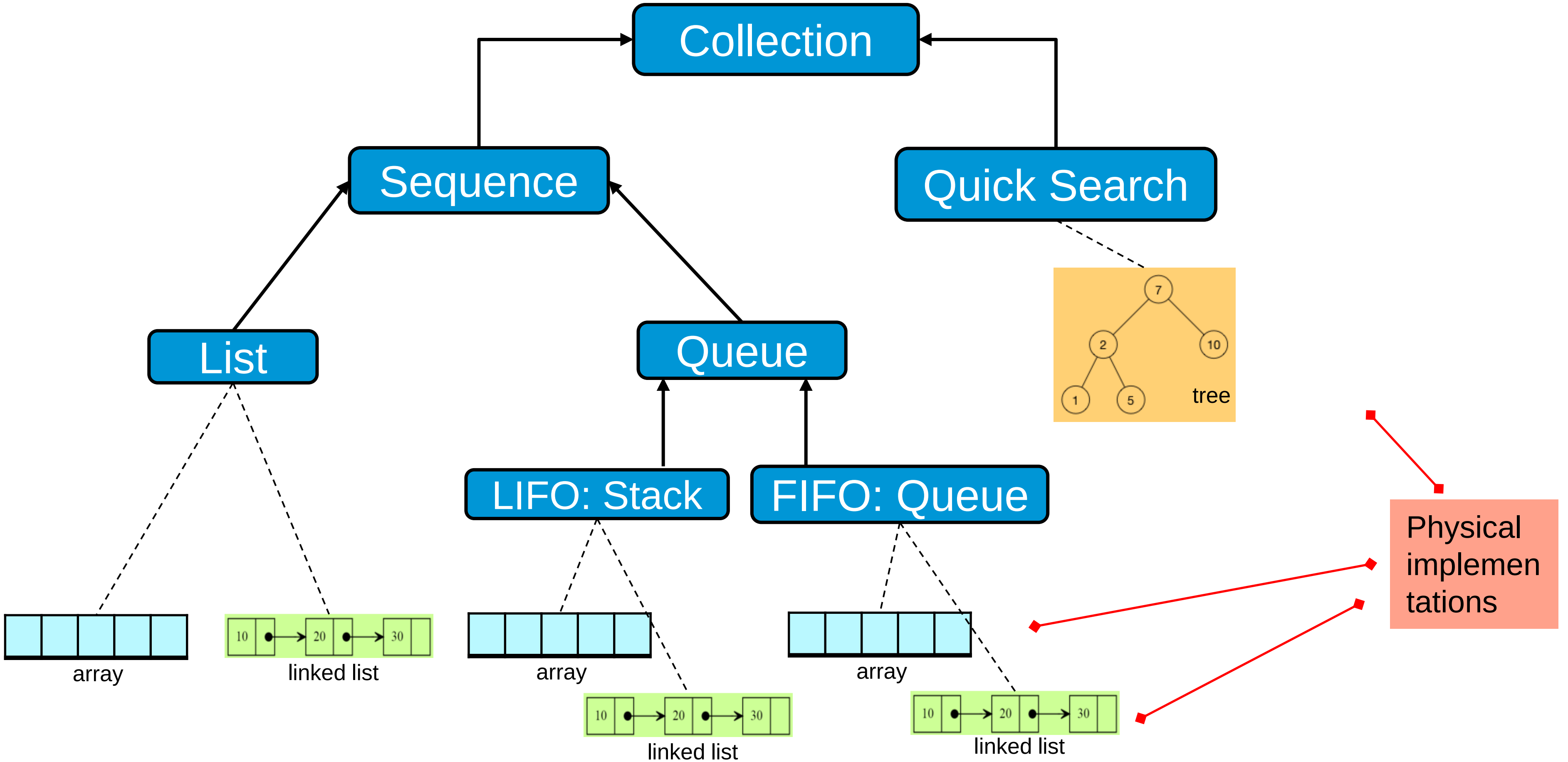
Binary Search Trees
Read operations

Lecture 18

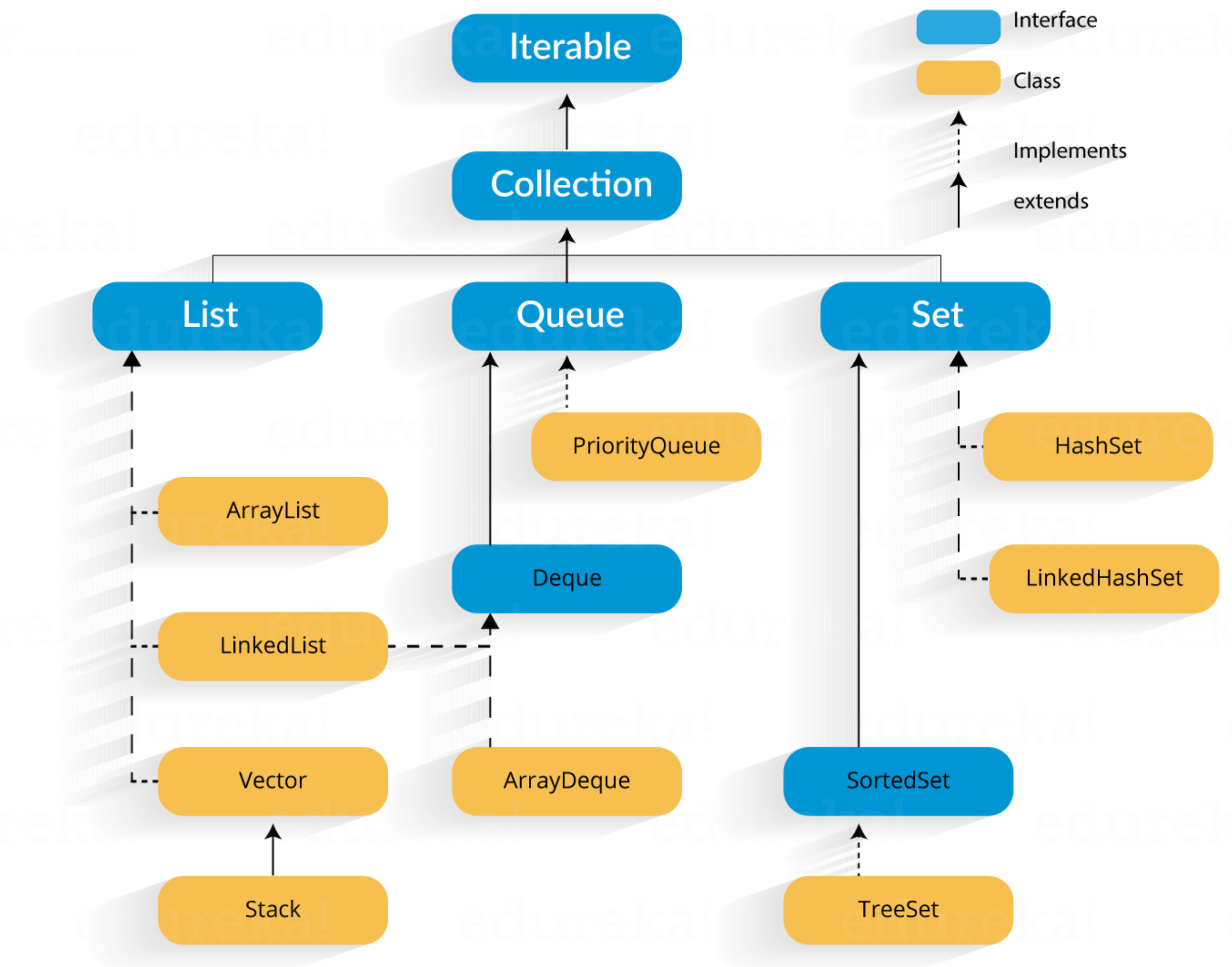
by Marina Barsky

Collection ADT



- *Collection* ADT is a general storage structure where order of elements is not necessarily maintained
- Supports addition, removal and retrieval of elements

Java Collections



Recap: Quick Search ADT

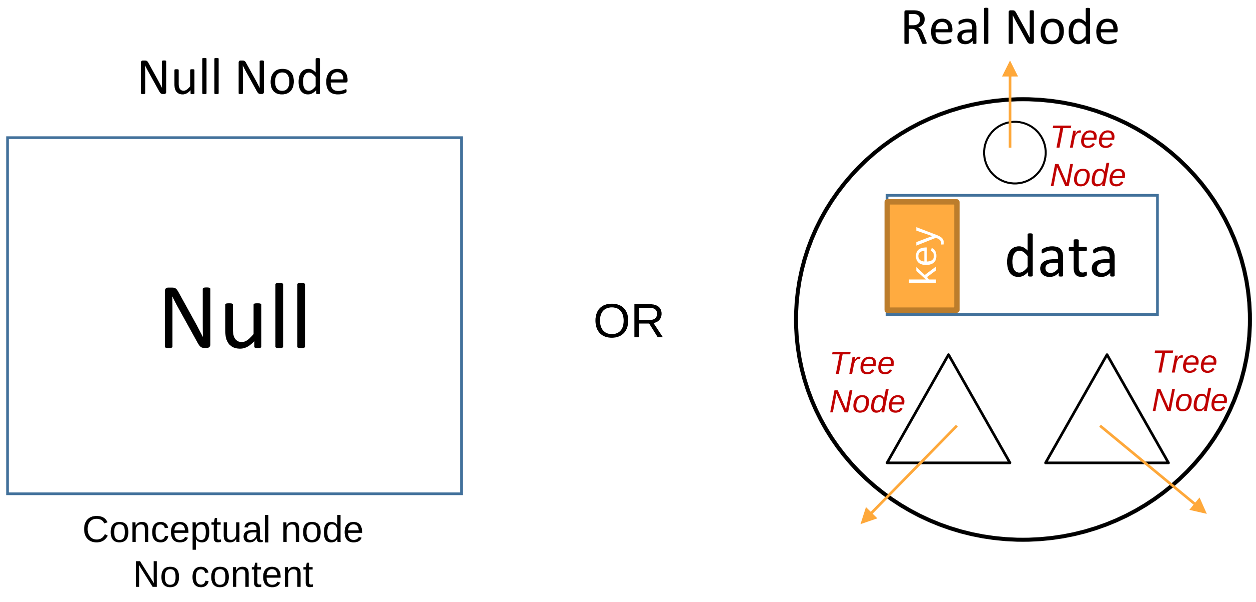
Specification

A ***Quick Search* ADT** stores a number of elements each with a *key* and supports the following operations:

- ➔ ***Search(x)***: returns the element with the key=*x*
- ➔ ***Range(lo, hi)***: returns all elements with keys between *lo* and *hi*
- ➔ ***NearestNeighbor(x)***: returns an element with the key closest to *x*
- ➔ ***Insert(x)***: adds an element with key *x*
- ➔ ***Remove(x)***: removes the element with key *x*

Recap: binary Tree can be defined by a single Tree Node variable

Tree Node root stores reference to:



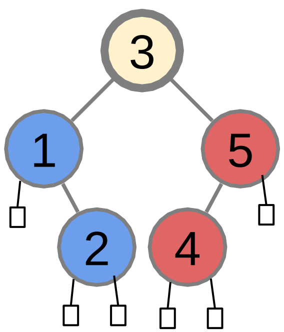
Every real *Tree Node* has exactly two children

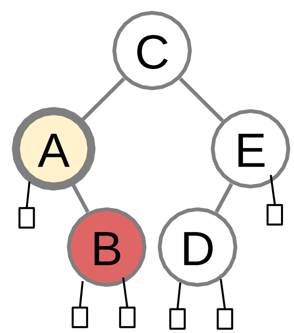
Each child is a Tree Node: Null node or Real node

Binary Search Tree

Definition

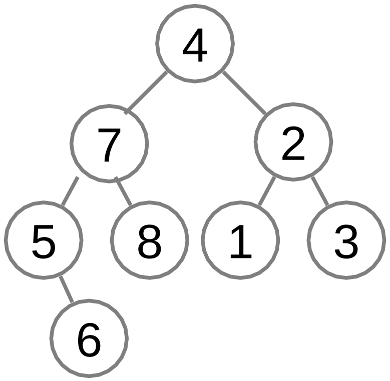
Binary search tree is a binary tree with the following property:
for each node with key x, all the real nodes in its **left subtree** have keys **smaller than** x, and all the keys in its **right subtree** are **greater* then** x.



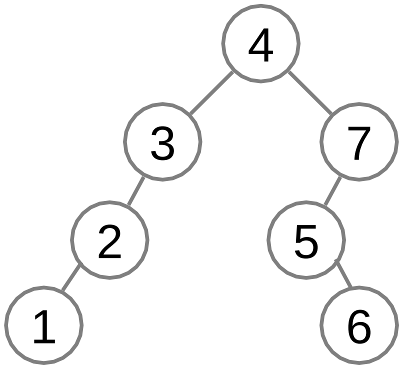


*To simplify the discussion we will assume that all keys are unique: there are no equal keys

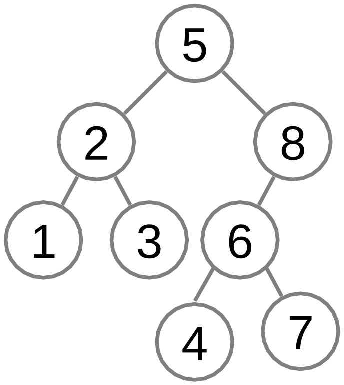
Which one is a Binary Search Tree?



A



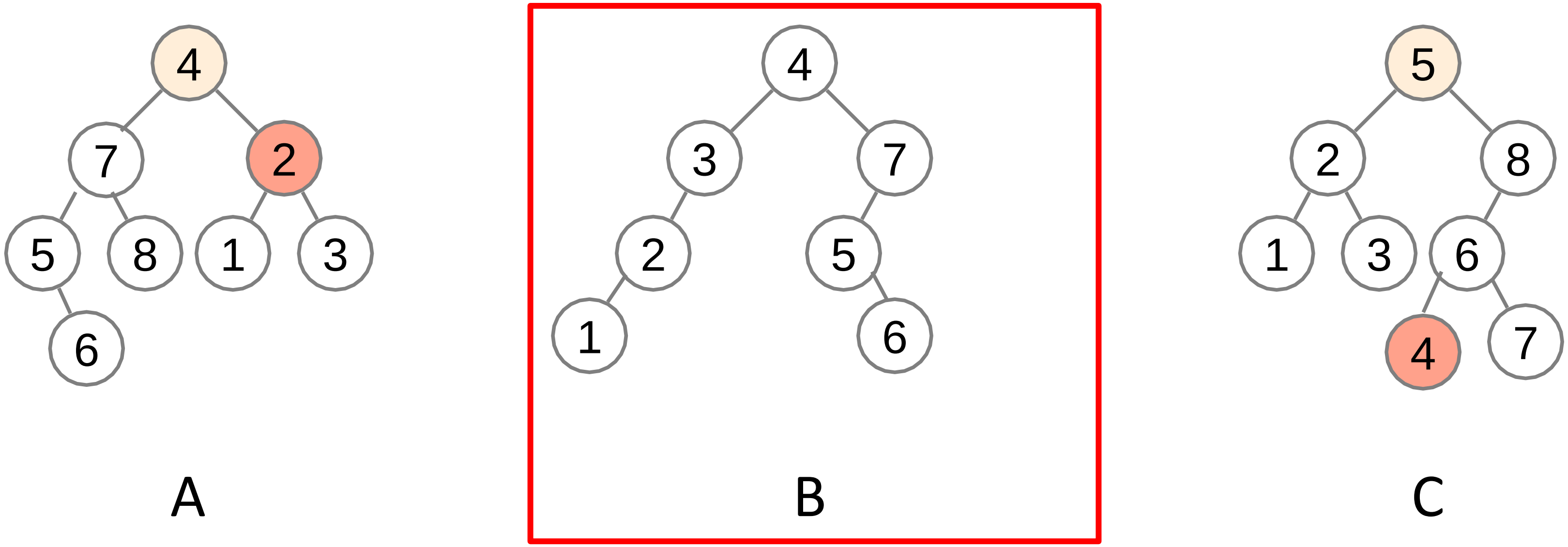
B



C

D. None of the above

Which one is a Binary Search Tree?



BST: read operations

- ➢ **Search (*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)

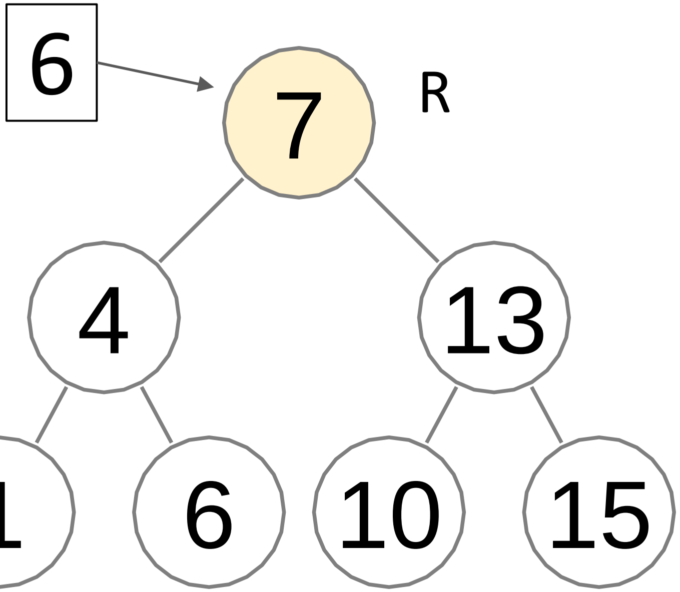
All these operations do not modify the tree

Algorithm ***Search***

Input: Key k, Tree Node R of BST

Output: The node with key k

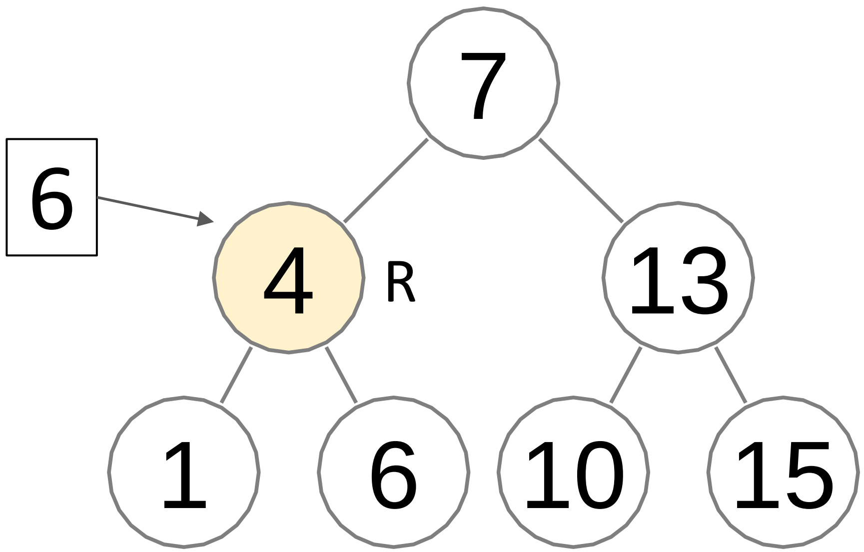
Example: search (6, node R)



6 < 7

Left child of 7 becomes R

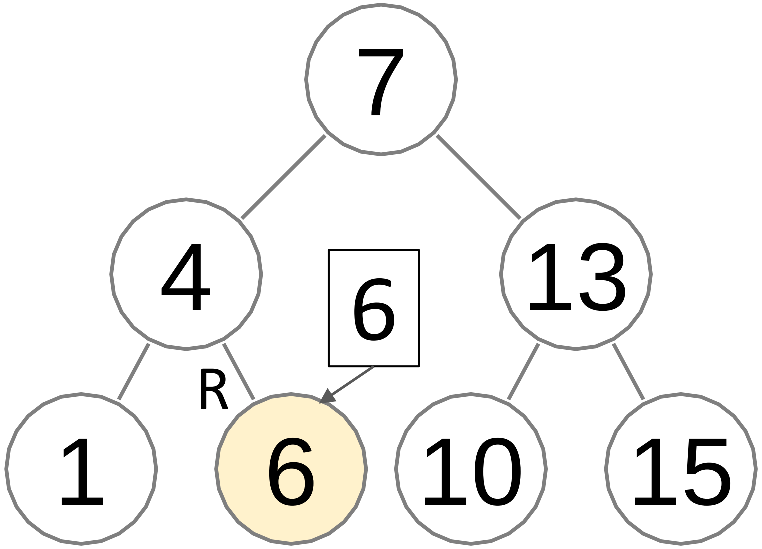
Example: search (6, node R)



6 > 4

Right child of 4 becomes R

Example: search (6, node R)



6 = 6

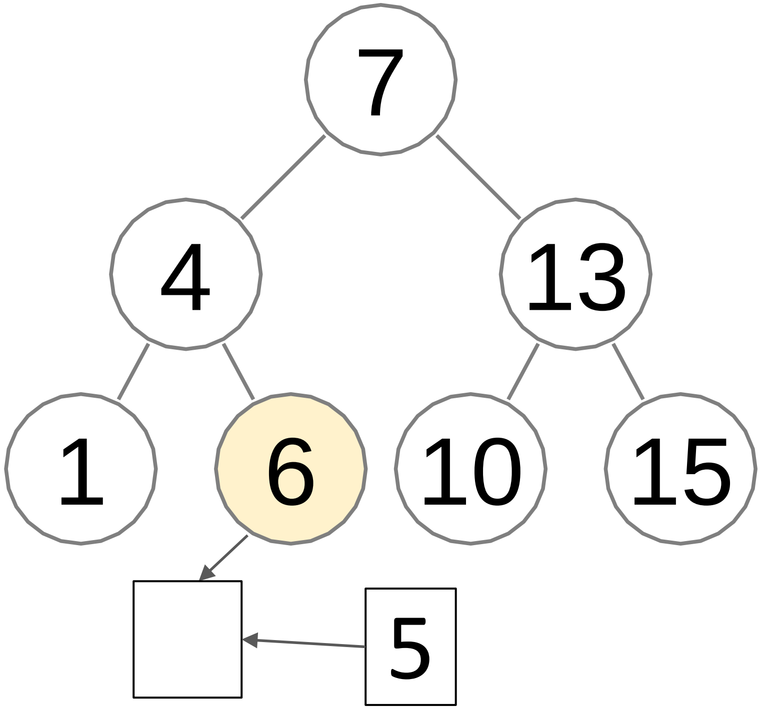
Return node R

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

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

Recursive algorithms are common and are easier to design that the corresponding non-recursive algorithms

Example: search (5, R)



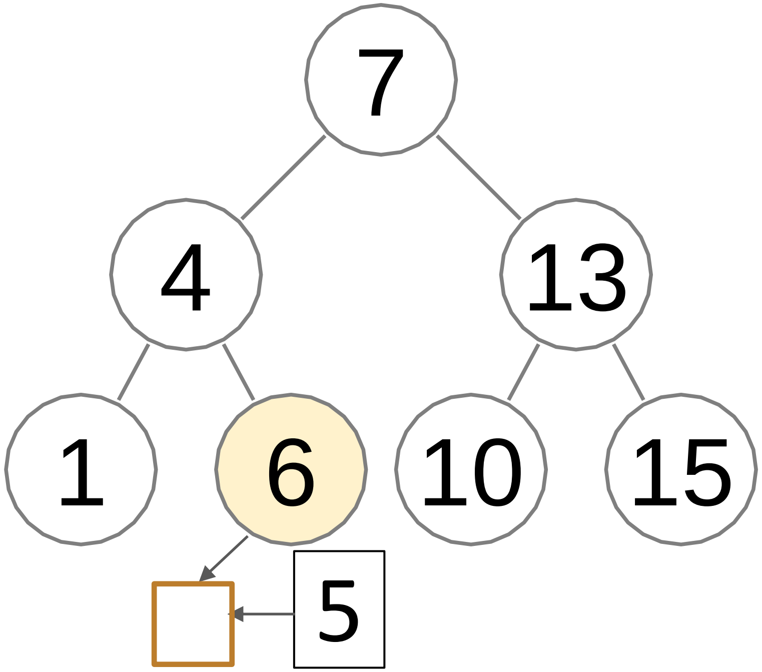
Missing key: return Null Node

Updated for the case of **missing key**

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)
```

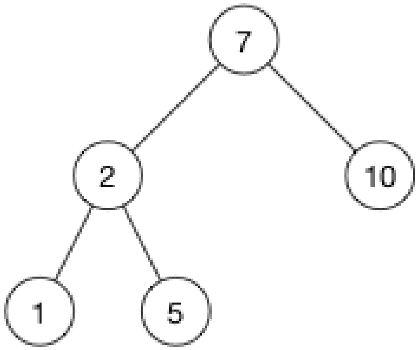
Missing key: search(5, R)



Note: If your search ended with the Null Node, this is the the place in the tree where *k* would fit.

Next in order

- BST represents the order of keys used for Binary Search
- In-order traversal of BST gets the keys in sorted order



In-order traversal:
1 2 5 7 10

What is the next after 5?

- Can we find the next key in the sorted sequence of keys without explicitly recovering the sorted sequence?

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

Algorithm ***Successor***

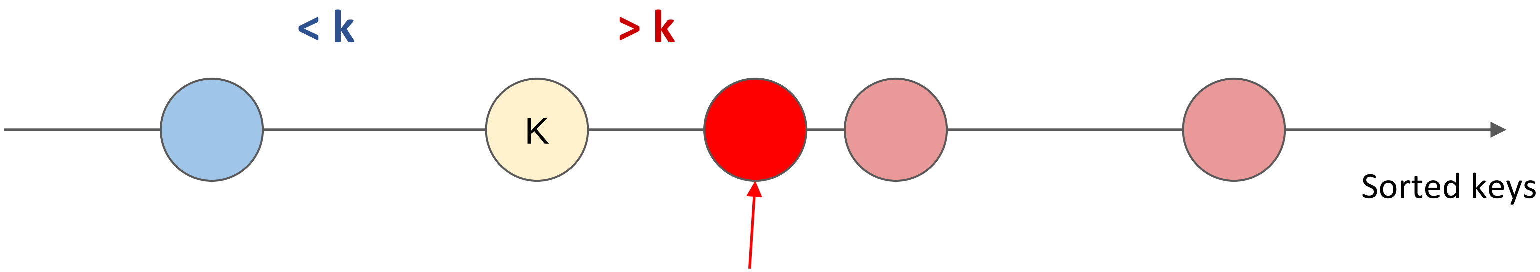
Input: key *k*

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

Algorithm ***Predecessor***

Input: key *k*

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



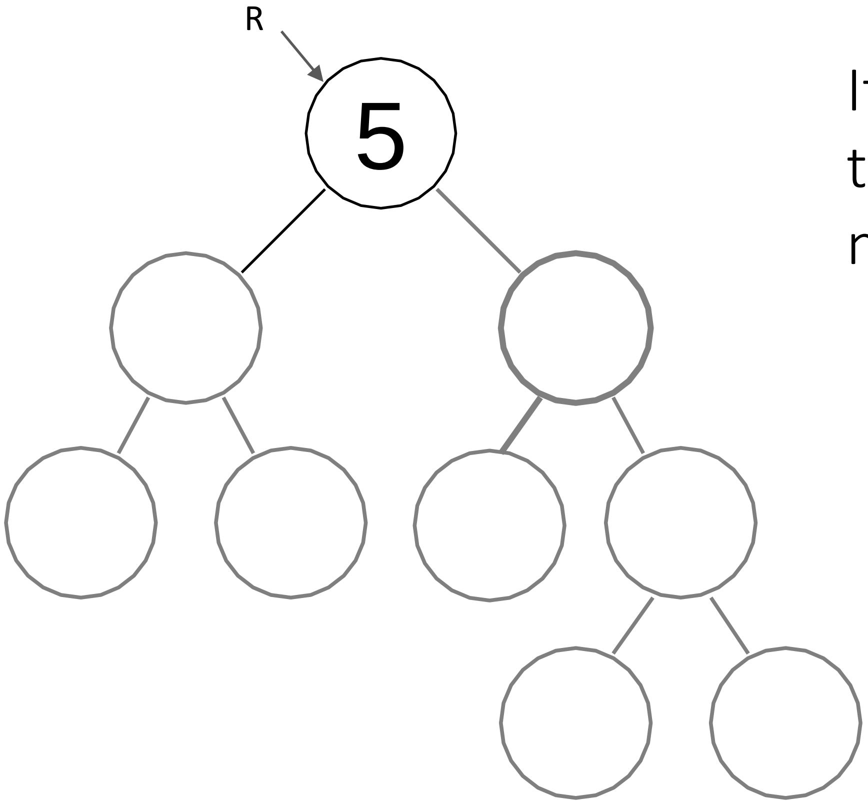
Algorithm ***Successor***

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
- To solve this we first need an algorithm for finding min key in a given tree: ***getMin***

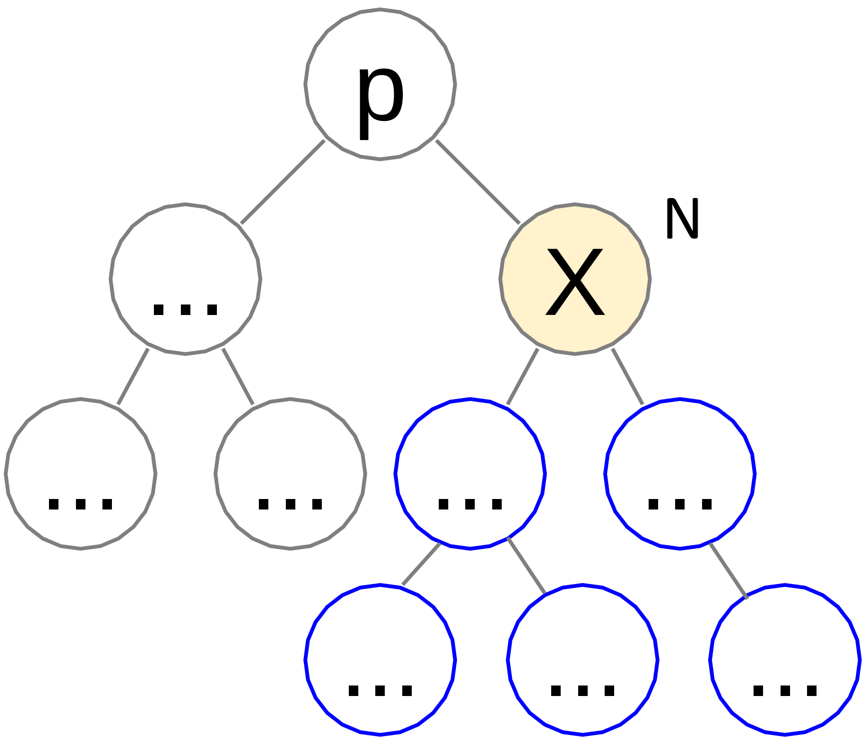
In search for *min*



If we are currently at the root R of the BST, where can we find the node with the minimum key?

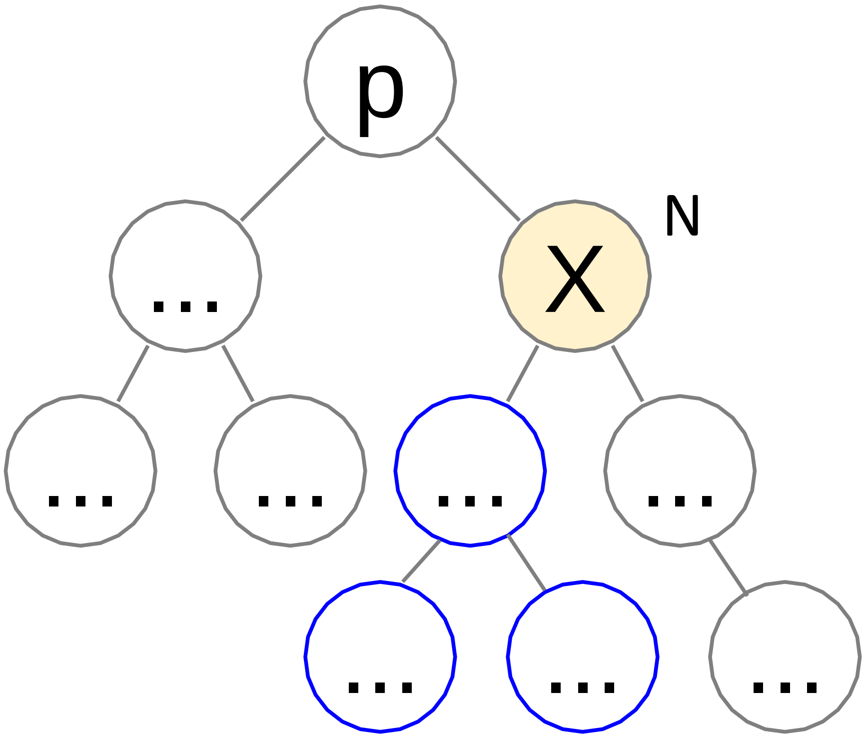
A. In the **right** subtree of R
B. In the **left** subtree of R
C. The *min* can be in **either right or left** subtree: depending on the tree

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



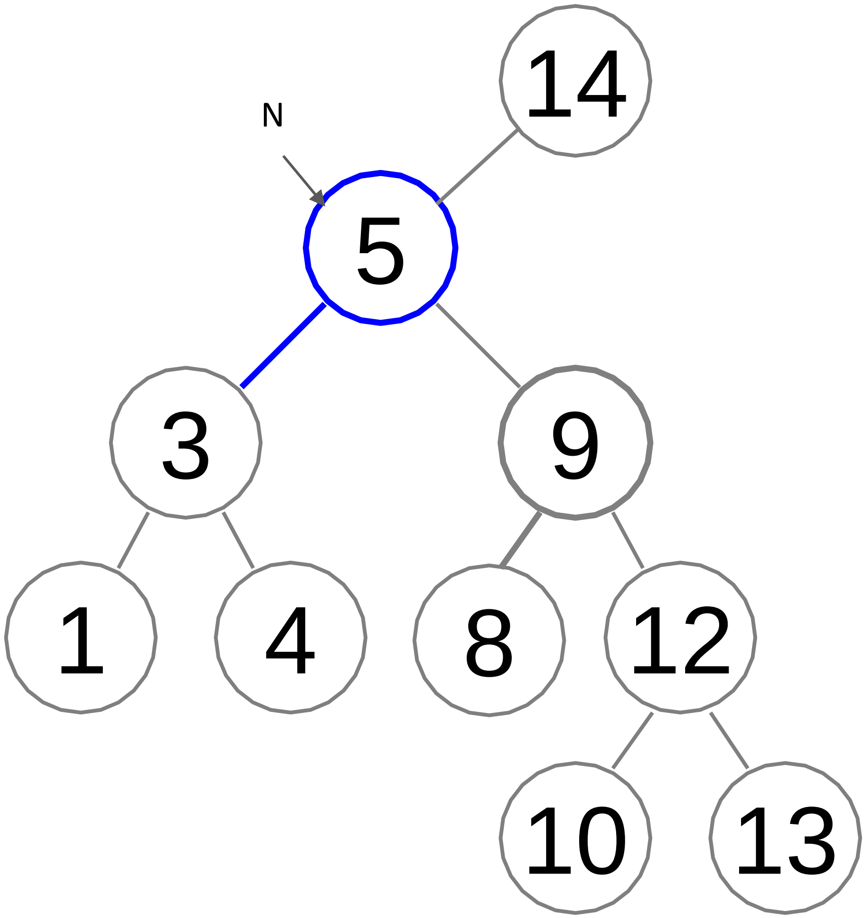
- We want the node with the smallest key in a subtree rooted at *N*

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



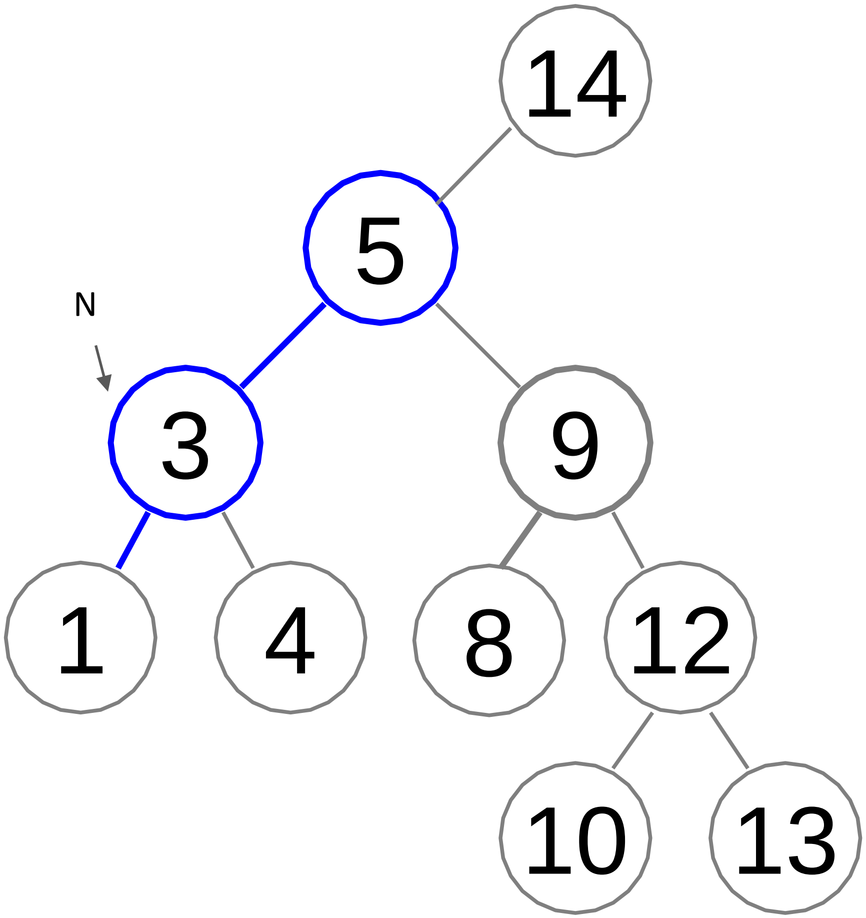
- ➢ We want the node with the smallest key in a subtree rooted at *N*
- ➢ Among all descendants of node *N* the only keys that are < *X* are in the left subtree of N

Example: *getMin* (N)



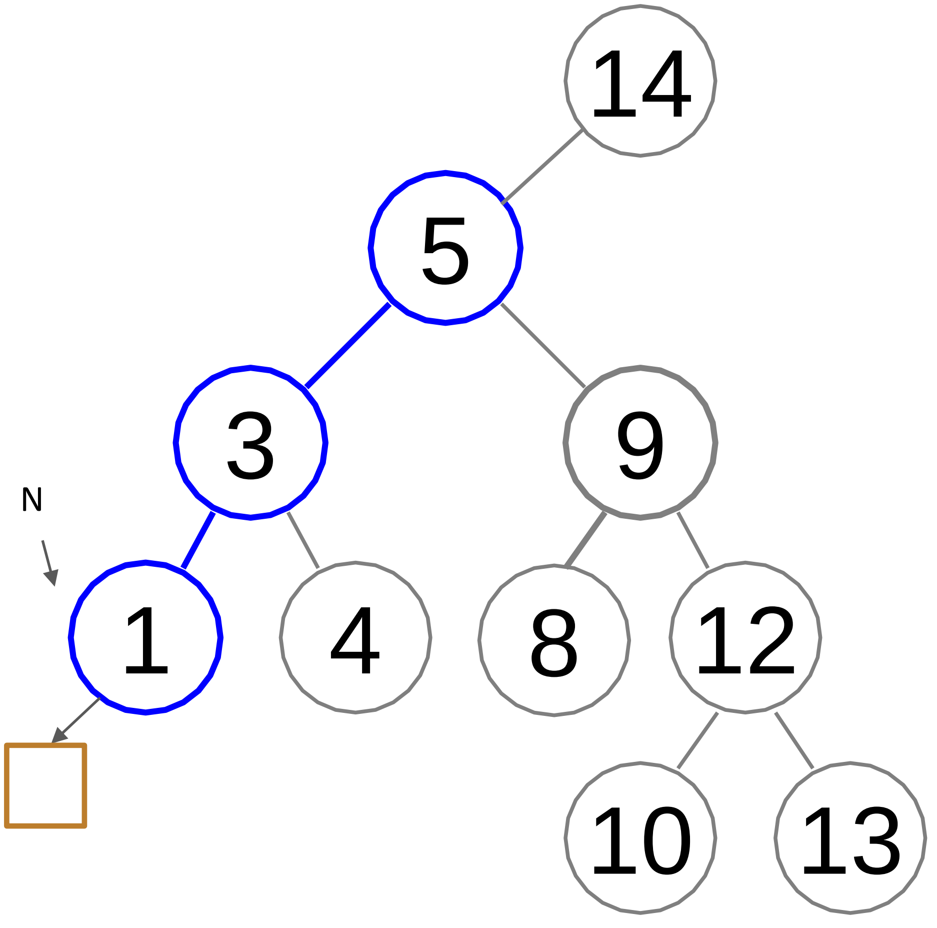
- ➔ Does node N have left child?
  Yes → there is a key smaller than 5
- ➔ Set N to be the left child and ask the same question (recursion!)

Example: *getMin* (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: *getMin* (N)



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

➔ N's key is the min

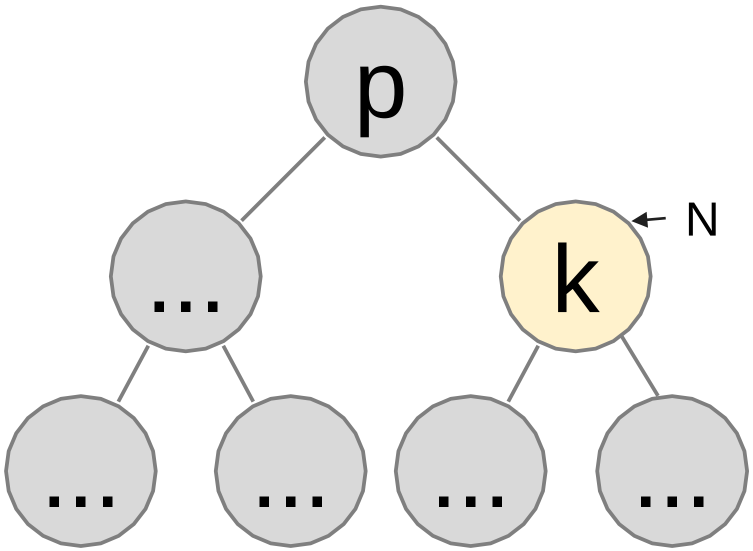
Follow the leftmost path in the tree - until N's left child becomes Null

Algorithm *getMin* (*N*)

```
if N is Null:
        ERROR: empty tree
if N.Left is Null:
        return N
else:
        return getMin (N.Left)
```

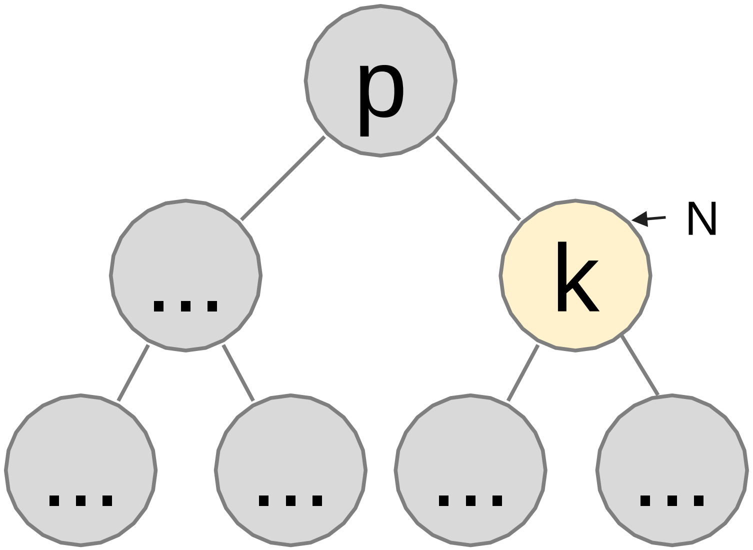
Successor (k)

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



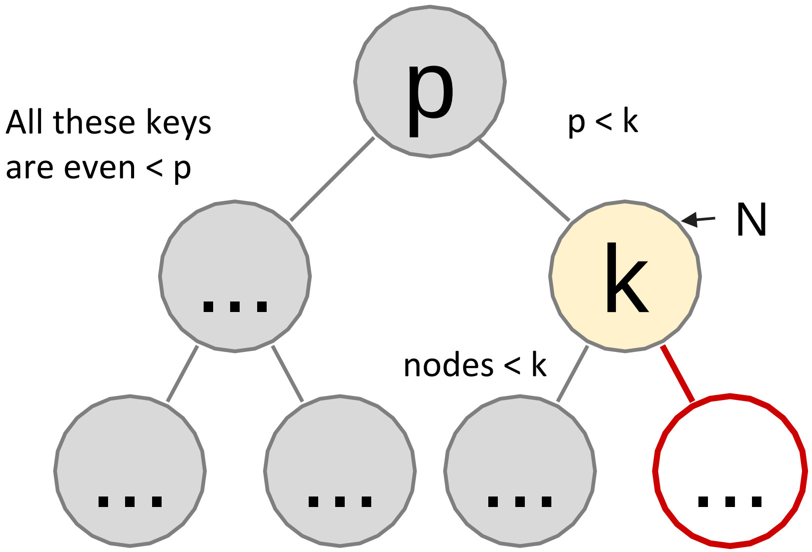
In search for *Successor* (k)

If node N with key k is the right child of its parent, we should search for its successor:



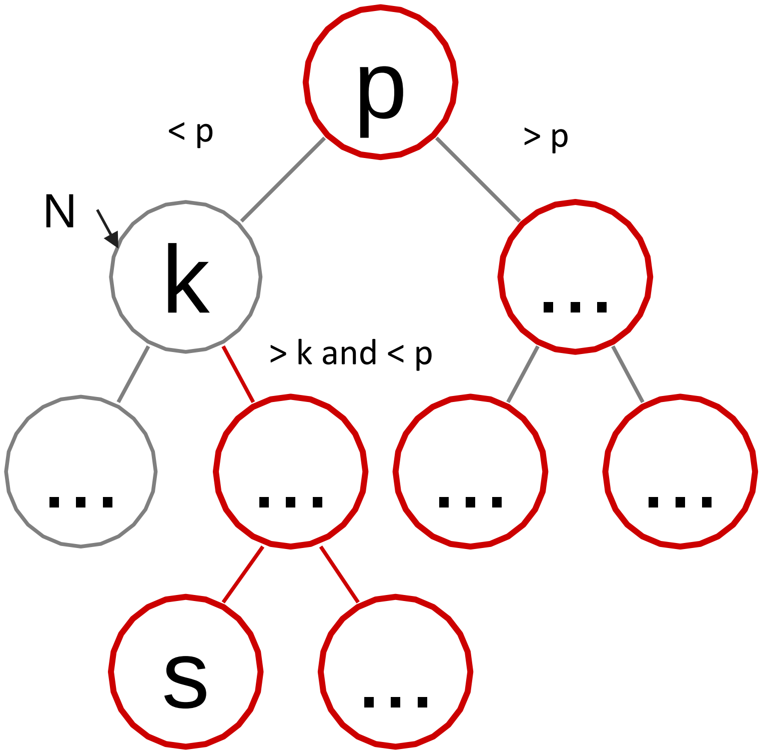
A. In the **right** subtree of N
B. In the **left** subtree of N
C. In the **left subtree** of the N's **parent**
D. None of the above (somewhere else)

Case 1A: *N* has right child and is by itself a right child of its parent



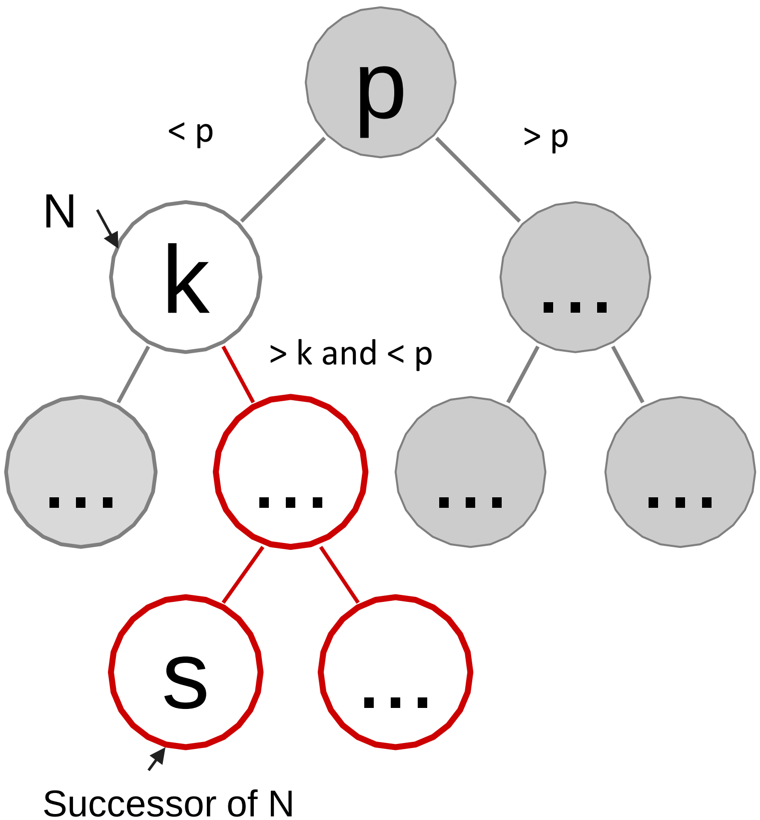
- In this situation all keys > *k* are in the right subtree of *N*

Case 1B: Node N has the right child, but N is a left child of its parent P 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*

Combined Case 1: Node N **has the right child**

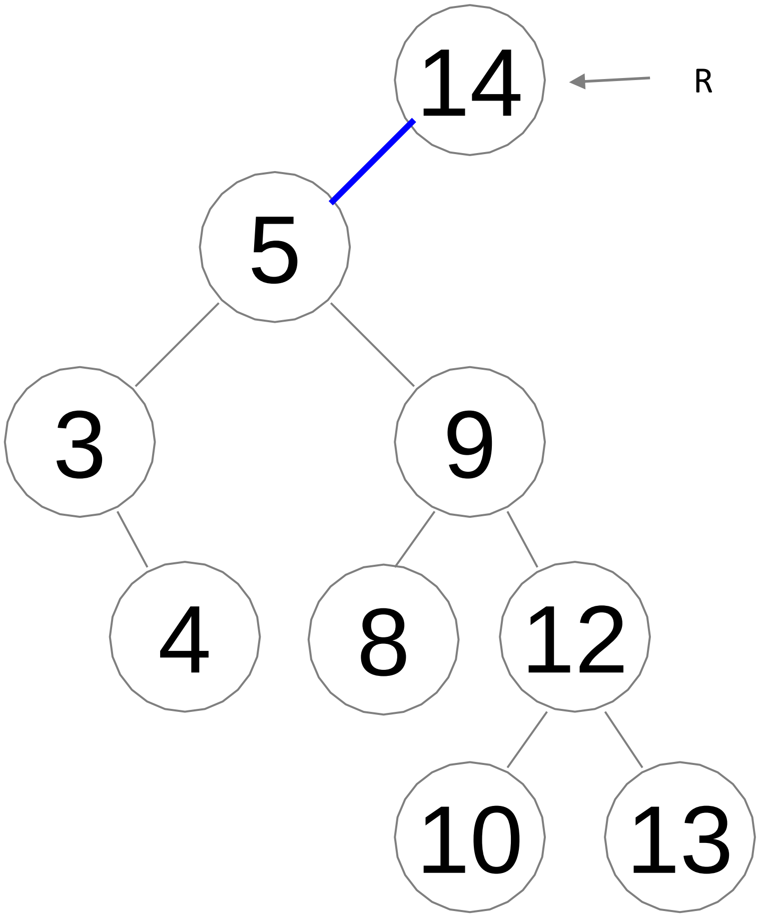


- The goal then becomes to find the smallest among all the keys in the right subtree of *N*
- Use *getMin* (*N*.right)

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

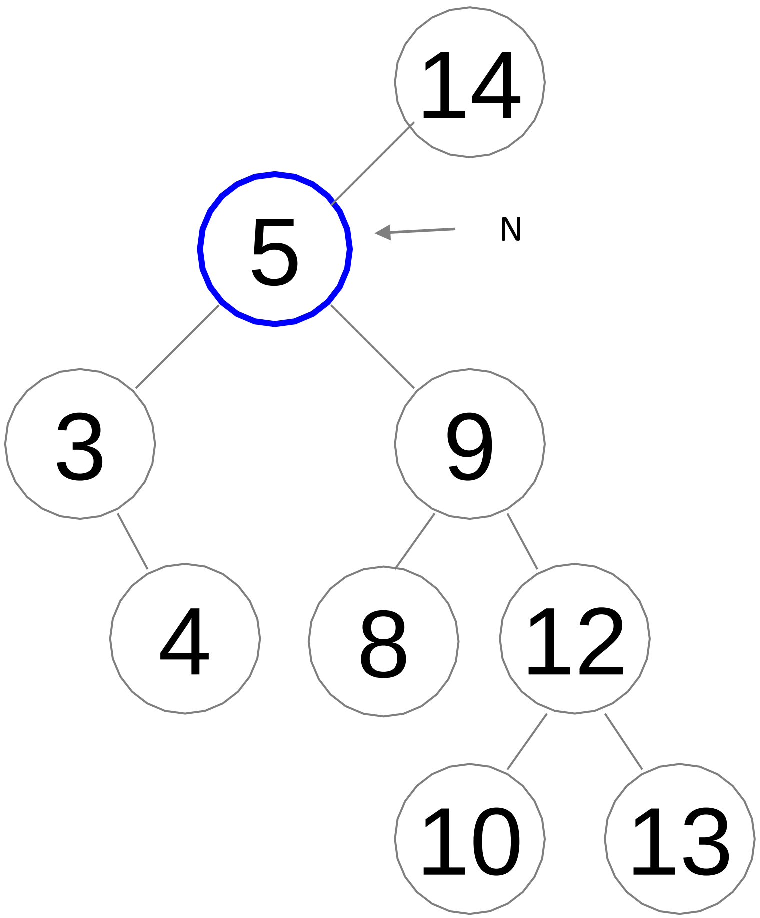
```
if R.Key = k : # found N
      if R.Right != Null :
            return getMin(R.Right)
      ...
if k < R.Key: # continue searching for N
   return Successor (k, R.Left)
      ...
if k > R.Key : # continue searching for N
   return Successor (k, R.Right)
     ...
```

Example: successor (5, R)



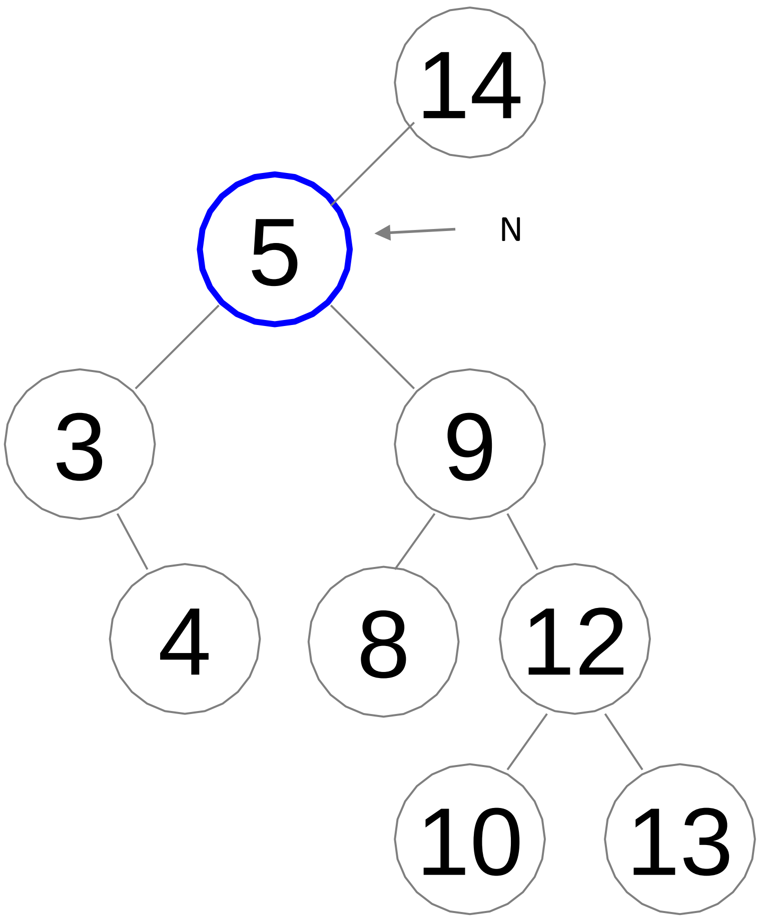
➔ Follow the left subtree:
$5 < 14$

Example: successor (5, R)



- ➔ Follow the left subtree: 5 < 14
- ➔ Found 5

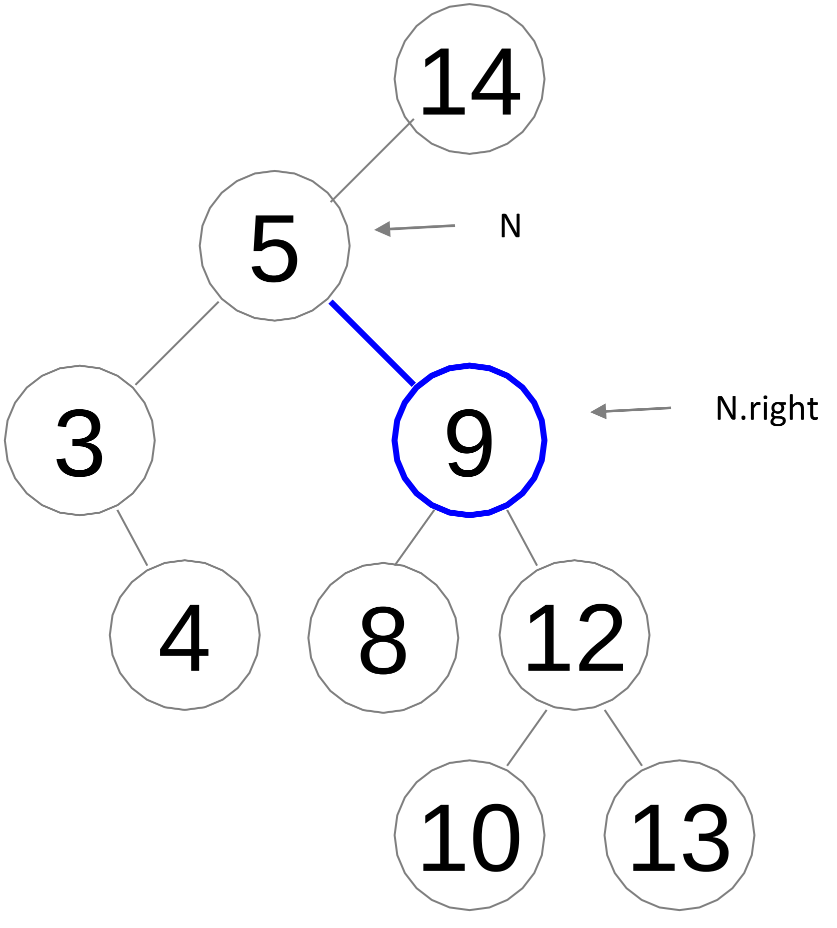
Example: successor (5, R)



- ➔ Follow the left subtree: 5 < 14
- ➔ Found 5

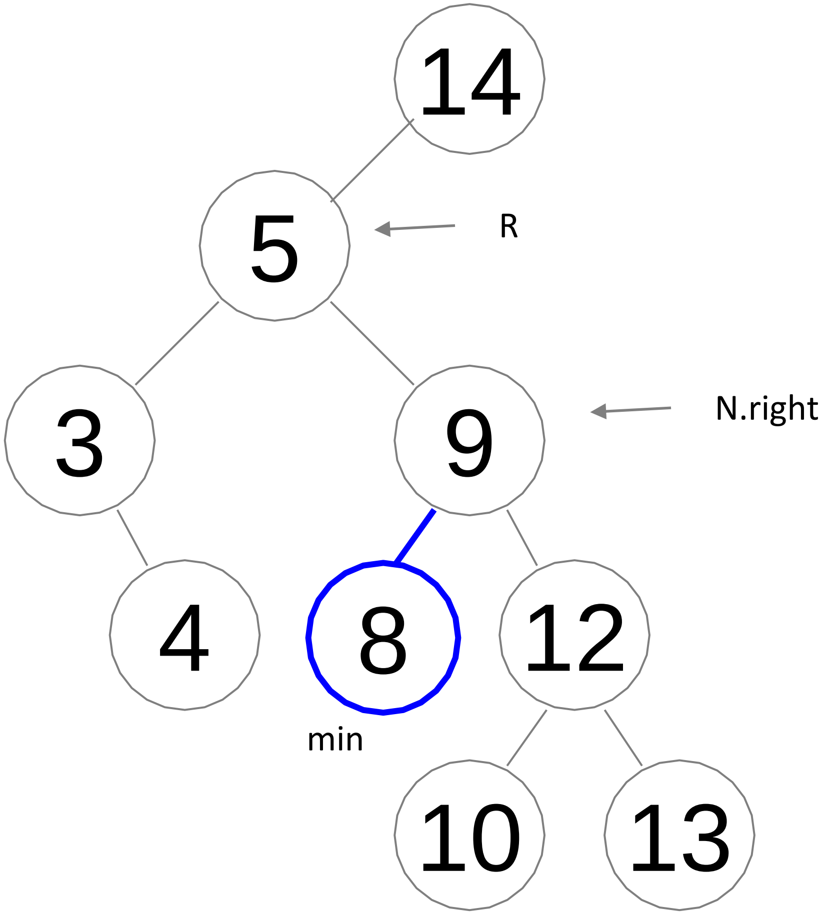
What is successor of 5?

Example: successor (5, R)



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

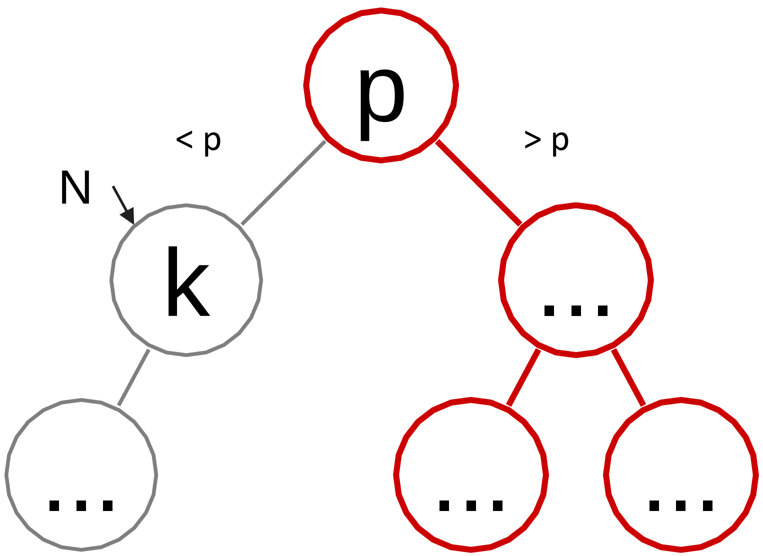
Example: successor (5, R)



- ➔ 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, but it is by itself in the left subtree of some parent node P



- 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 time when we took the left turn in the search for *N*

Successor – initially Null

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

```
if R.Key = k : # found N
    if R.Right != Null :
        return getMin(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

Successor – initially Null

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

```
if R.Key = k : # found N
    if R.Right != Null:
        return getMin(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)
```

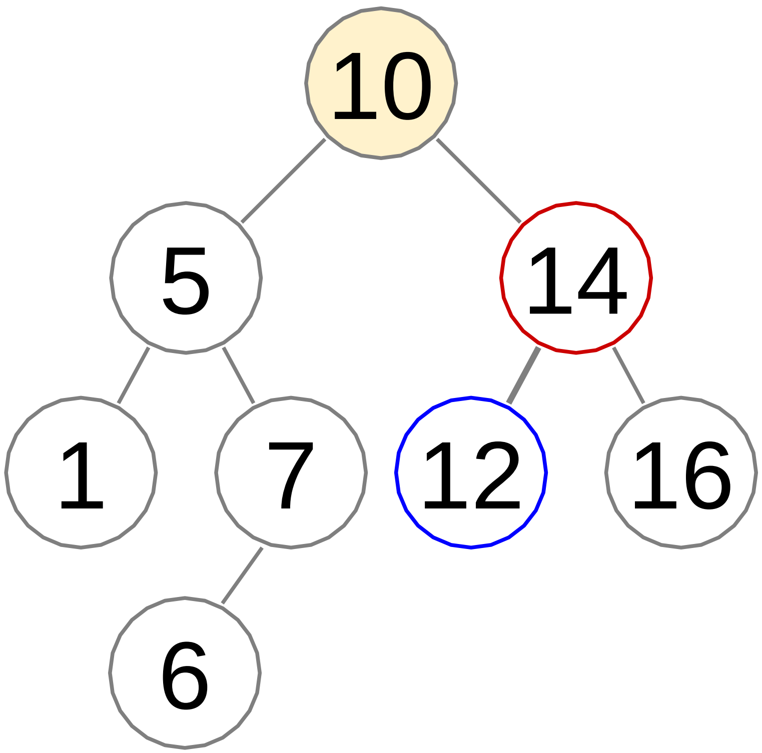
What happens if *k* is not in the tree?

Can we find the next value to k?

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

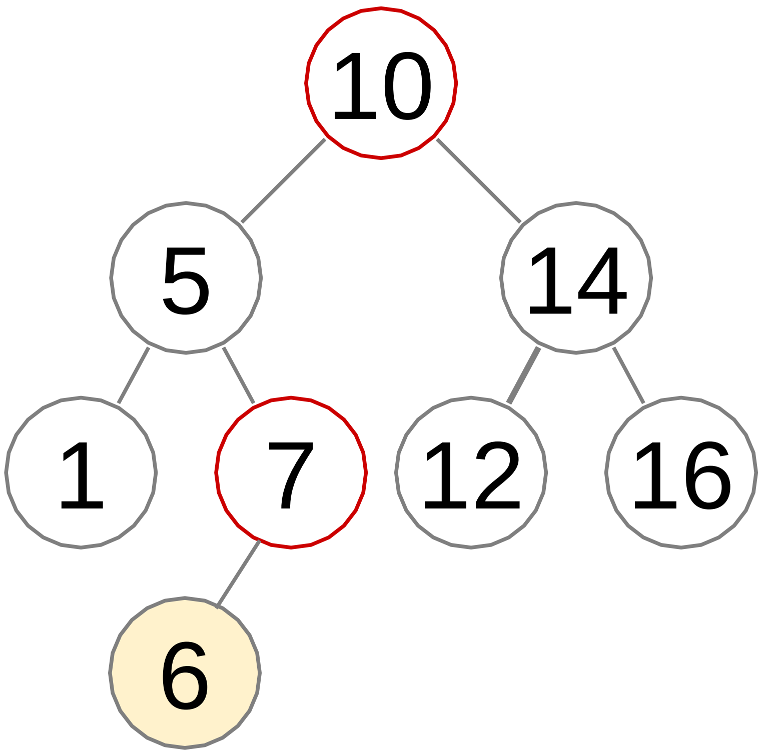
```
if R = Null: # k is not in the tree
    return S # Null node has no right child

if R.Key = k : # found N
    if R.Right != Null:
        return getMin (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)
```

Example: *Successor* (10, R)



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

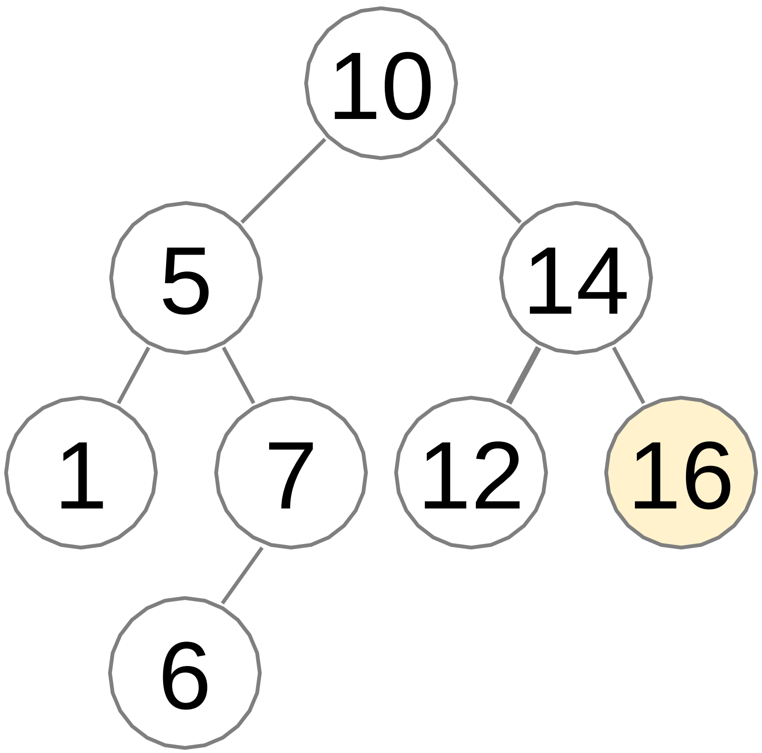
Example: *Successor* (6, R)



- ➔ 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

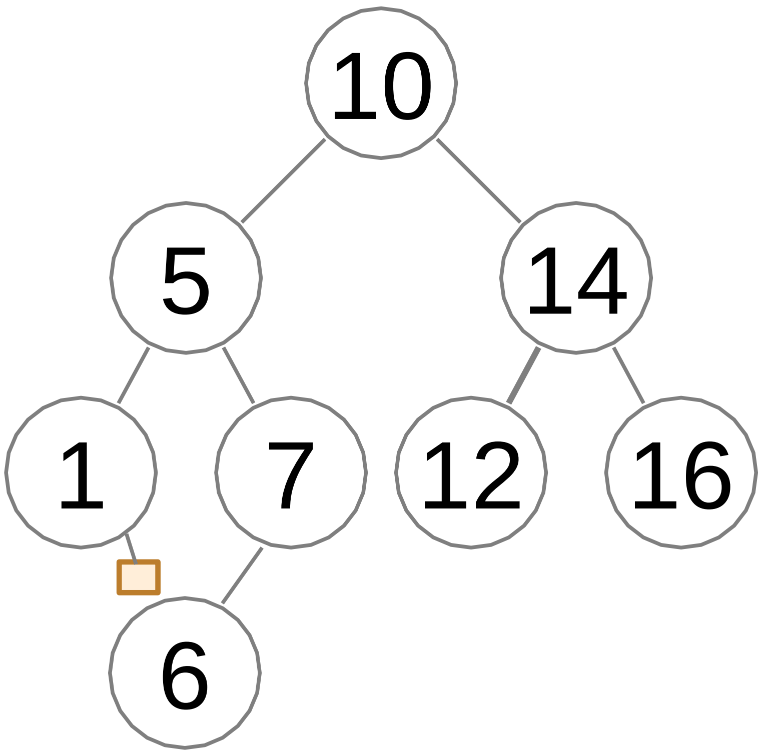
Example: *Successor* (16, R)



- ➔ 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

Example: *Successor* (3, R)



- ➔ While searching for 3: we took the left turn first at 10 then at 5
- ➔ We did not find 3 but found a null node instead
- ➔ We return the next larger number:

  Successor (3) → 5

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

Algorithm *Range*

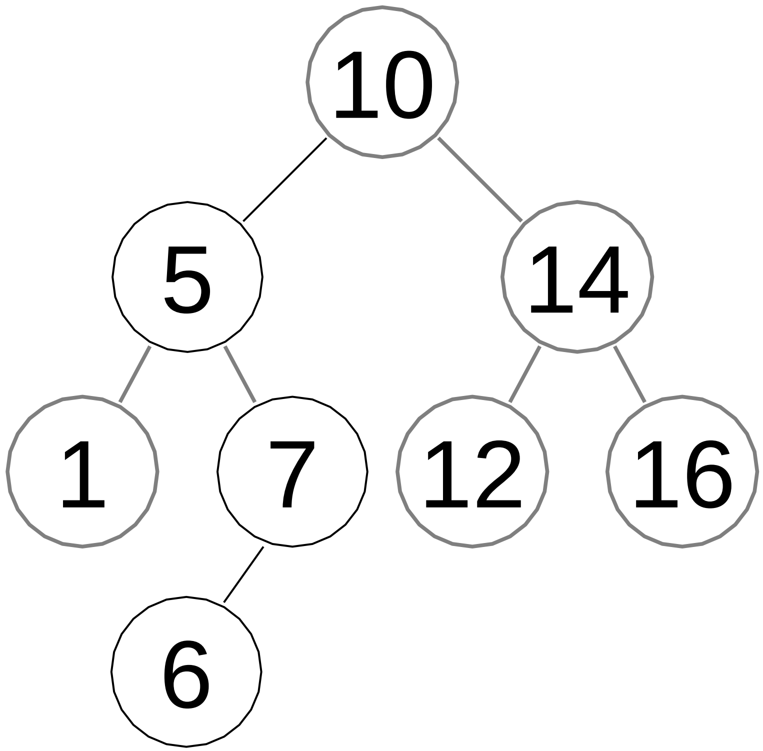
Input: Keys *lo*, *hi*, root *R*

Output: A list of nodes with keys between *lo* and *hi*

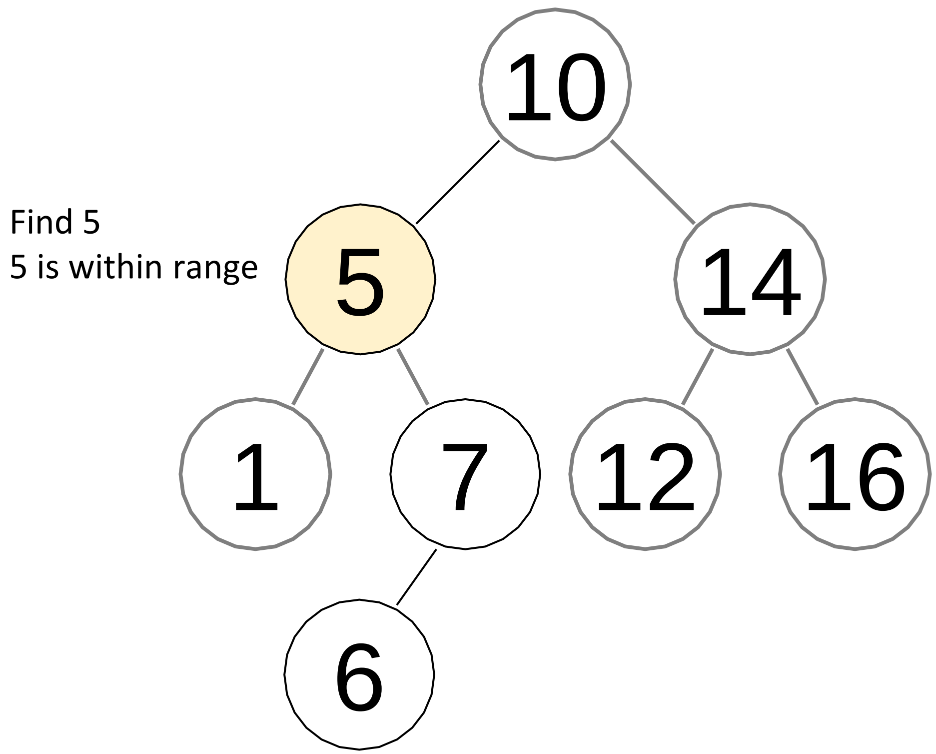
Algorithm RangeSearch (lo , hi , R)

```
L ← empty list

N ← Successor (lo , R)
while N is not Null and N.Key ≤ hi
    L ← L + N
    N ← Successor (N.Key, R, Null)
return L
```

Example: range search (5, 13)

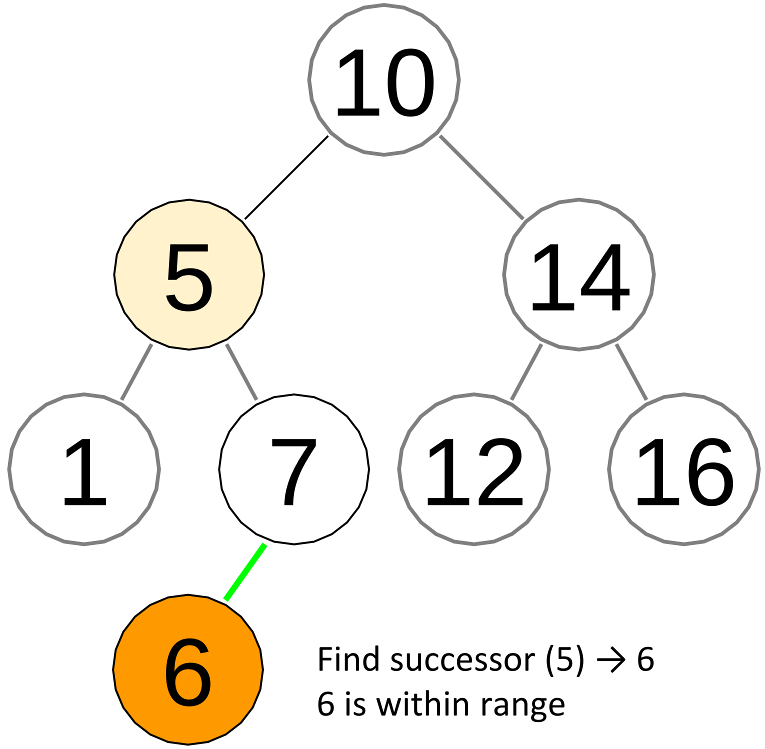


Example: range search (5, 13)



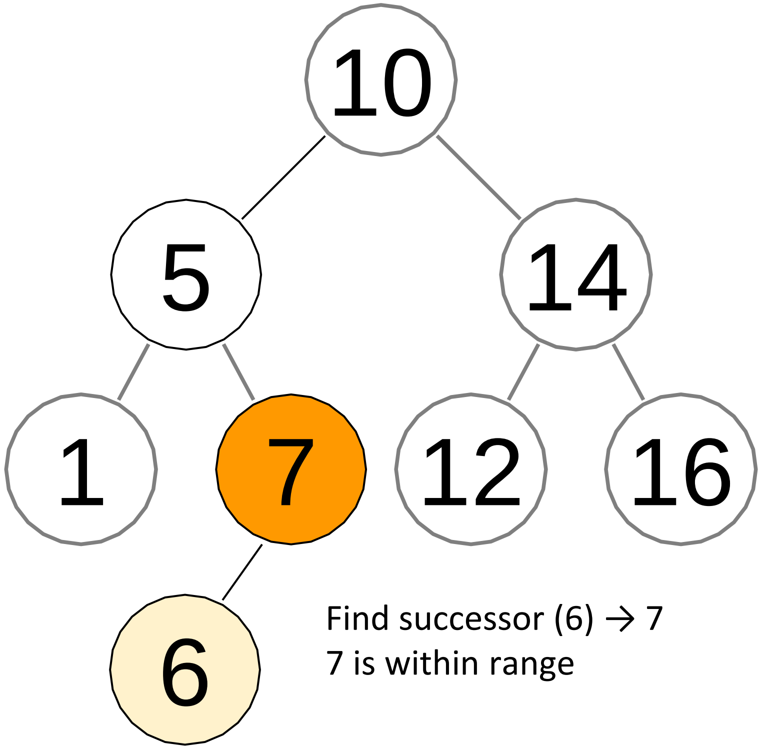
Result: 5

Example: range search (5, 13)



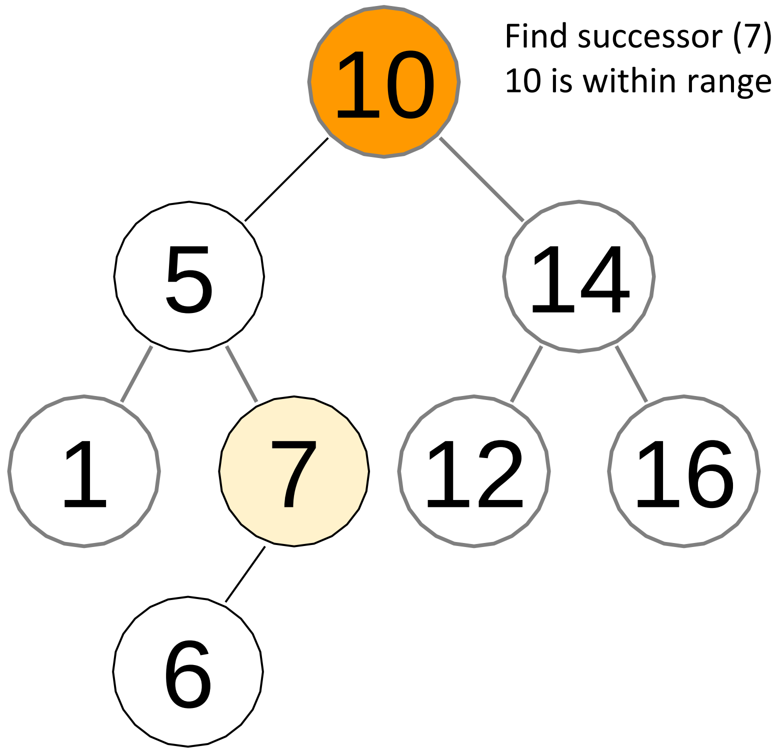
Result: 5, 6

Example: range search (5, 13)



Result: 5, 6, 7

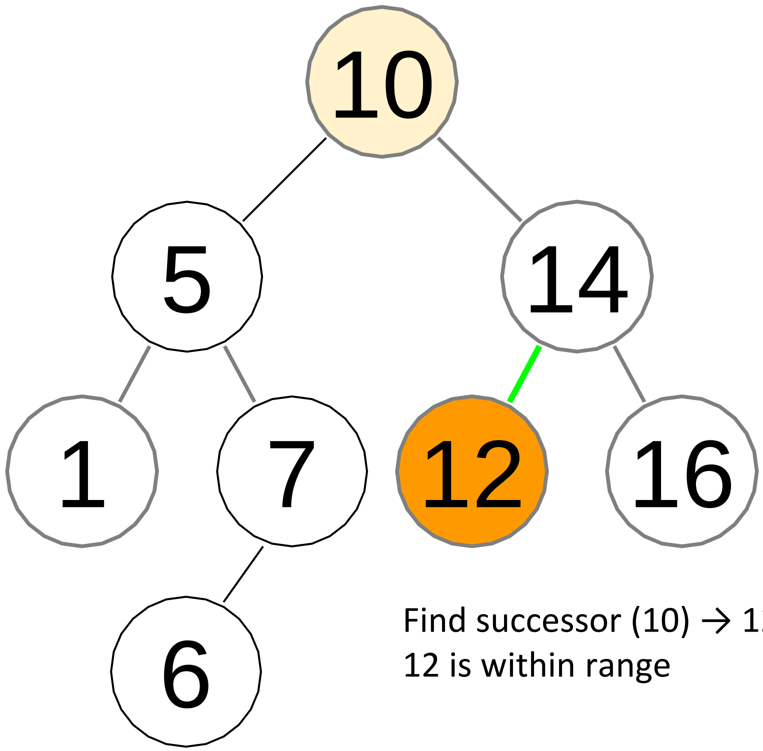
Example: range search (5, 13)



Find successor (7) → 10
10 is within range

Result: 5, 6, 7, 10

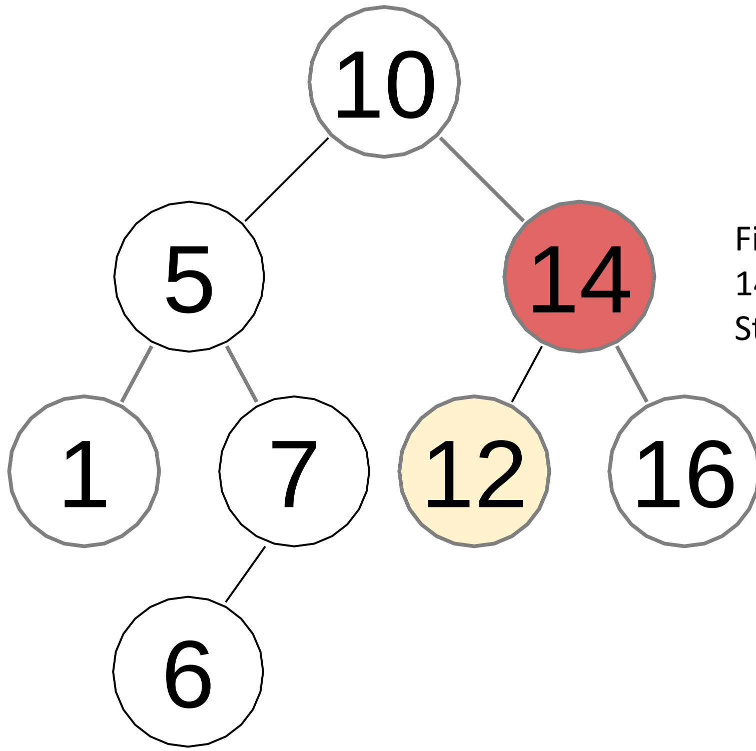
Example: range search (5, 13)



Find successor (10) → 12
12 is within range

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

Algorithm *Predecessor* (*k*, *R*...)

```
if R.Key = k : # found N
      if R.         != Null :
            return           (R.        )
      ...
if k < R.Key :
      ...
      return Predecessor (k, R.Left...)
if k > R.Key:
      ...
      return Predecessor (k, R.Right...)
```

Fill in blanks:

A. right getMin right
B. left getMin left
C. left getMax left
D. left Predecessor left
E. None of the above