we do not know if *N* will have a right subtree or not
- ➔ 6 does not have the right subtree
- ➔ Successor is the last ancestor of 6 when we moved into the left subtree:

  Successor (6) → 7

Example: *Successor* (16, R)



➔ While searching for 16: we never took the left turn

➔ 16 does not have the right subtree

➔ 16 also does not have a successor - it is the largest key in the tree

Successor (16) → null

Now that we know how to find a successor, we can solve the range query

Operation *Range*

Input: Numbers x, y, root R

Output: A list of nodes with keys between x and y

Algorithm RangeSearch (x , y , R)

$L \leftarrow$ `empty list`

$N \leftarrow$ ***Find***(x , R)

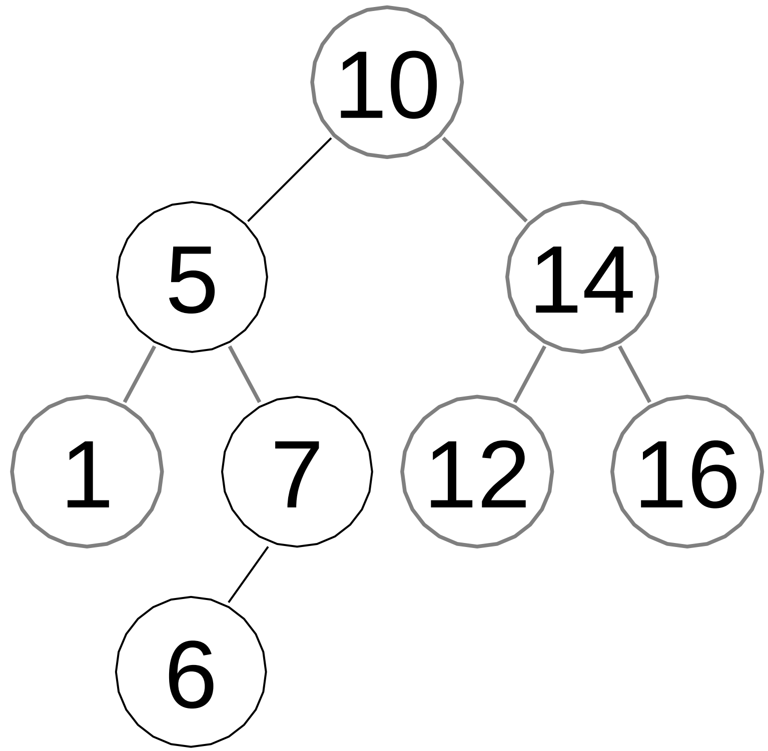
`while` N `is not Null and` N.`Key` $\leq y$

  $L \leftarrow L + N$

  $N \leftarrow$ ***Successor*** (N.`Key`, R, `Null`)

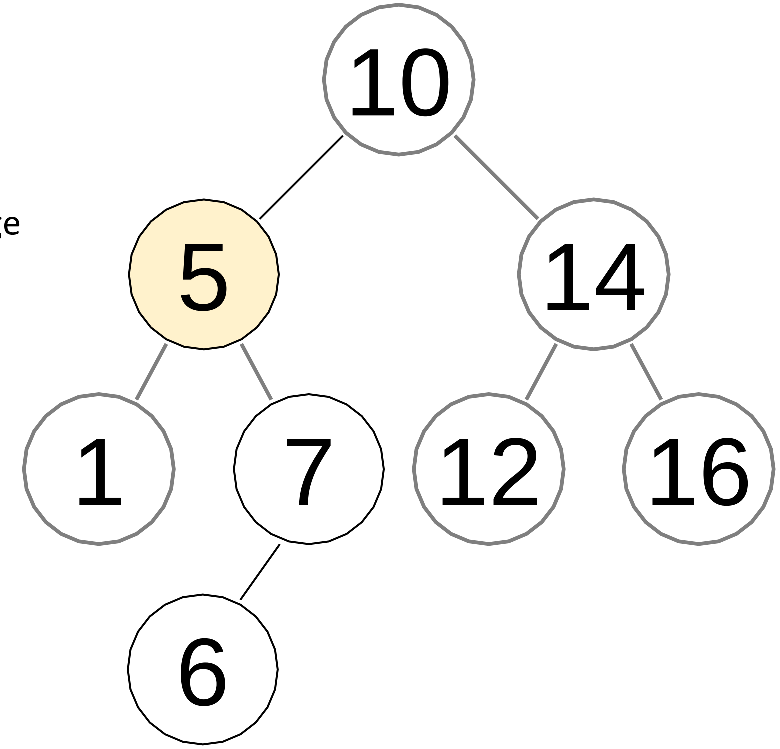
`return` L

Example: range search (5, 13)



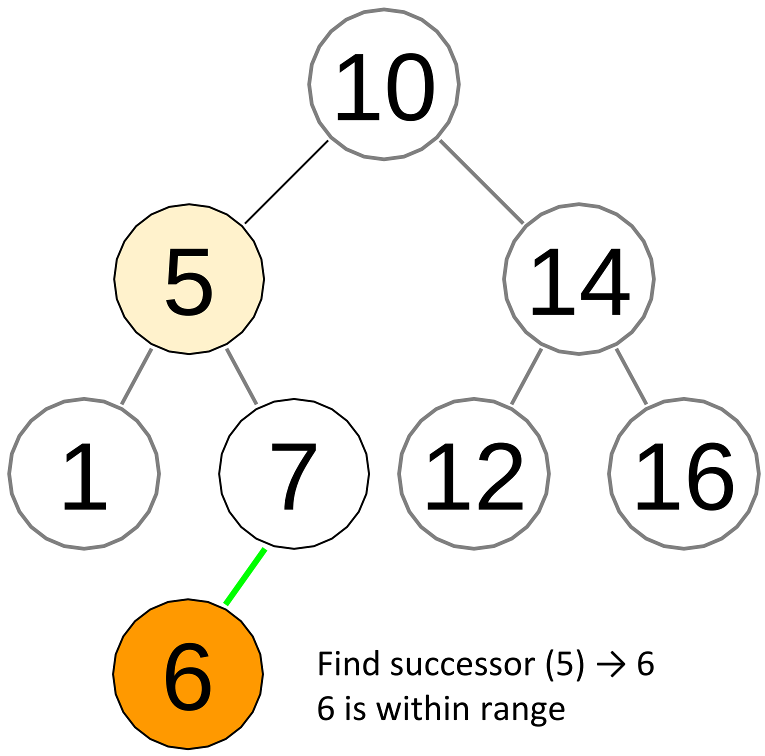
Example: range search (5, 13)

Find 5
5 is within range



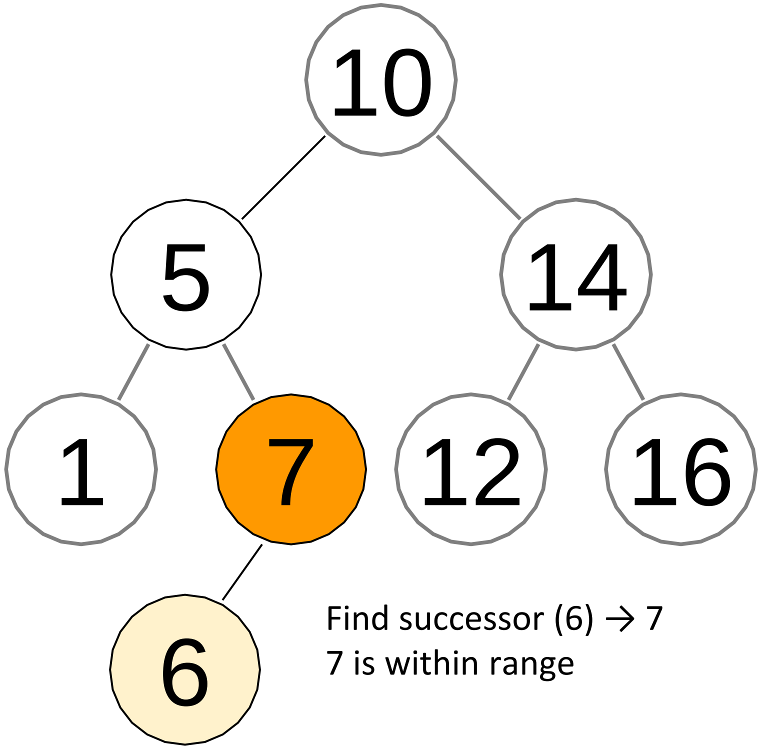
Result: 5

Example: range search (5, 13)



Result: 5, 6

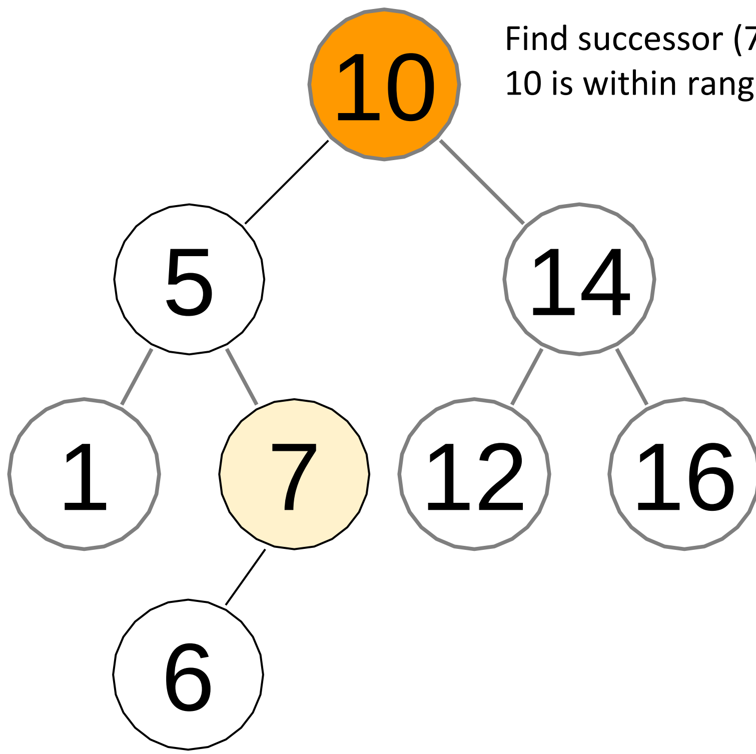
Example: range search (5, 13)



Find successor (6) → 7
7 is within range

Result: 5, 6, 7

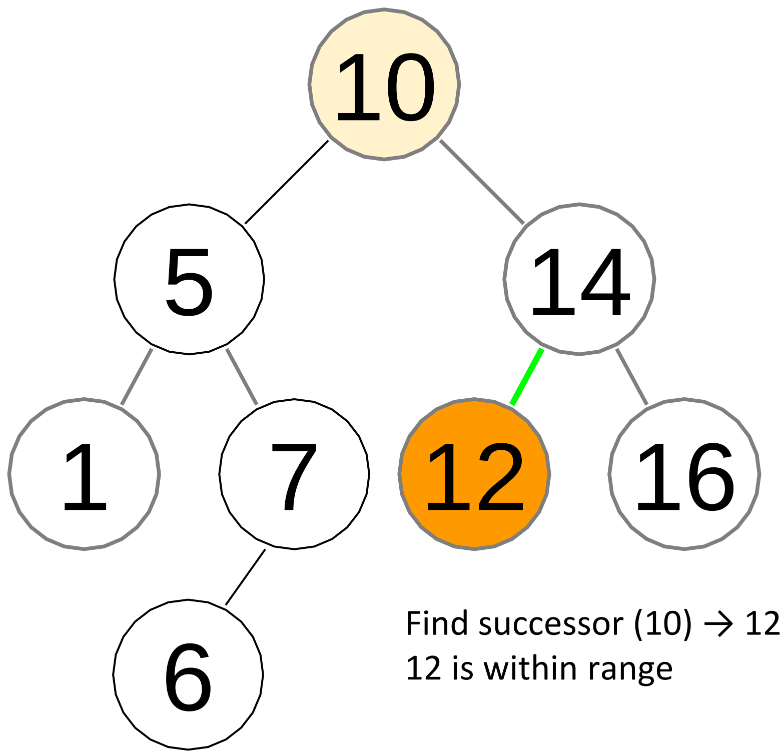
Example: range search (5, 13)



Find successor (7) → 10
10 is within range

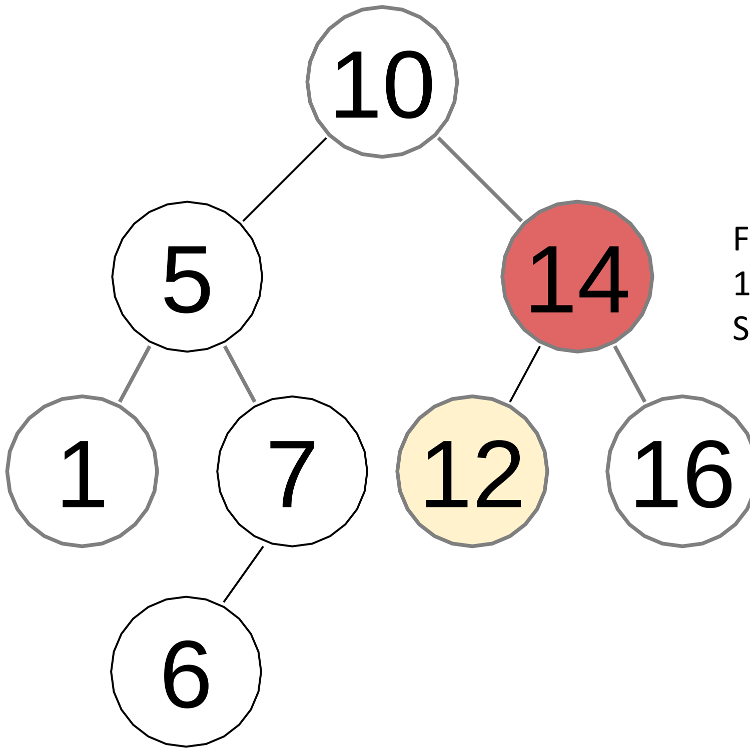
Result: 5, 6, 7, 10

Example: range search (5, 13)



Result: 5, 6, 7, 10, 12

Example: range search (5, 13)



Find successor (12) → 14
14 is outside range
Stop

Result: 5, 6, 7, 10, 12

BST: update operations

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

- ***Delete*** (k): deletes the node with key k such that the BST property of the tree is preserved

We already have all the sub-operations to implement these

Operation ***Insert***

Input: Key k

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

Algorithm *Find* (*k*, *R*)

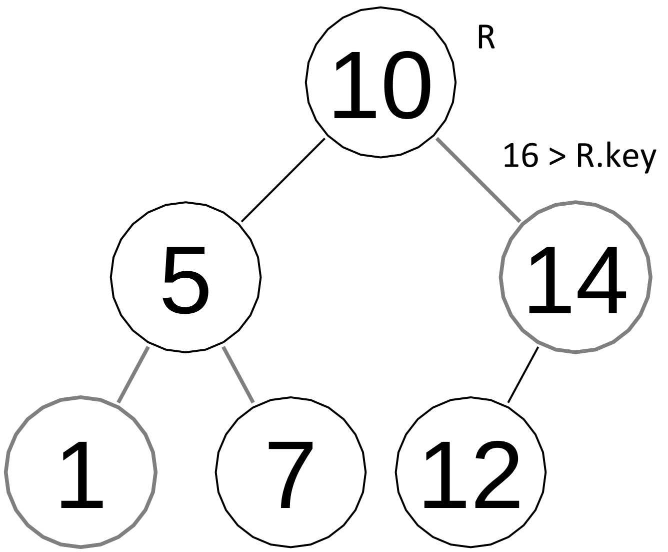
```
if R is Null or R.Key = k:
  return R
if R.Key > k:
  return Find(k, R.Left)
else if R.Key < k:
  return Find(k, R.Right)
```

We need to slightly modify ***Find***

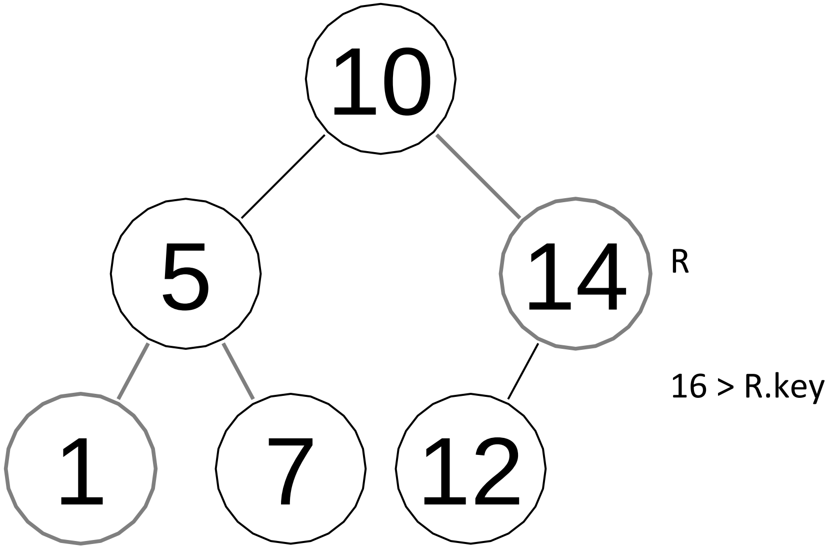
Algorithm *Insert* (*k*, *R*)

```
if R != Null and R.Key = k:
  return ERROR
if R is Null:
      return new Node(k)
if k < R.Key:
      R.left = Insert( k, R.left)
    return R
if k > R.Key:
      R.right = Insert( k, R.right)
    return R
```

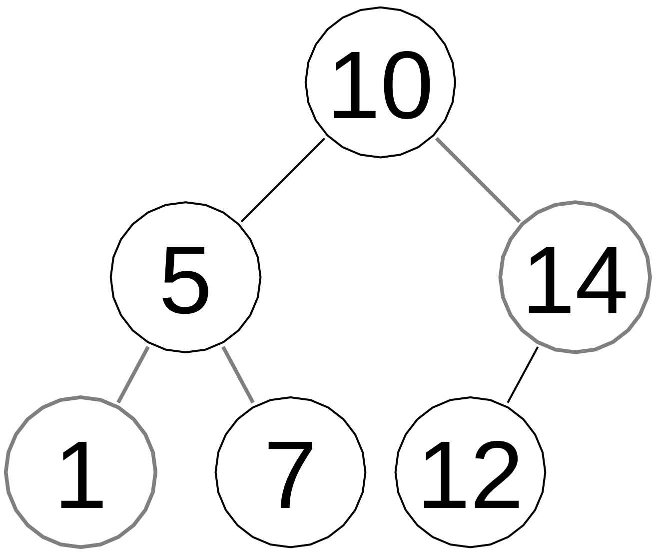
Example: insert (16, R)



Example: insert (16, R)



Example: insert (16, R)

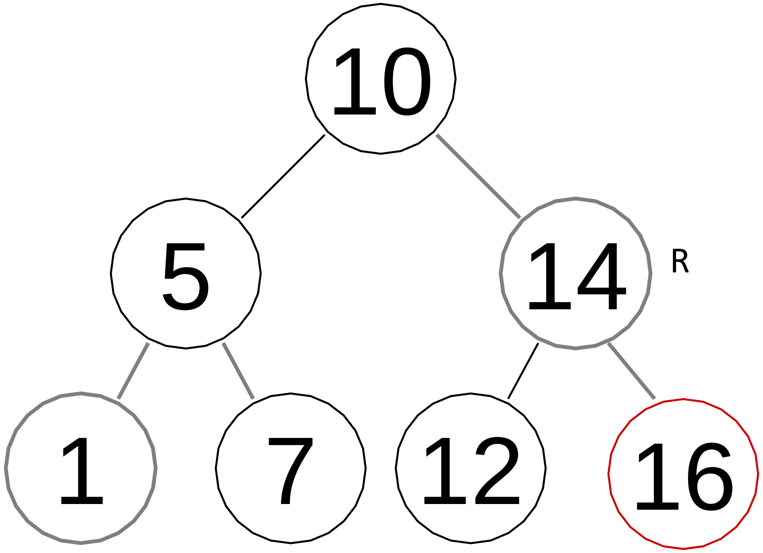


R

16 > R.key

R has no right child - this is the place to insert new node

Example: insert (16, R)



Update right child of R and return updated node 14

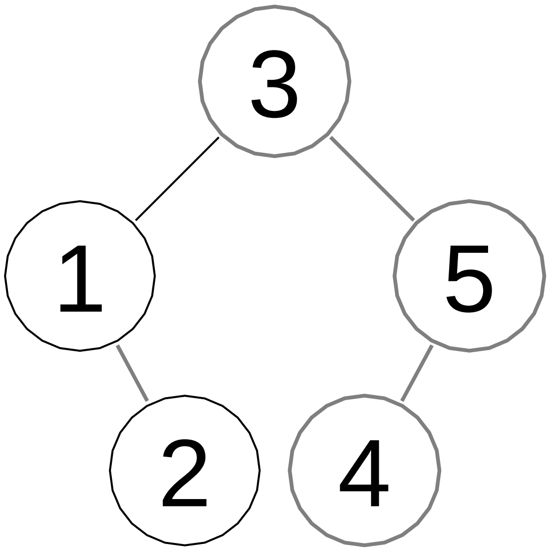
Operation Delete

Input: Key k

Output: BST without node N *with key* k

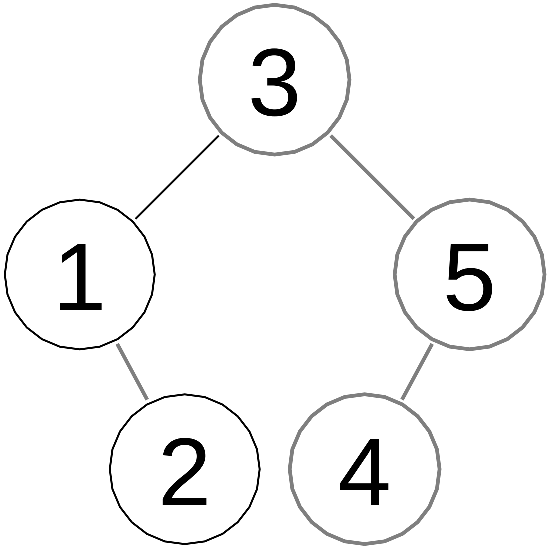
The most challenging algorithm in this module

Delete node *N* with key *k*



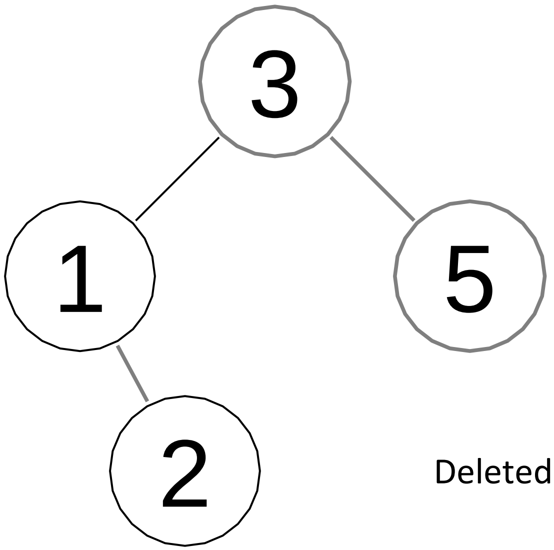
- ➢First, find *N*
- ➢Easy case (*N* has no children)
    - ○Just detach *N* from the tree

Example: *delete*(4)



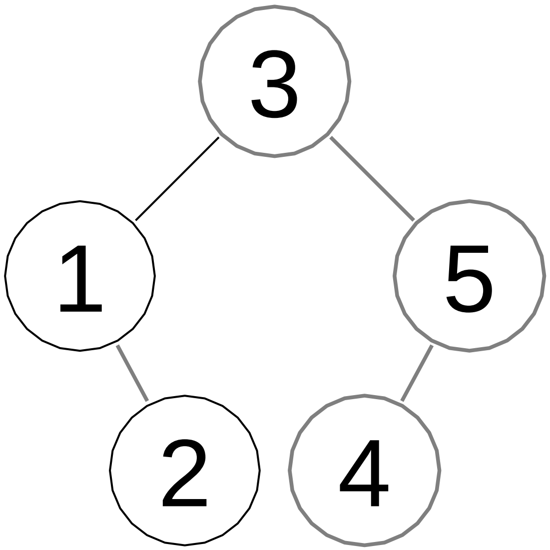
- ➢First, find *N*
- ➢Easy case (*N* has no children)
  - ○Just detach *N* from the tree

Example: *delete*(4)



- First, find *N*
- Easy case (*N* has no children)
    - Just detach *N* from the tree

Delete node *N* with key *k*

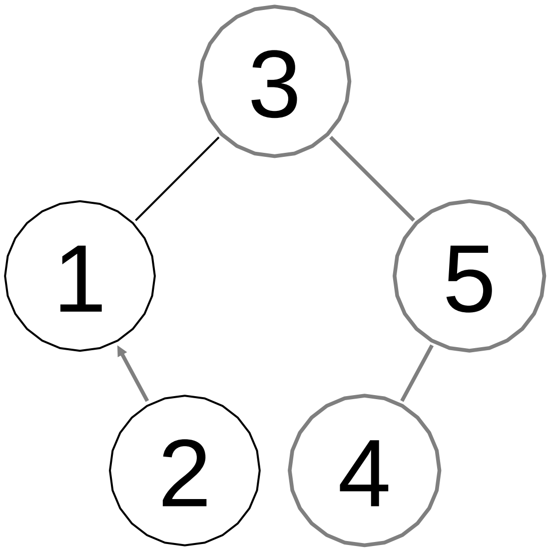


- Medium case (N has one child):

  Just "splice out" node *N*

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

Example: *delete*(1)

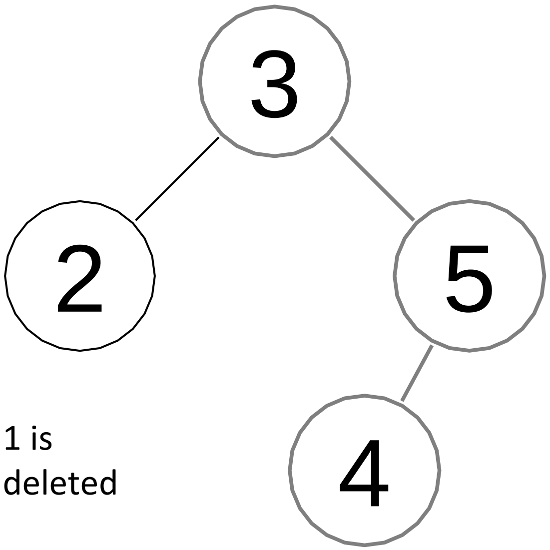


- Medium case (N has one child):

  Just “splice out” node *N*

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

Example: *delete*(1)



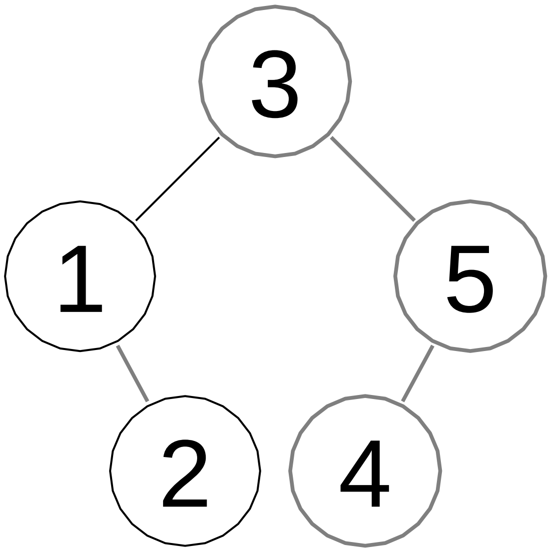
1 is
deleted

➢ Medium case (N has one child):

Just "splice out" node *N*

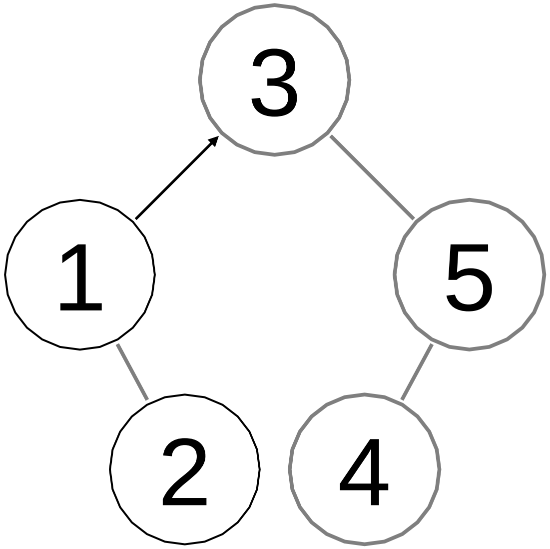
- Its unique child assumes the position previously occupied by *N* – gets ***promoted*** to its place

Delete node *N* with key *k*



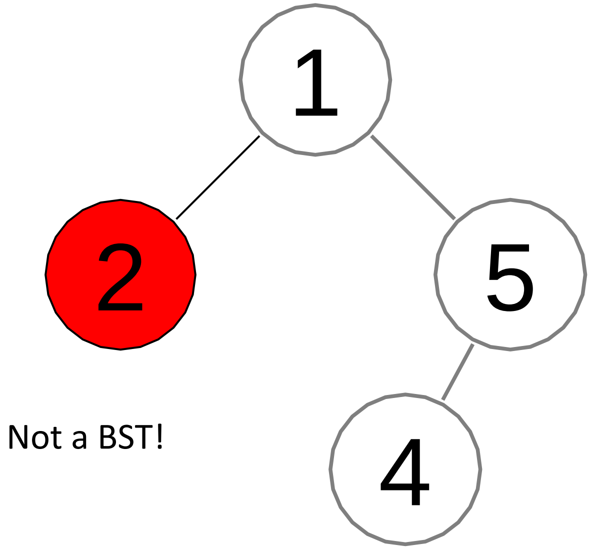
- ➢Difficult case (N has 2 children):

Example: *delete*(3)



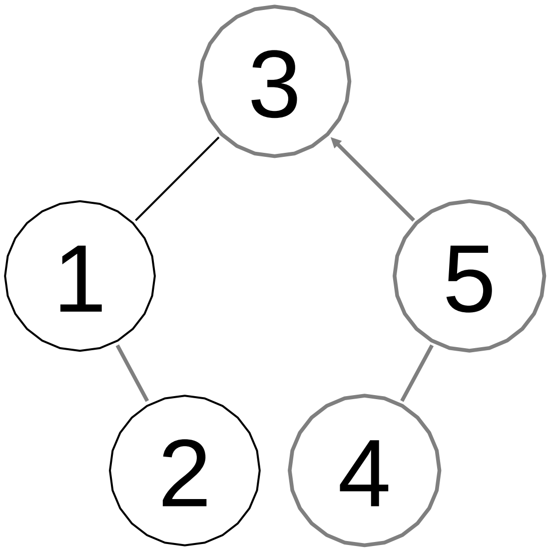
- ➢Difficult case (N has 2 children):
  - Promote 1?

Example: *delete*(3)



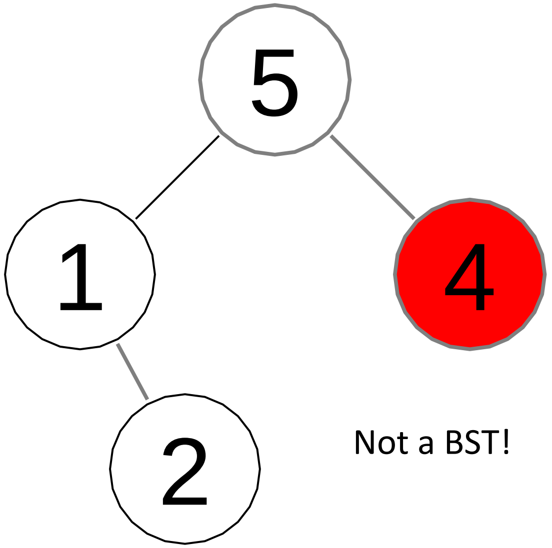
- Difficult case (N has 2 children):
  - Promote 1?

Example: *delete*(3)



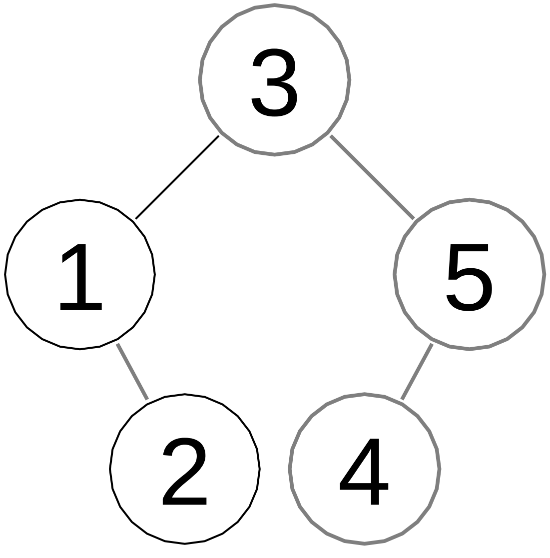
➢ Difficult case (N has 2 children):
- Promote 5?

Example: *delete*(3)



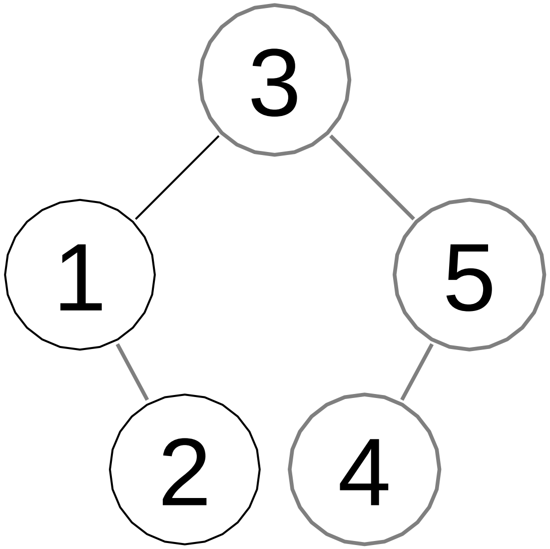
- Difficult case (N has 2 children):
  - Promote 5?

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



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

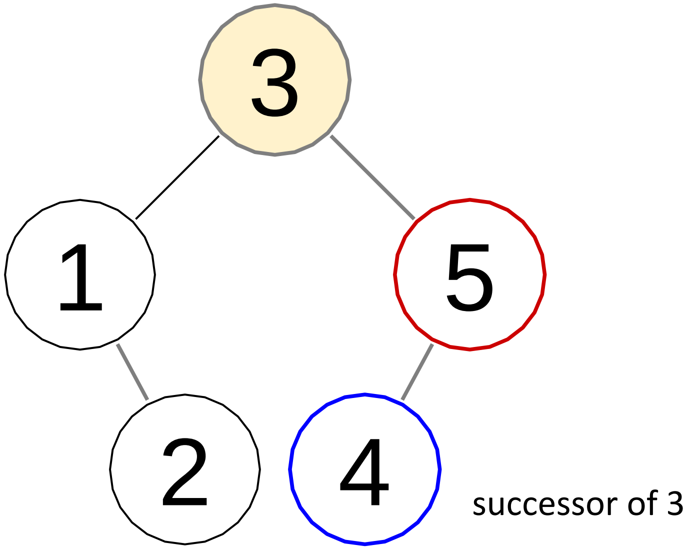
Delete node *N* with key *k:* difficult case



➢Difficult case (N has 2 children):

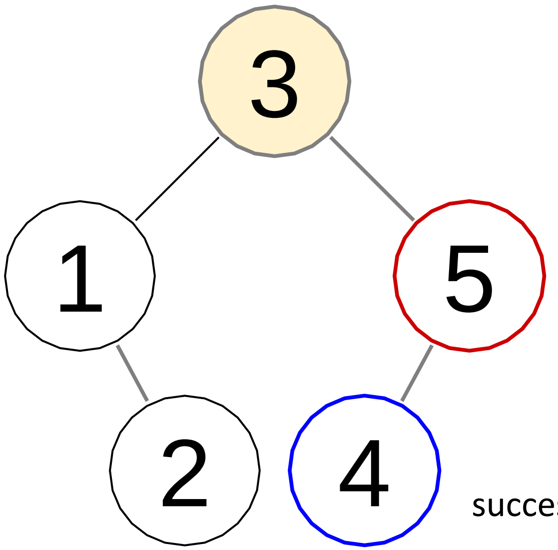
- Replace node N with its successor (with the next largest key)
- Luckily we know that N has the right child
- To find successor - look for a min in its right subtree

Example: *delete*(3)



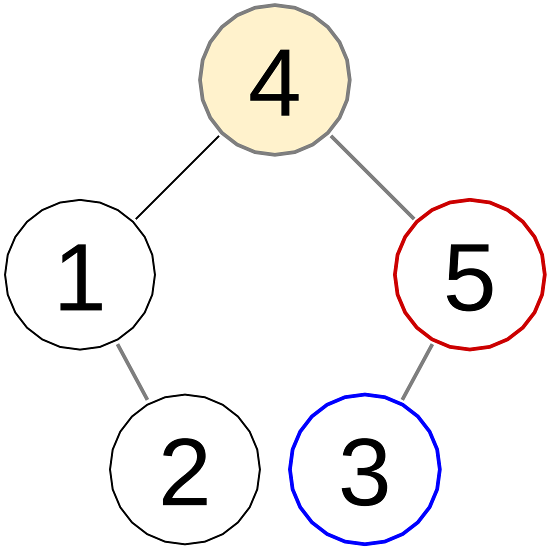
➢ Difficult case (N has 2 children):
- Replace node N with its successor (with the next largest key)
- To find successor - look for a min in its right subtree

Example: *delete*(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

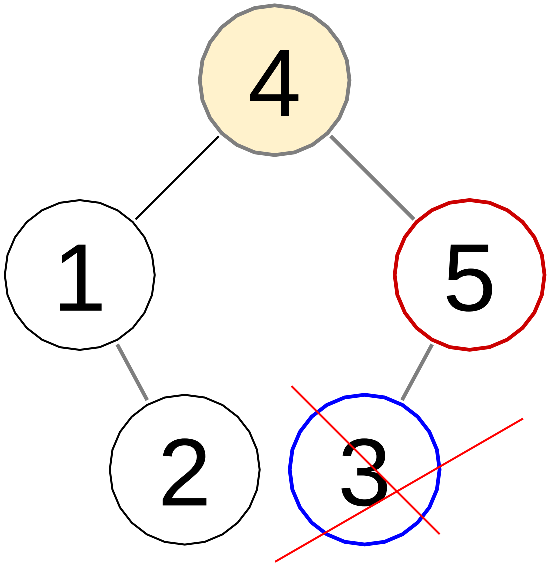
Example: *delete*(3)



➢ 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

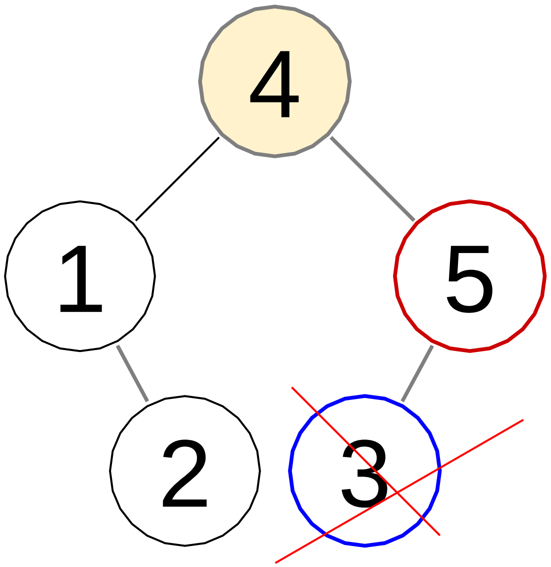
Example: *delete*(3)



➢ 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: *delete*(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)