Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Introduction

Search Tree

Search tree is an ADT that supports the following 7 operations on a set S of elements:

$\textsc{Search}(key)$, $\textsc{Insert}(x)$, $\textsc{Delete}(x)$, $\textsc{Minimum}()$, $\textsc{Maximum}()$

$\textsc{Successor}(x)$: returns the element whose key is the next larger after x in S, or NULL if x is the maximum key.

$\textsc{Predecessor}(x)$: returns the the element whose key is the next smaller element after x in S, or NULL if x is the minimum key.

Introduction

- Example: $S = \{2, 3,5, 7, 11, 13, 17\}$, then:

$$\textsc{Successor}(11) = 13$$
$$\textsc{Predecessor}(13) = 11$$
$$\textsc{Successor}(17) = \text{NULL}$$
$$\textsc{Predecessor}(2) = \text{NULL}$$

Introduction

$\textsc{Search}(key)$
$\textsc{Insert}(x)$
$\textsc{Delete}(x)$

Dictionary

$\textsc{Insert}(x)$
$\textsc{Delete}(x)$
$\textsc{Minimum}()/\textsc{Maximum}()$

Min/Max Priority Queue

$\textsc{Minimum}()$
$\textsc{Successor}(x)$

Can be used for sorting

Introduction

- An efficient implementation of a search tree can be used as a **dictionary**, **priority queue**, and also as a data structure to **sort** the elements dynamically.
- Ideally, we want all these operations to be done in $\boldsymbol{O(\log n)}$ time.
- But let's first look at the previous data structures and see why the fail to achieve this.

Previous data structures

$\textsc{Search}(key)$

$\textsc{Insert}(x)$

$\textsc{Delete}(x)$

$\textsc{Minimum}()$

$\textsc{Maximum}()$

$\textsc{Successor}(x)$

$\textsc{Predecessor}(x)$

We assume that a pointer or the index to the element x, is given

Previous data structures

Unsorted list/array

$\textsc{Search}(key)$
$\textsc{Insert}(x)$
$\textsc{Delete}(x)$
$\textsc{Minimum}()$
$\textsc{Maximum}()$
$\textsc{Successor}(x)$
$\textsc{Predecessor}(x)$

Previous data structures

Unsorted list/array

$O(n)$	$\textsc{Search}(key)$
$O(1)$	$\textsc{Insert}(x)$
$O(1)$ for list, $O(n)$ for array	$\textsc{Delete}(x)$
$O(n)$	$\textsc{Minimum}()$
$O(n)$	$\textsc{Maximum}()$
$O(n)$	$\textsc{Successor}(x)$
$O(n)$	$\textsc{Predecessor}(x)$

Previous data structures

$O(n)$	$\text{Search}(key)$
$O(1)$	$\text{Insert}(x)$
$O(1)$ for list, $O(n)$ for array	$\text{Delete}(x)$
$O(n)$	$\text{Minimum}()$
$O(n)$	$\text{Maximum}()$
$O(n)$	$\text{Successor}(x)$
$O(n)$	$\text{Predecessor}(x)$

Unsorted list/array

Note 1: Insert in a list can be done at the head. And insert in array can be done at the last empty index. So, insert only takes O(1) time. We assume that our array has enough capacity to include all elements of S.

Previous data structures

$O(n)$	$\textsc{Search}(key)$
$O(1)$	$\textsc{Insert}(x)$
$O(1)$ for list, $O(n)$ for array	$\textsc{Delete}(x)$
$O(n)$	$\textsc{Minimum}()$
$O(n)$	$\textsc{Maximum}()$
$O(n)$	$\textsc{Successor}(x)$
$O(n)$	$\textsc{Predecessor}(x)$

Unsorted list/array

Note 2: Delete takes x as the argument instead of key. So, we can assume that we have a pointer the element we want to delete (in case of a list), or we have the index of it (in case of an array). After deleting x from the array we have to shift other elements; hence, $O(n)$ time.

Previous data structures

Sorted array

$\textsc{Search}(key)$

$\textsc{Insert}(x)$

$\textsc{Delete}(x)$

$\textsc{Minimum}()$

$\textsc{Maximum}()$

$\textsc{Successor}(x)$

$\textsc{Predecessor}(x)$

Previous data structures

Sorted array

$O(\log n)$	$\textsc{Search}(key)$
$O(n)$	$\textsc{Insert}(x)$
$O(n)$	$\textsc{Delete}(x)$
$O(1)$	$\textsc{Minimum}()$
$O(1)$	$\textsc{Maximum}()$
$O(1)$	$\textsc{Successor}(x)$
$O(1)$	$\textsc{Predecessor}(x)$

Previous data structures

Sorted array

Note : We want to keep the array sorted after an insert or a delete, so we have to shift O(n) other elements to make room for the new element, or concatenate the two parts after an element is removed. Successor, and predecessor can be found by returning the element to the right (successor) or element to the left (predecessor).

$O(\log n)$	$\textsc{Search}(key)$
$O(n)$	$\textsc{Insert}(x)$
$O(n)$	$\textsc{Delete}(x)$
$O(1)$	$\textsc{Minimum}()$
$O(1)$	$\textsc{Maximum}()$
$O(1)$	$\textsc{Successor}(x)$
$O(1)$	$\textsc{Predecessor}(x)$

Previous data structures

Sorted list

$\textsc{Search}(key)$

$\textsc{Insert}(x)$

$\textsc{Delete}(x)$

$\textsc{Minimum}()$

$\textsc{Maximum}()$

$\textsc{Successor}(x)$

$\textsc{Predecessor}(x)$

Previous data structures

This is under the assumption that we know where to insert.

if we keep a pointer to the last element

$O(n)$	$\textsc{Search}(key)$
$O(1)$	$\textsc{Insert}(x)$
$O(1)$	$\textsc{Delete}(x)$
$O(1)$	$\textsc{Minimum}()$
$O(1)$	$\textsc{Maximum}()$
$O(1)$	$\textsc{Successor}(x)$
$O(1)$	$\textsc{Predecessor}(x)$

Previous data structures

Sorted list

Note: There is no point in using a binary search on a linked list since we can't have random access to the middle position. We should get to the middle position sequentially by traversing the list which takes linear time.

$O(n)$	$\textsc{Search}(key)$
$O(1)$	$\textsc{Insert}(x)$
$O(1)$	$\textsc{Delete}(x)$
$O(1)$	$\textsc{Minimum}()$
$O(1)$	$\textsc{Maximum}()$
$O(1)$	$\textsc{Successor}(x)$
$O(1)$	$\textsc{Predecessor}(x)$

Previous data structures

Heap

$\textsc{Search}(key)$

$\textsc{Insert}(x)$

$\textsc{Delete}(x)$

$\textsc{Minimum}()$

$\textsc{Maximum}()$

$\textsc{Successor}(x)$

$\textsc{Predecessor}(x)$

Previous data structures

Heap

Say we keep a min-heap and a max-heap simultaneously!

$O(n)$	$\textsc{Search}(key)$
$O(\log n)$	$\textsc{Insert}(x)$
$O(\log n)$	$\textsc{Delete}(x)$
$O(1)$	$\textsc{Minimum}()$
$O(1)$	$\textsc{Maximum}()$
$O(n)$	$\textsc{Successor}(x)$
$O(n)$	$\textsc{Predecessor}(x)$

Previous data structures

Hash Table

$\textsc{Search}(key)$
$\textsc{Insert}(x)$
$\textsc{Delete}(x)$
$\textsc{Minimum}()$
$\textsc{Maximum}()$
$\textsc{Successor}(x)$
$\textsc{Predecessor}(x)$

Previous data structures

Hash Table

$O(1)$	$\textsc{Search}(key)$
$O(1)$	$\textsc{Insert}(x)$
$O(1)$	$\textsc{Delete}(x)$
$O(n)$	$\textsc{Minimum}()$
$O(n)$	$\textsc{Maximum}()$
$O(n)$	$\textsc{Successor}(x)$
$O(n)$	$\textsc{Predecessor}(x)$

Idea behind BST

Combining these two

Sorted list	Sorted array	
$O(n)$	$O(\log n)$	$\textsc{Search}(key)$
$O(1)$	$O(n)$	$\textsc{Insert}(x)$
$O(1)$	$O(n)$	$\textsc{Delete}(x)$
$O(1)$	$O(1)$	$\textsc{Minimum}()$
$O(1)$	$O(1)$	$\textsc{Maximum}()$
$O(1)$	$O(1)$	$\textsc{Successor}(x)$
$O(1)$	$O(1)$	$\textsc{Predecessor}(x)$

Idea behind BST

- The useful property in a sorted array is that if you look at an index, everything to the right is larger, and every thing to left is smaller.
- This enables us to do the binary search in a divide-and-conquer paradigm and take only $O(\log n)$ time.

1															16
2	**3**	**5**	**7**	**11**	**13**	**14**	**17**	**23**	**28**	**31**	**39**	**55**	**90**	**97**	**99**

Binary Search Tree

- BST is a binary tree such that each node x has the following fields, and has the **BST property**.

***x*.*key*:** *x's key*

***x*.*left*:** *pointer to the root of x's left subtree*

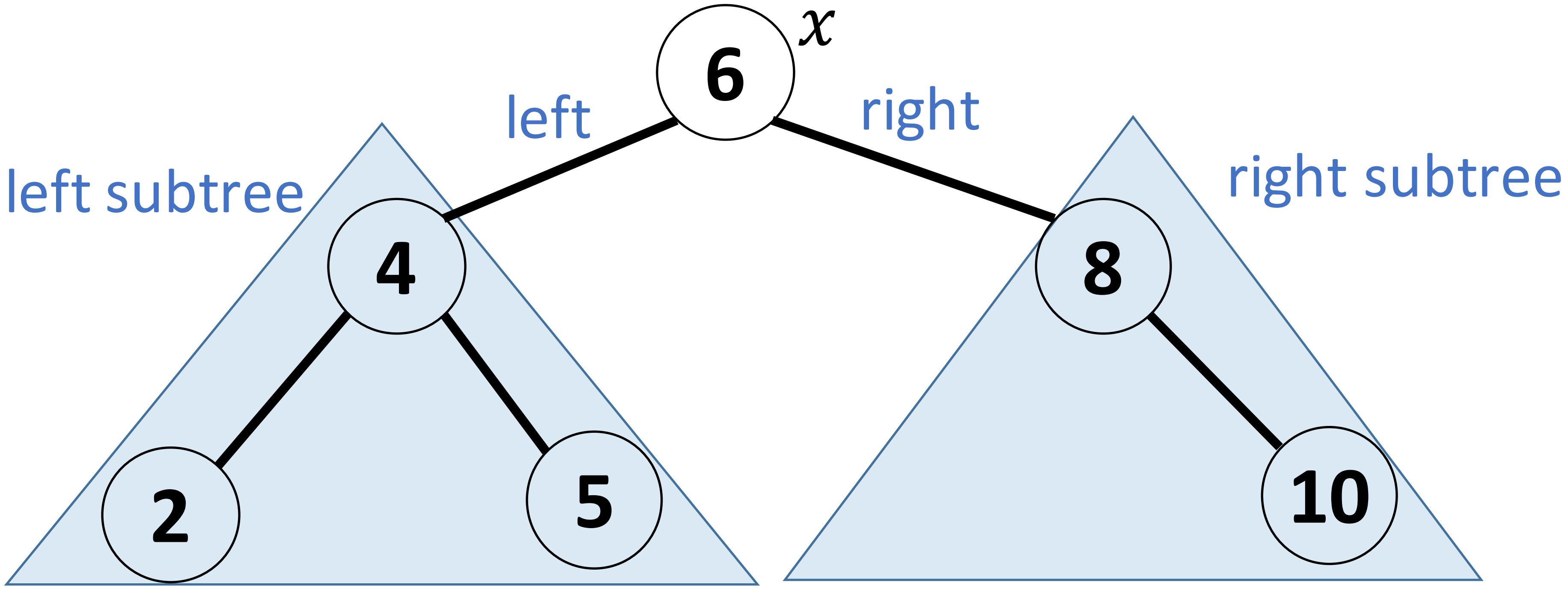
***x*.*right*:** *pointer to the root x's right subtree*

***x*.*p*:** *pointer to x's parent*

- We can add **a data field** to the node, as well.

Binary Search Tree

- Example:

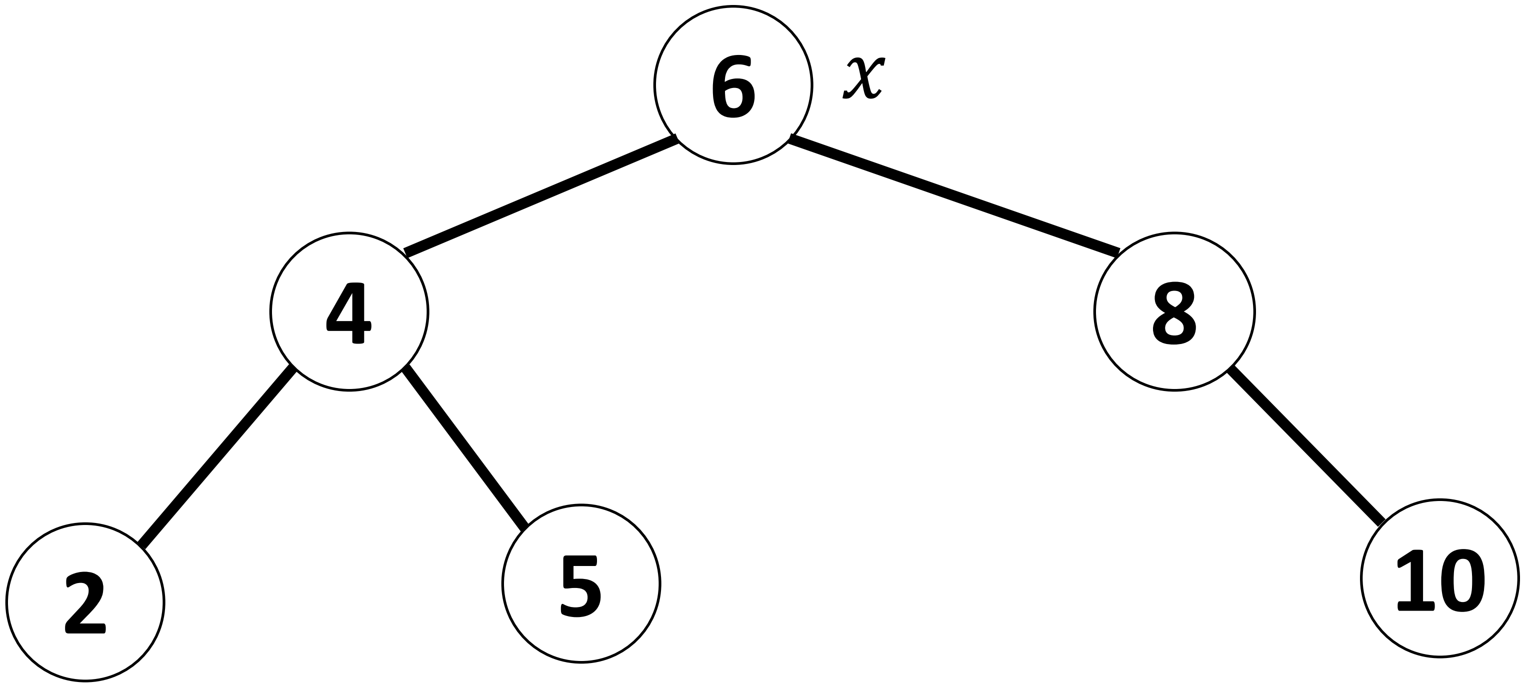


Binary Search Tree

- **BST's invariant:**

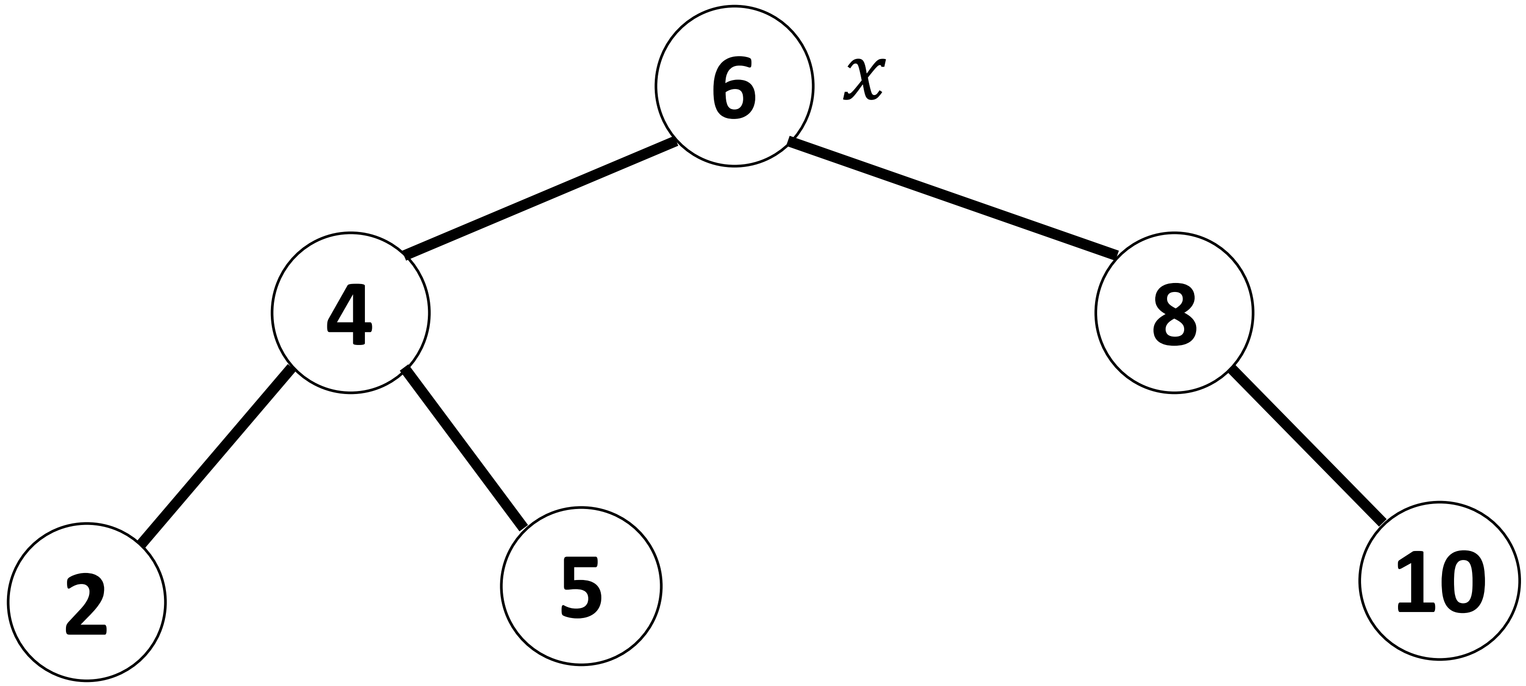
(1) Every node to the left of x has key $< x.key$, and

(2) Every node to the right of x has key $> x.key$.



Sorting using BST

- **Question:** Having a BST, how can we print all elements in sorted order recursively?



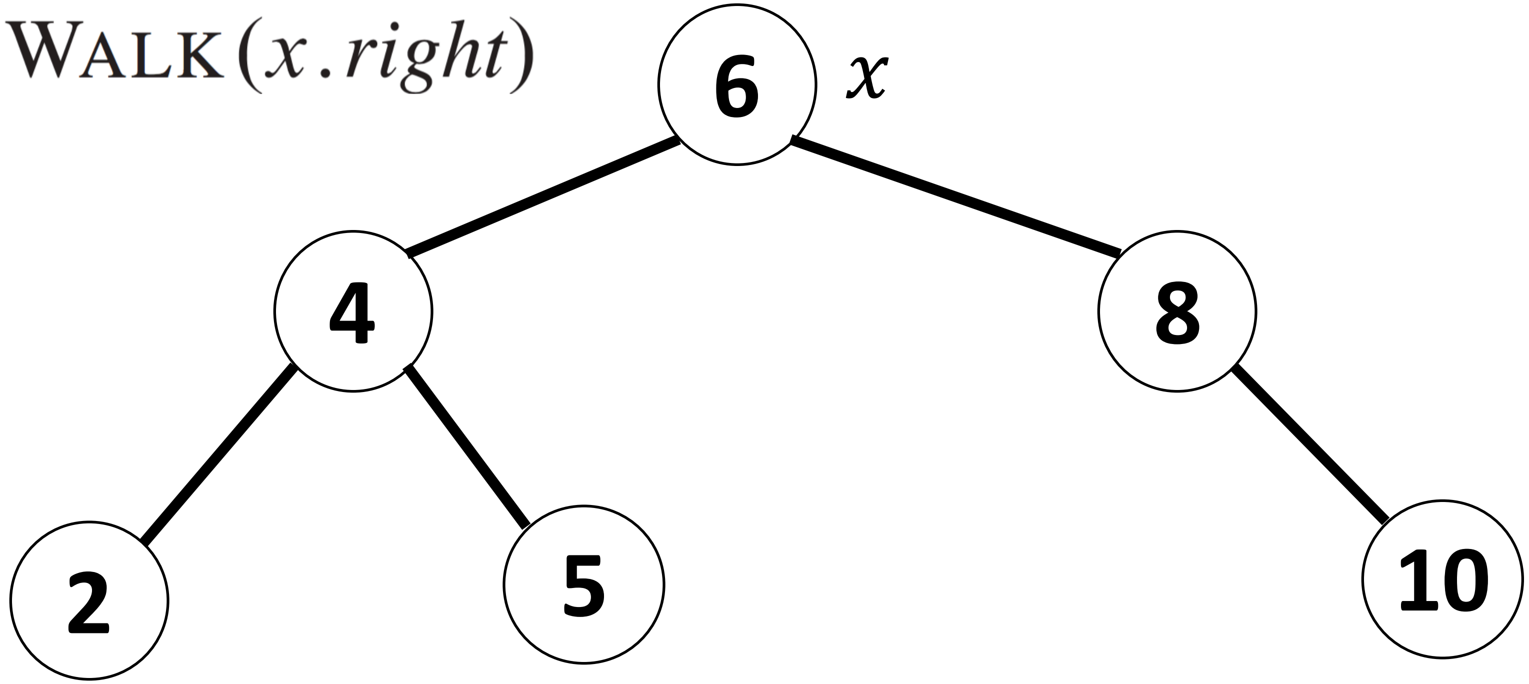
Sorting using BST

- **Question:** Having a BST, how can we print all elements in sorted order recursively?
- **Answer:**

1. print the left subtree in sorted order
2. print the root
3. print the right subtree in sorted order

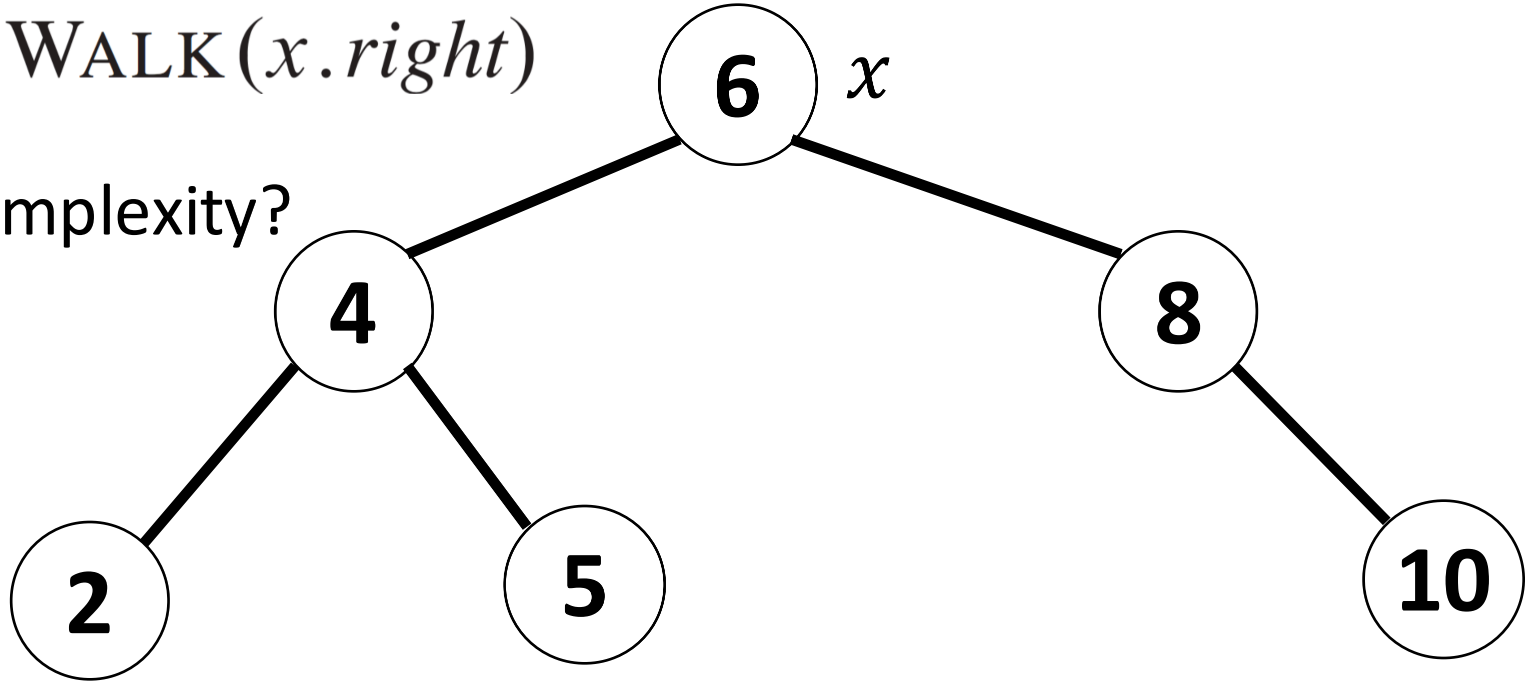
Inorder walk

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```



Inorder walk

```
INORDER-TREE-WALK(x)
    if x ≠ NIL
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```

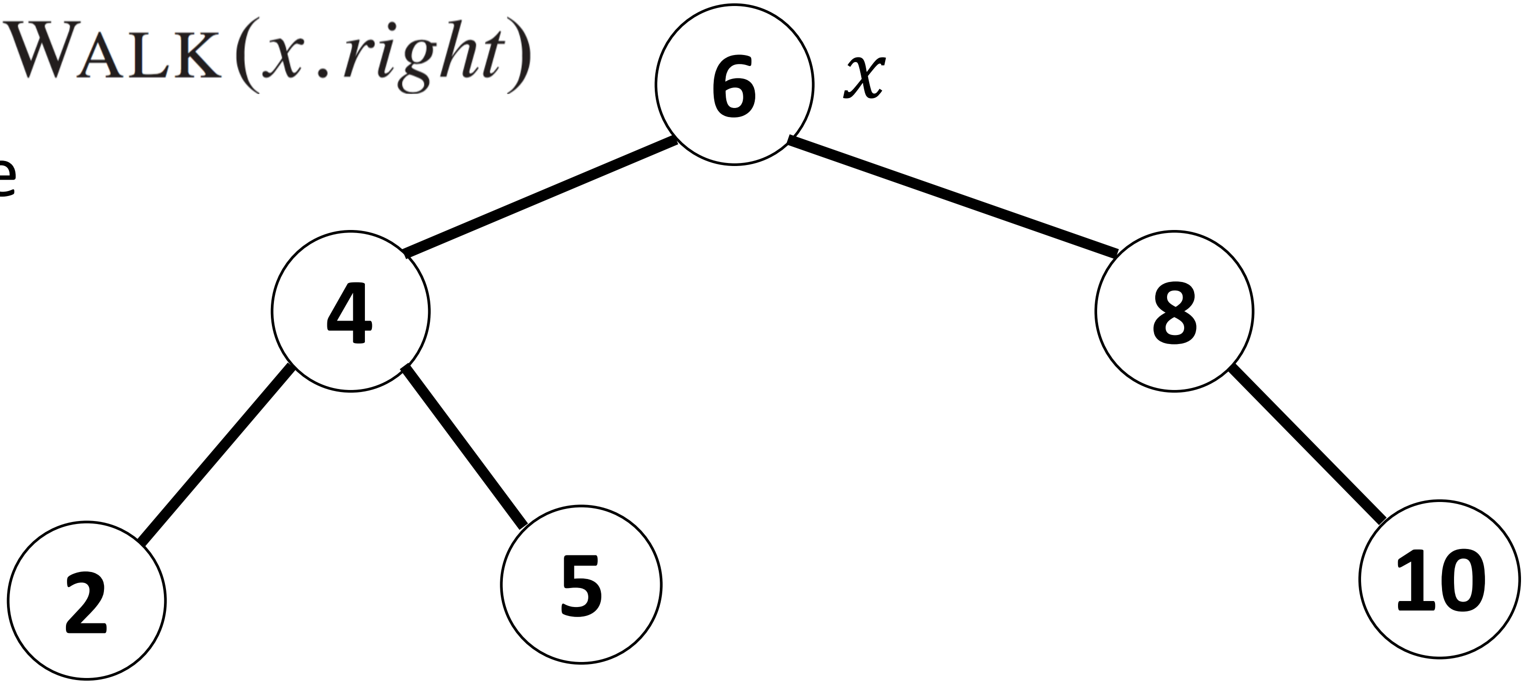


Question: What is the time complexity?

Inorder walk

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

Answer: $\Theta(n)$ since each node is visited exactly once and the amount of work per node is constant: 1 condition check 2 function calls, 1 print, and a few memory accesses



```
PREORDER-WALK(x)
    if x != NIL
        print x.key
        PREORDER-WALK(x.left)
        PREORDER-WALK(x.right)
```

```
POSTORDER-WALK(x)
    if x != NIL
        POSTORDER-WALK(x.left)
        POSTORDER-WALK(x.right)
        print x.key
```

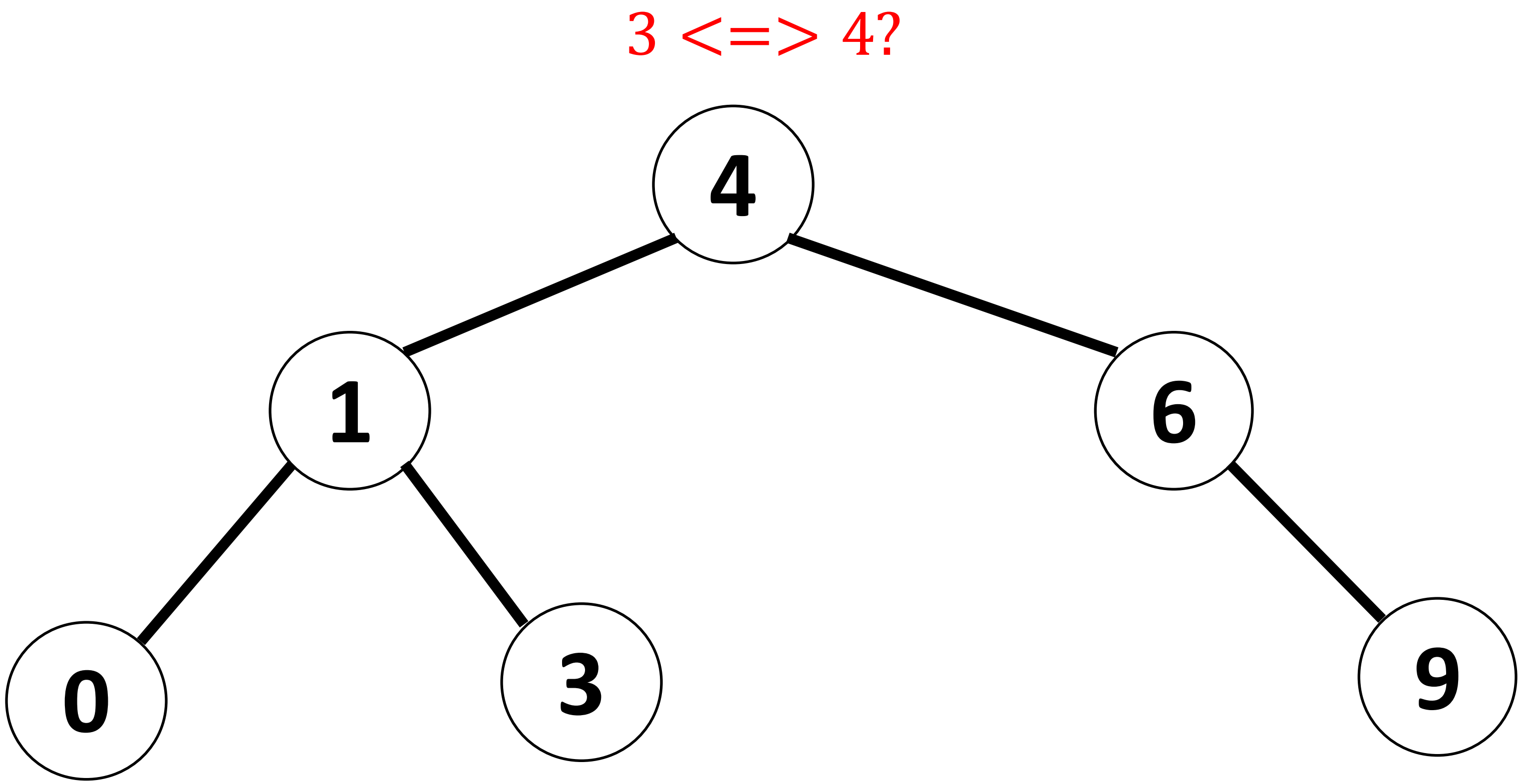
```
PREORDER-WALK(x)
    if x != NIL
        print x.key
        PREORDER-WALK(x.left)
        PREORDER-WALK(x.right)
```

```
POSTORDER-WALK(x)
    if x != NIL
        POSTORDER-WALK(x.left)
        POSTORDER-WALK(x.right)
        print x.key
```

- A preorder walk approach can be used when we want to copy the nodes of a tree. First we copy the root then the subtrees.
- A postorder walk approach can be used when we want to delete the nodes of a tree. First we delete the subtrees then the root.

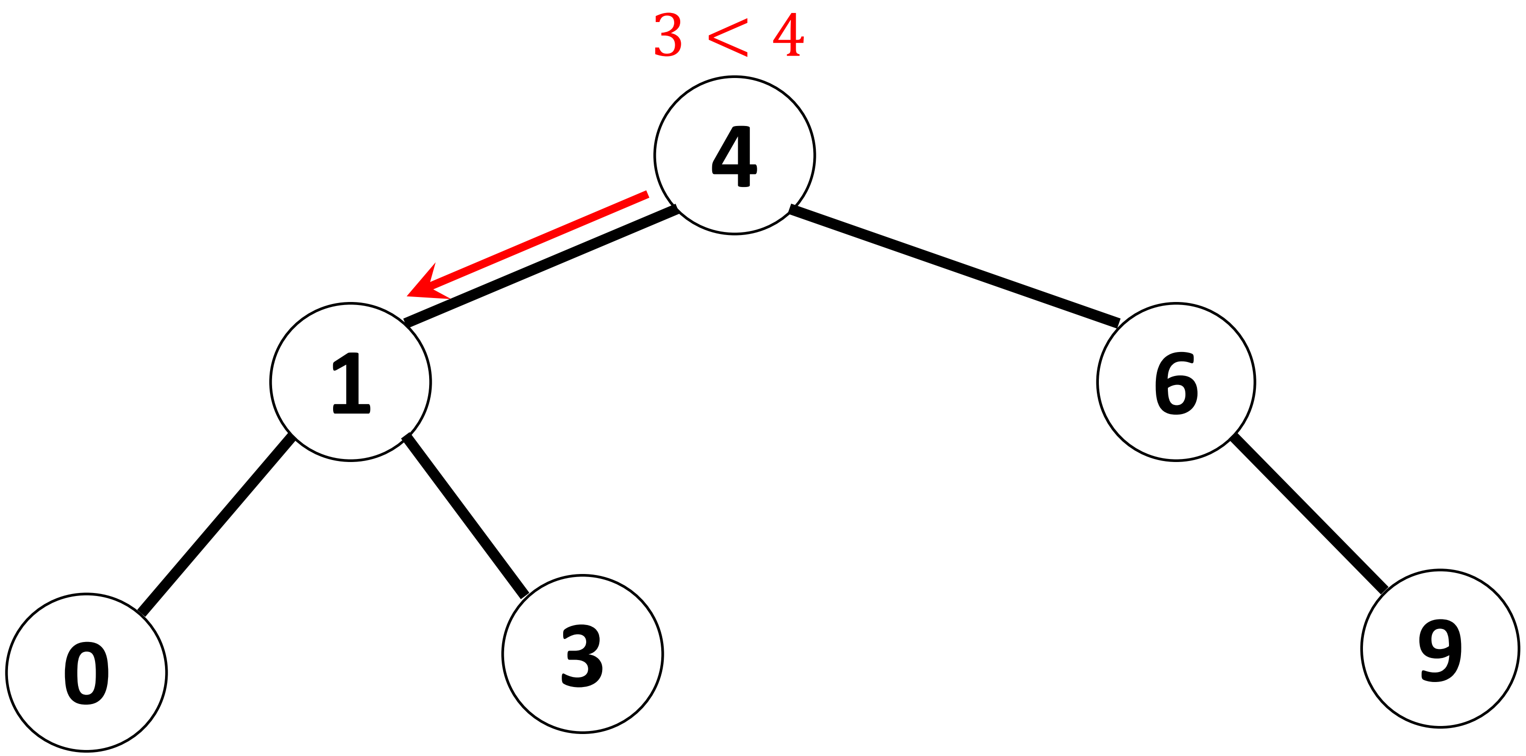
Search

- Search(3)



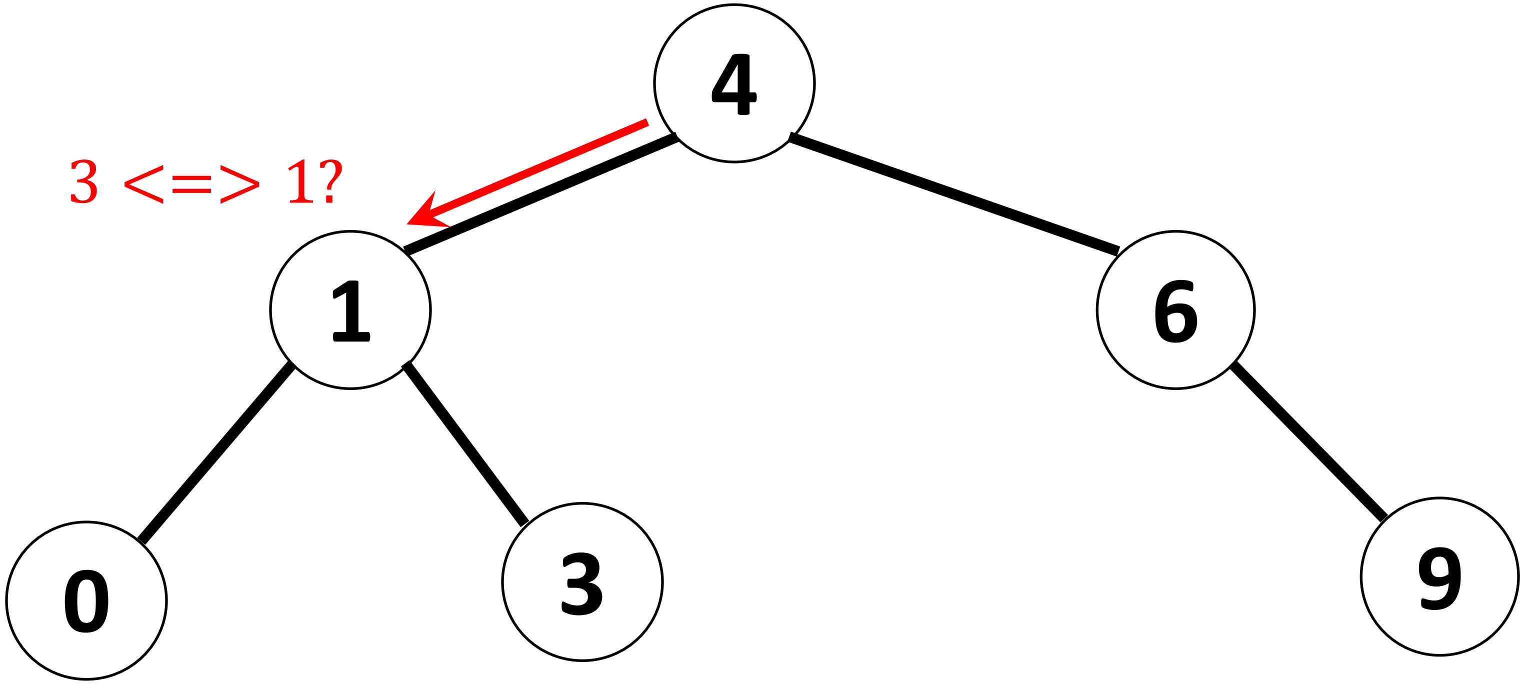
Search

- Search(3)



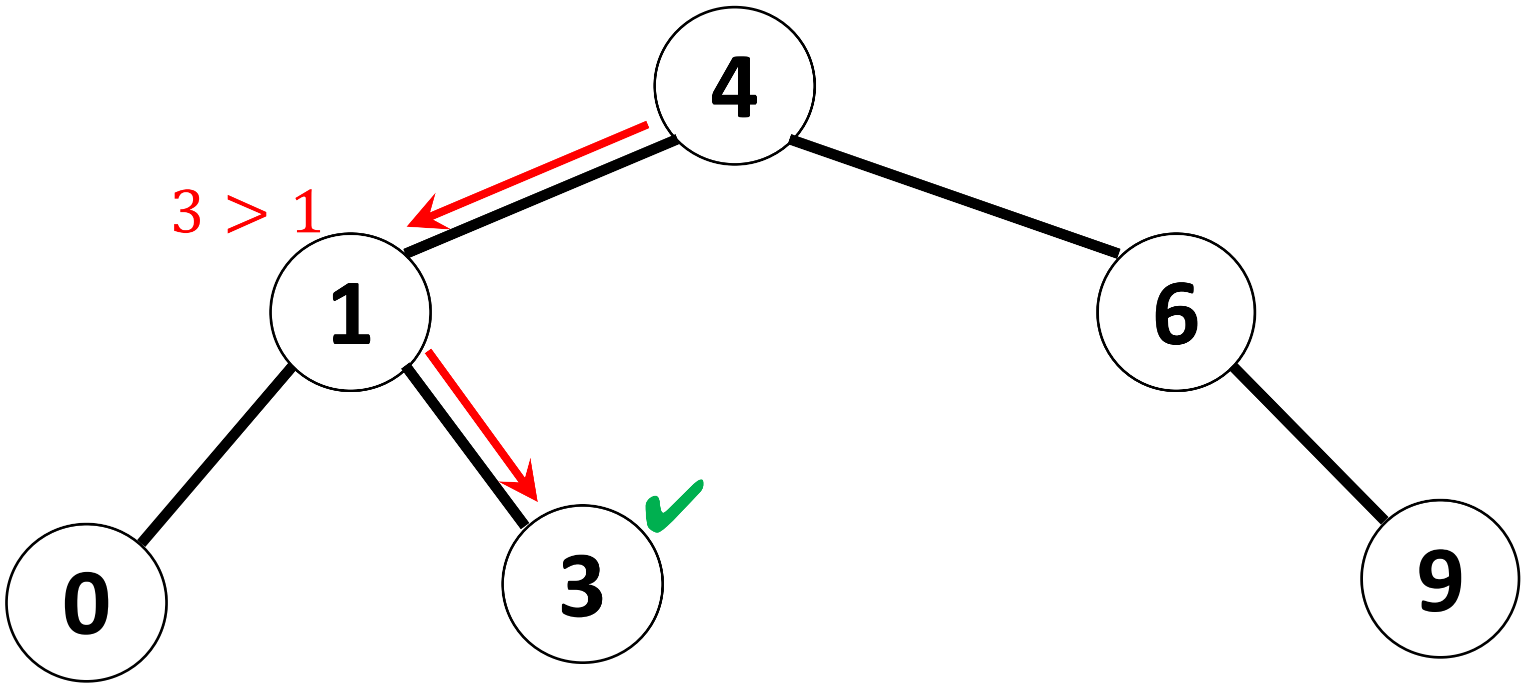
Search

- Search(3)



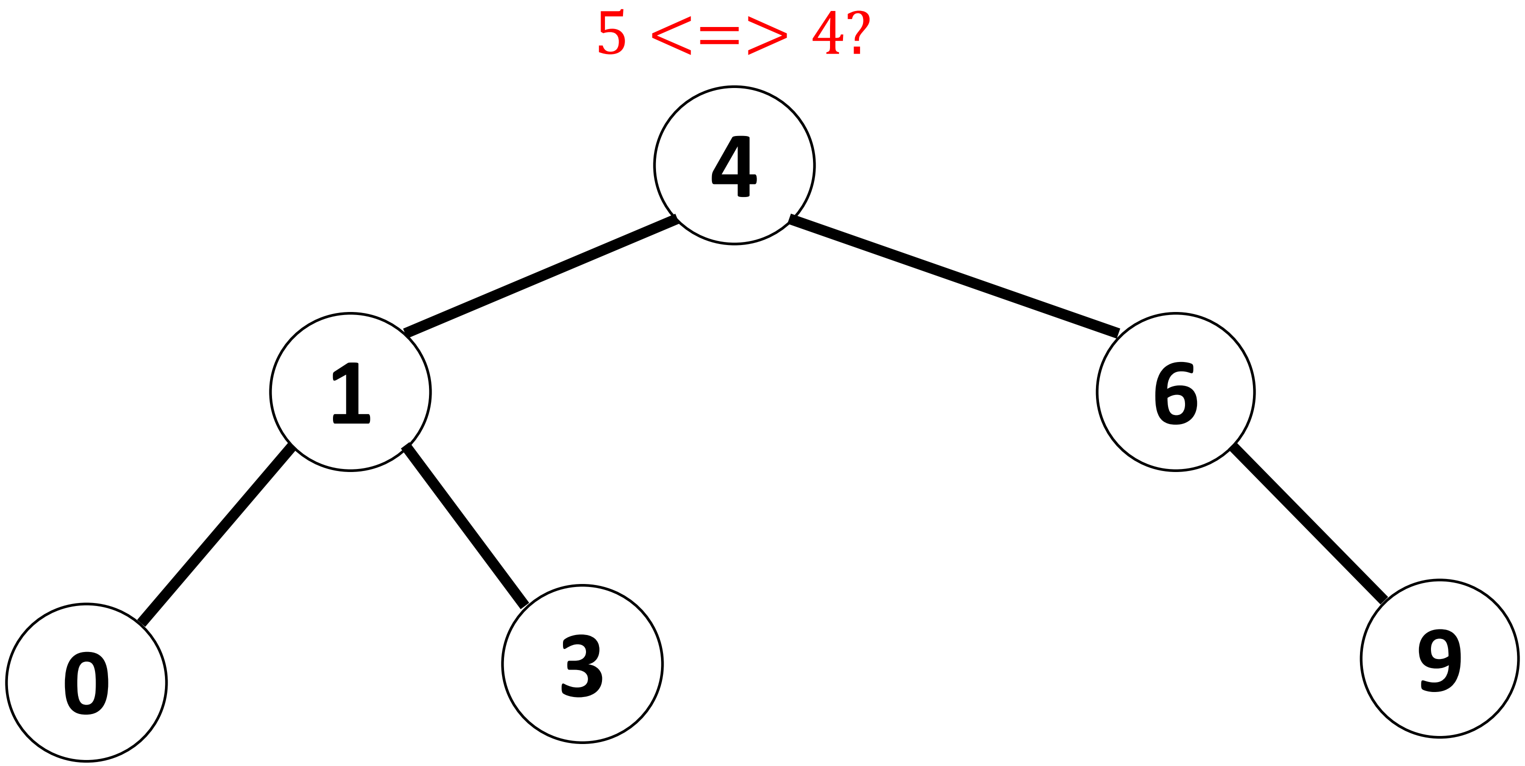
Search

- Search(3)



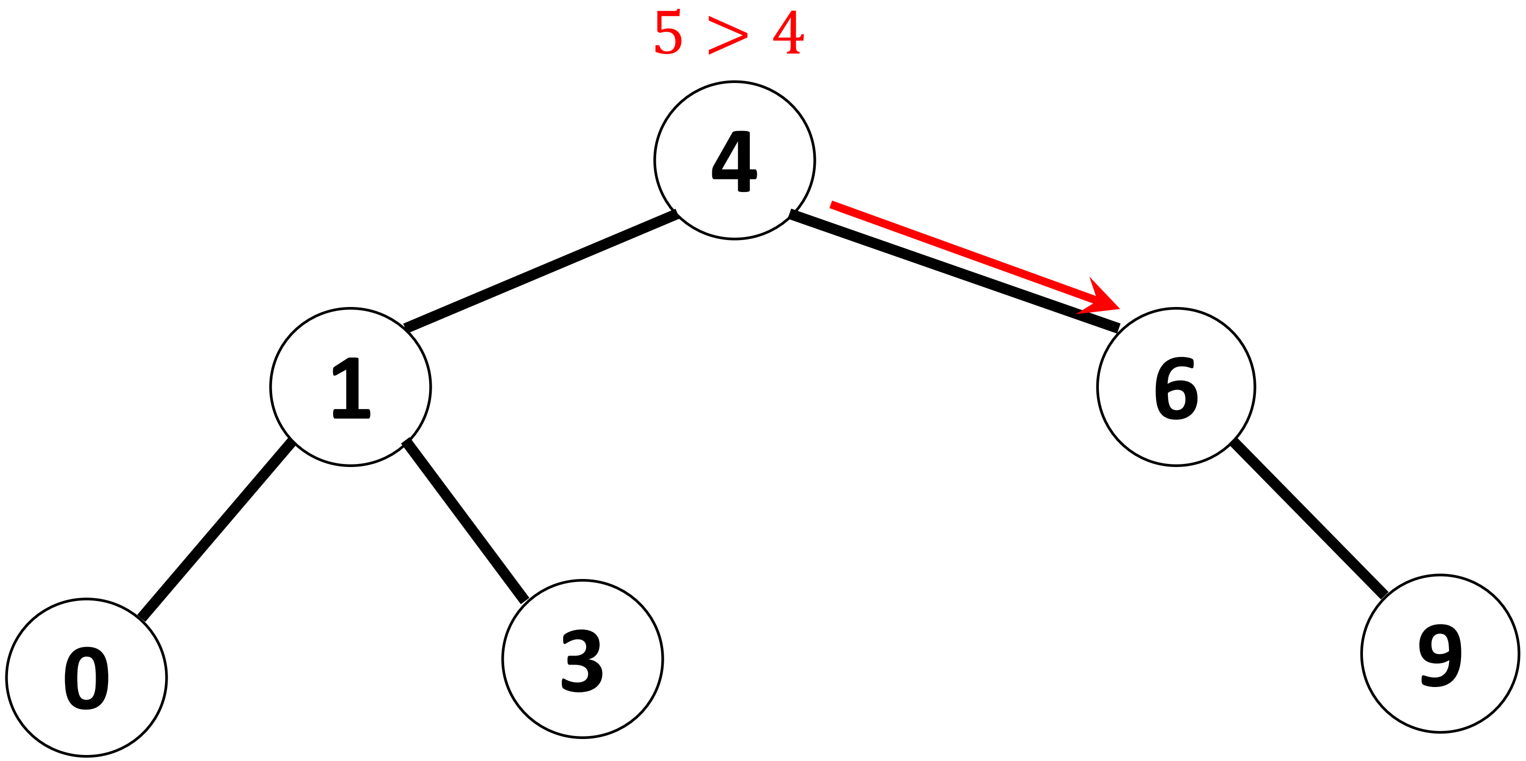
Search

- Search(5)



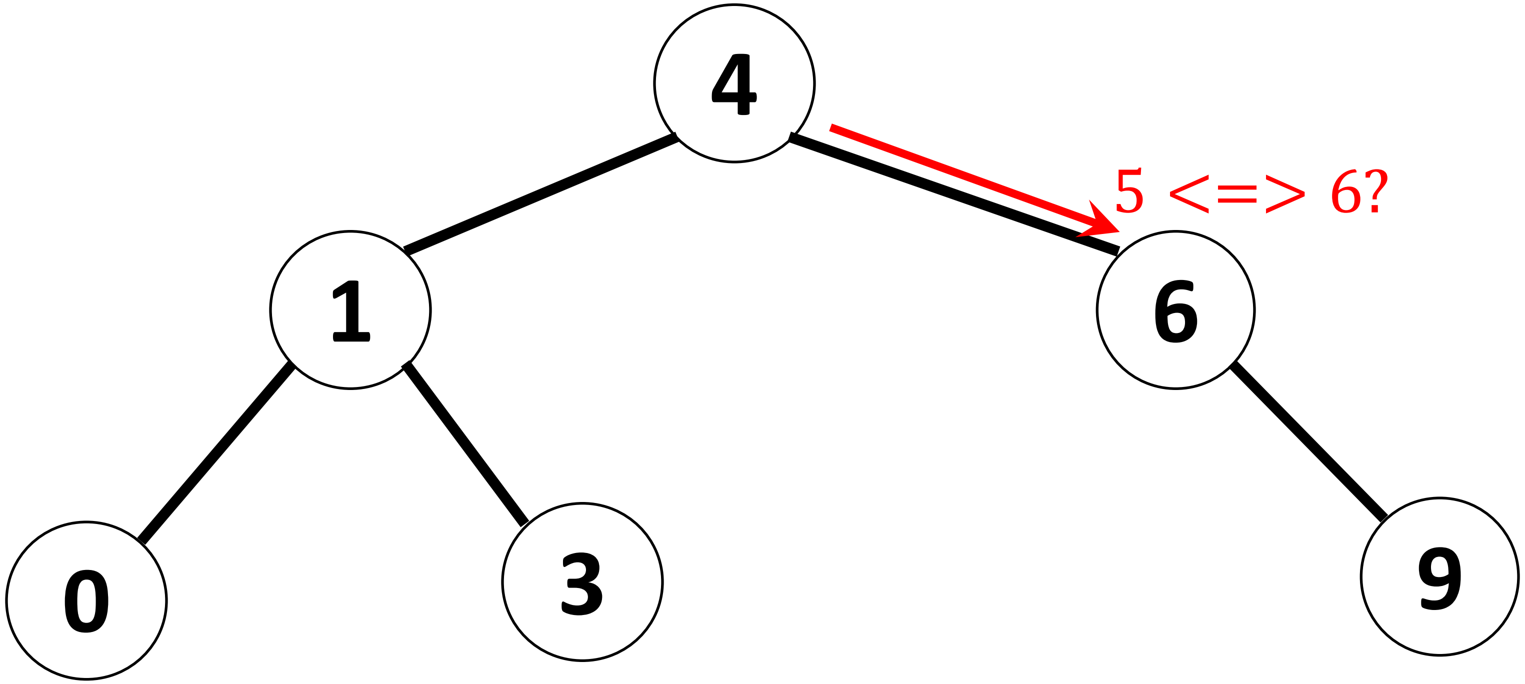
Search

- Search(5)



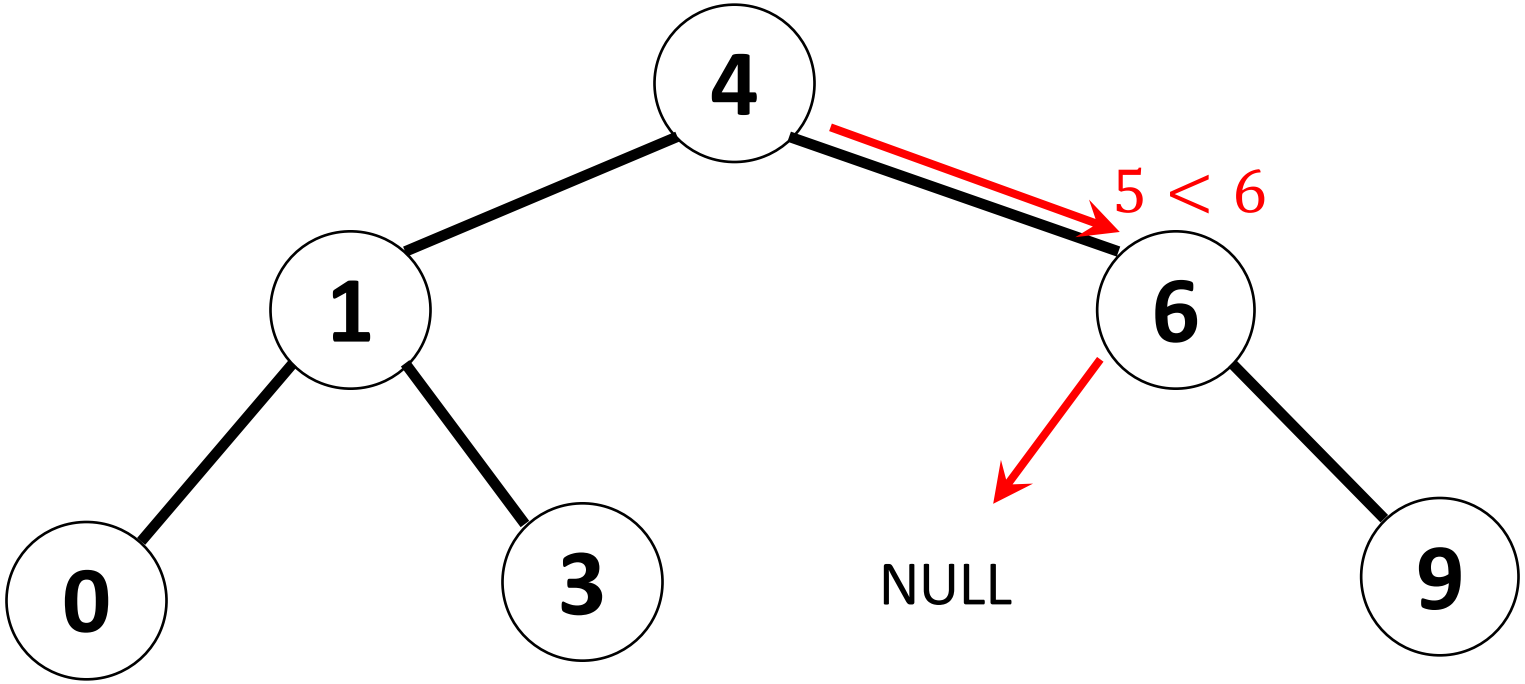
Search

- Search(5)



Search

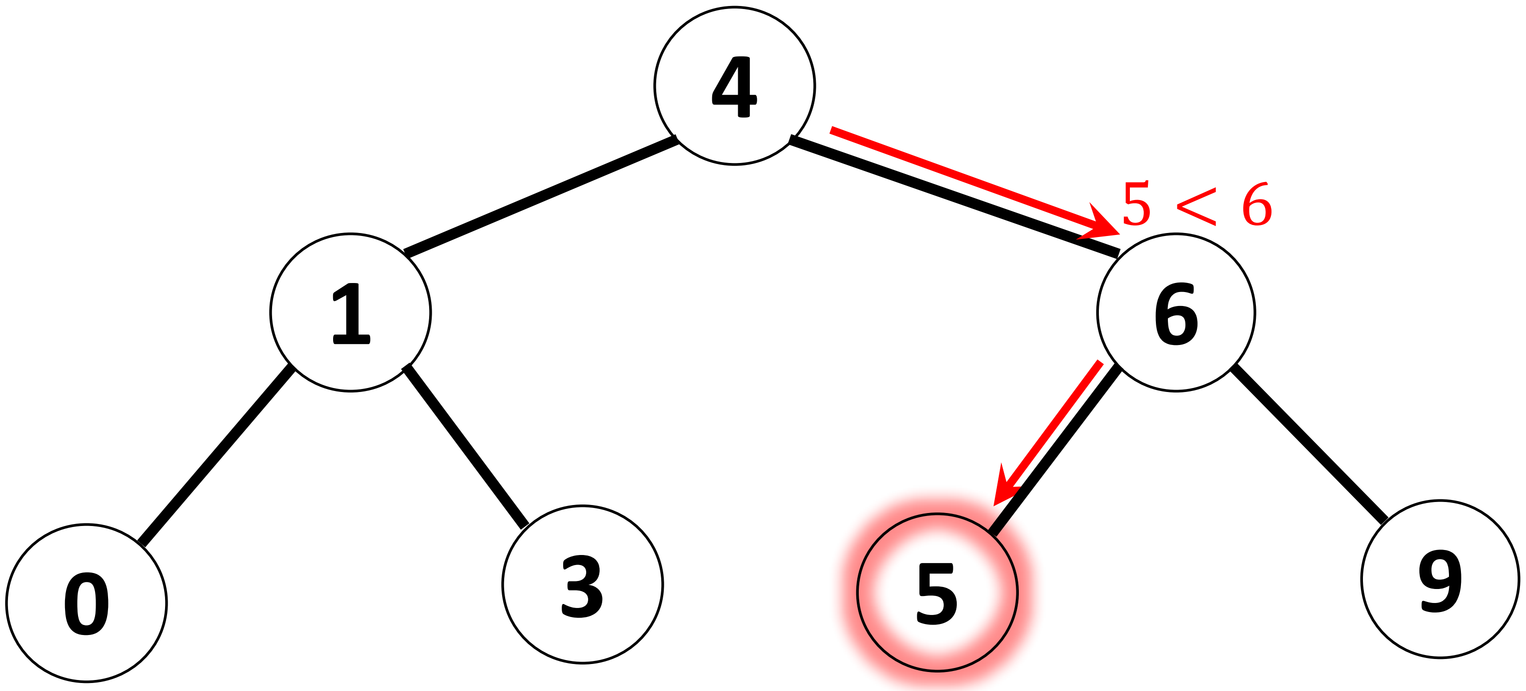
- Search(5)



But this is useful!

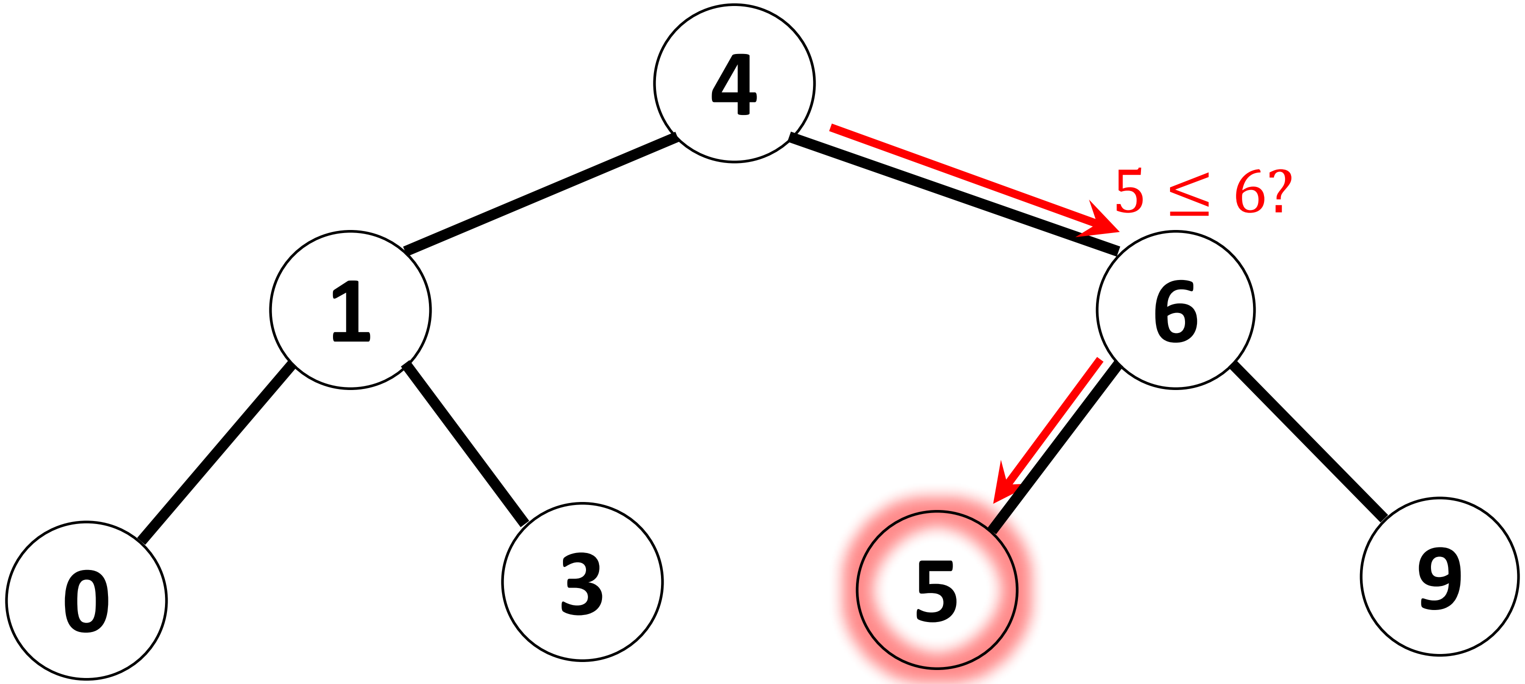
Insert

- Insert(5)



Insert

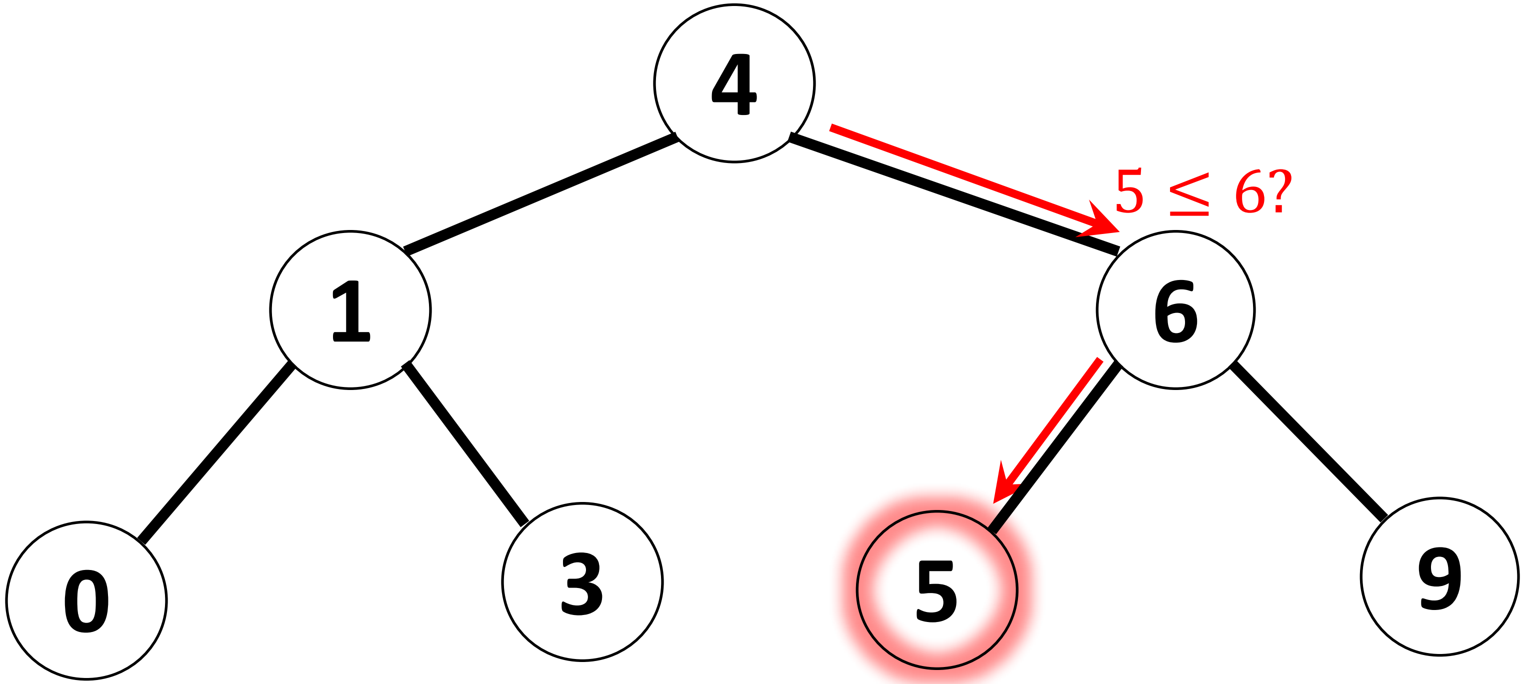
- Insert(5)



Question: What is the running time of insert and search?

Insert

- Insert(5)



Question: What is the running time of insert and search?
Answer: $O(h)$, where h is the height of the tree.

$\textsc{Search}(key)$

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

- For a binary search tree T, the operation $\textsc{Search}(key)$ is equivalent to calling $\textsc{Tree-Search}(T.root, key)$
- Section 12.2 in CLRS

INSERT(x)

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

- INSERT(x) can be implemented by calling TREE-INSERT(T, x).
- Section 12.3 in CLRS

INSERT(x)

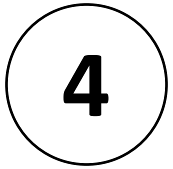
TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z            // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

- INSERT(x) can be implemented by calling TREE-INSERT(T, x).
- Section 12.3 in CLRS
- **Watch out for the elseif vs. else if!!!**

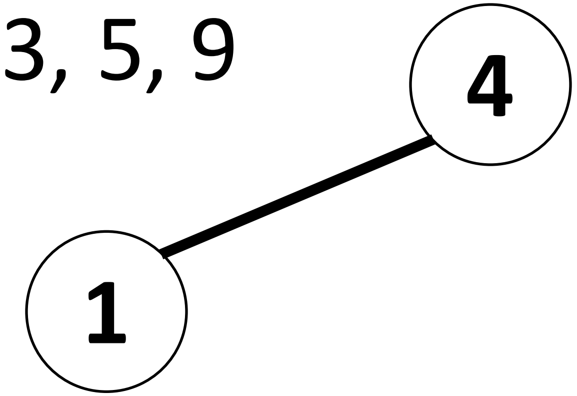
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



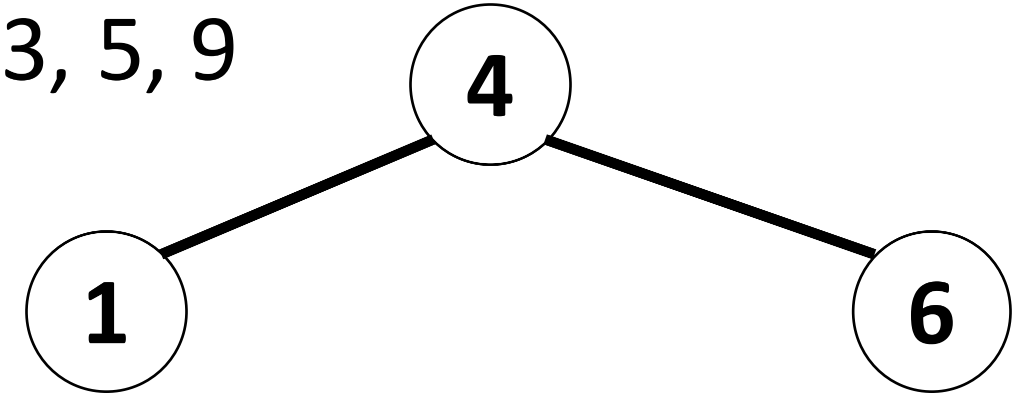
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



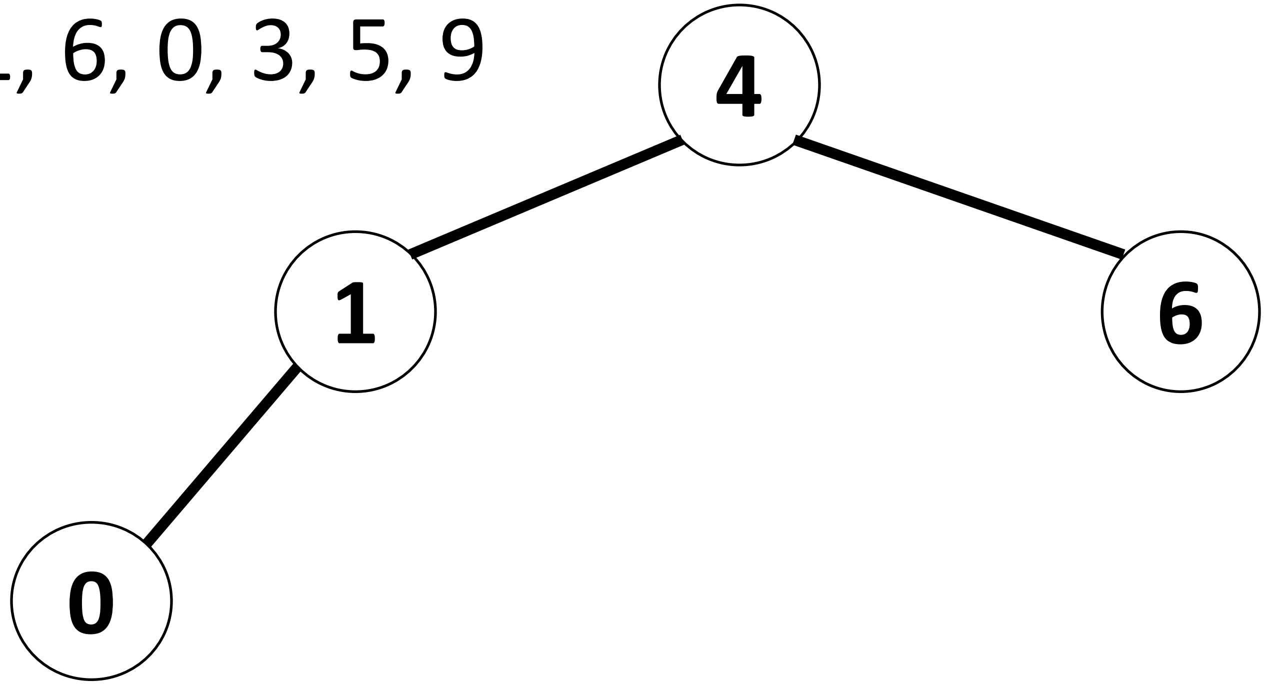
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



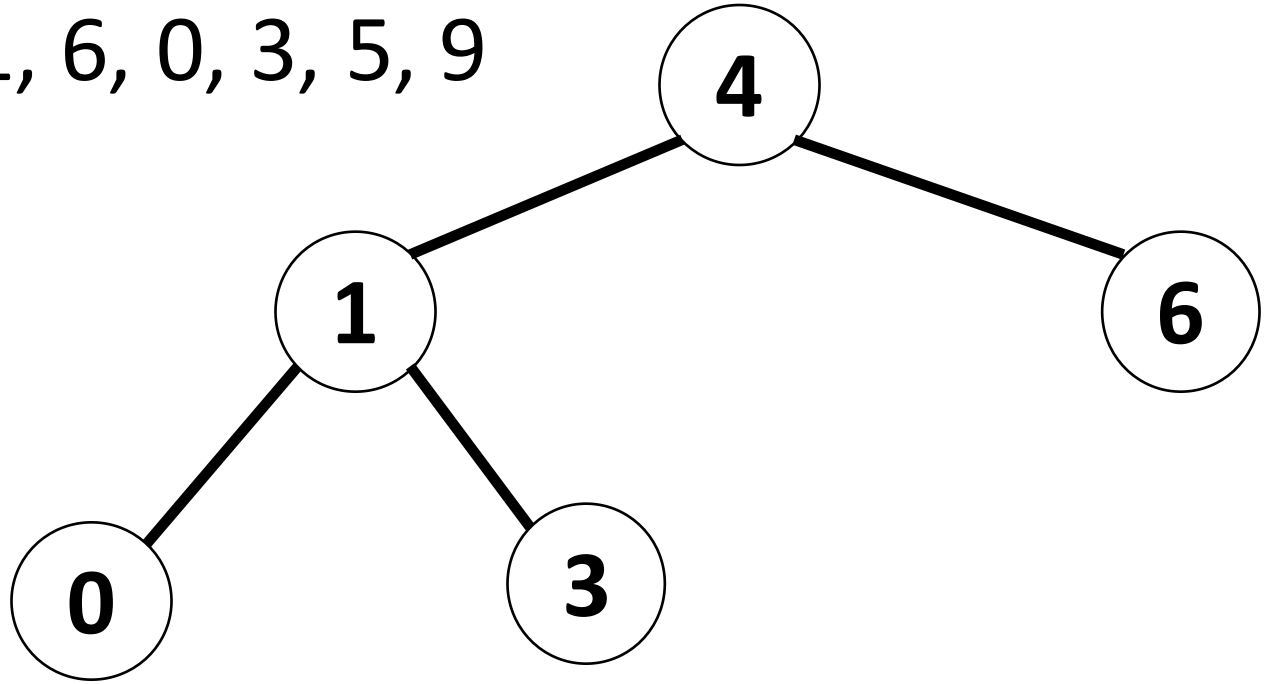
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



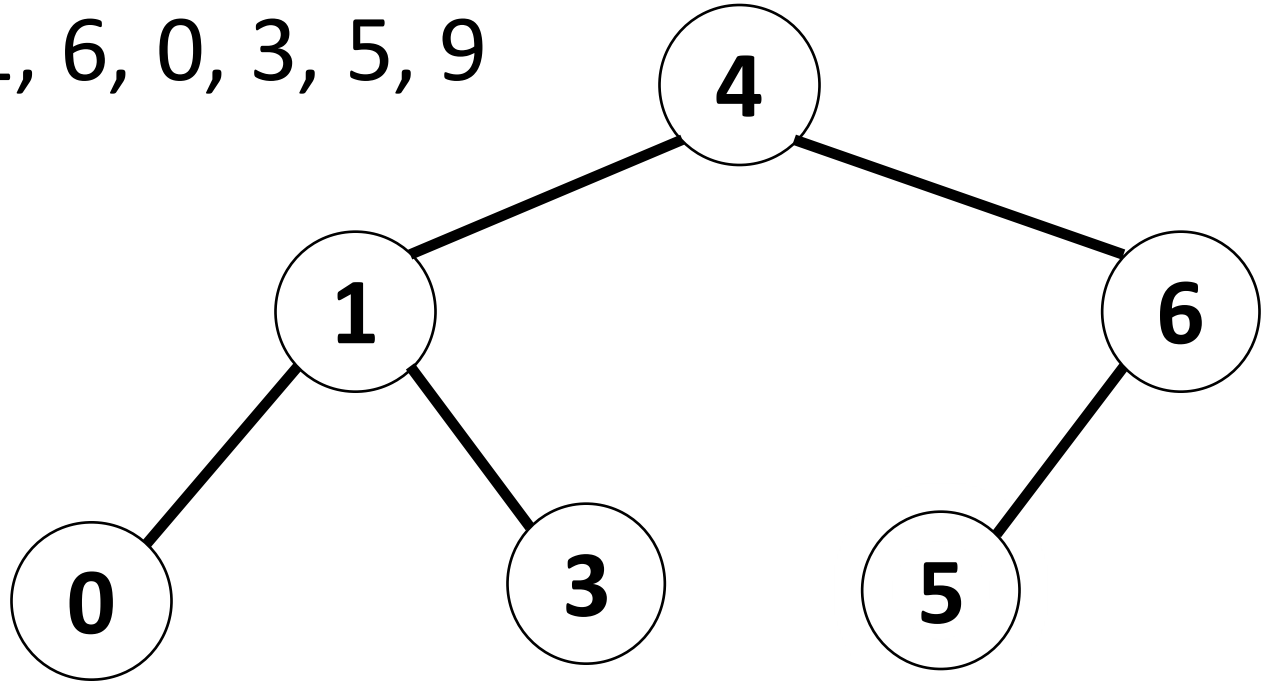
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



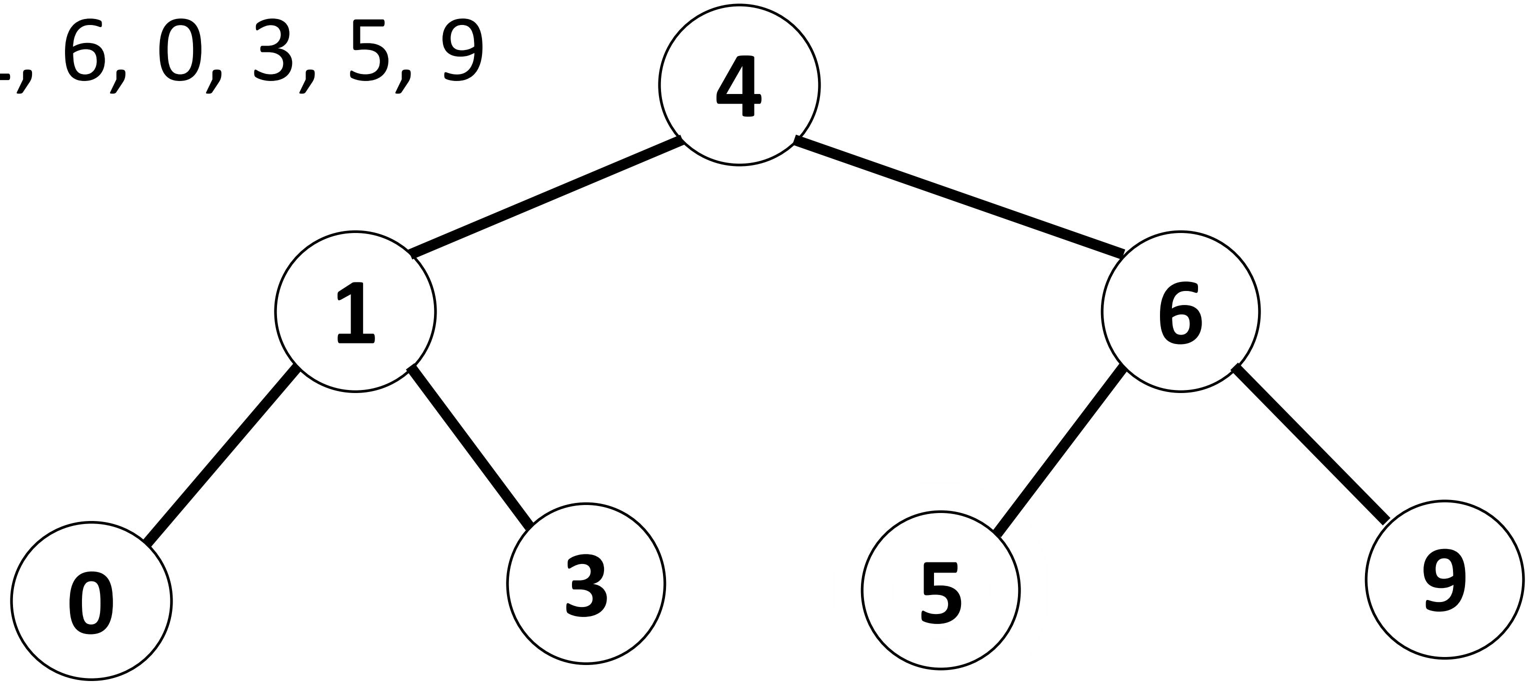
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



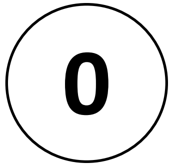
Building binary search tree

- To build the initial binary tree on a set of elements we can do the **insert operation** on the input elements one by one.
- Insert 4, 1, 6, 0, 3, 5, 9



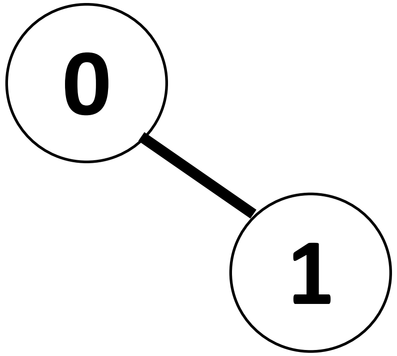
Building binary search tree

- However, order of insertion matters!
- Insert 0, 1, 3, 4, 5, 6, 9.



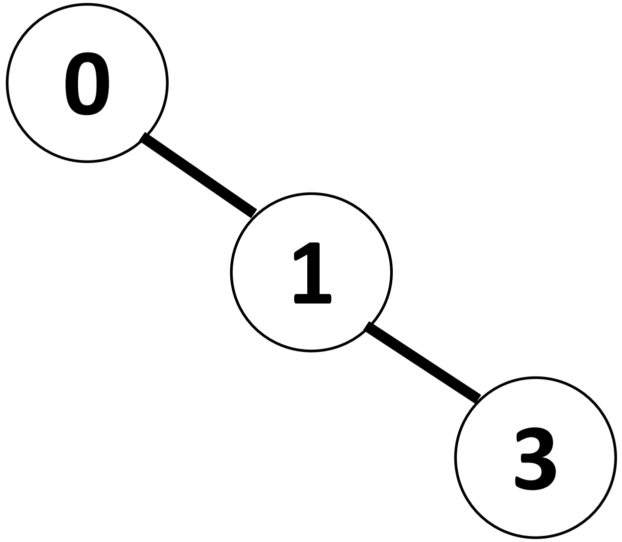
Building binary search tree

- However, order of insertion matters!
- Insert 0, 1, 3, 4, 5, 6, 9.



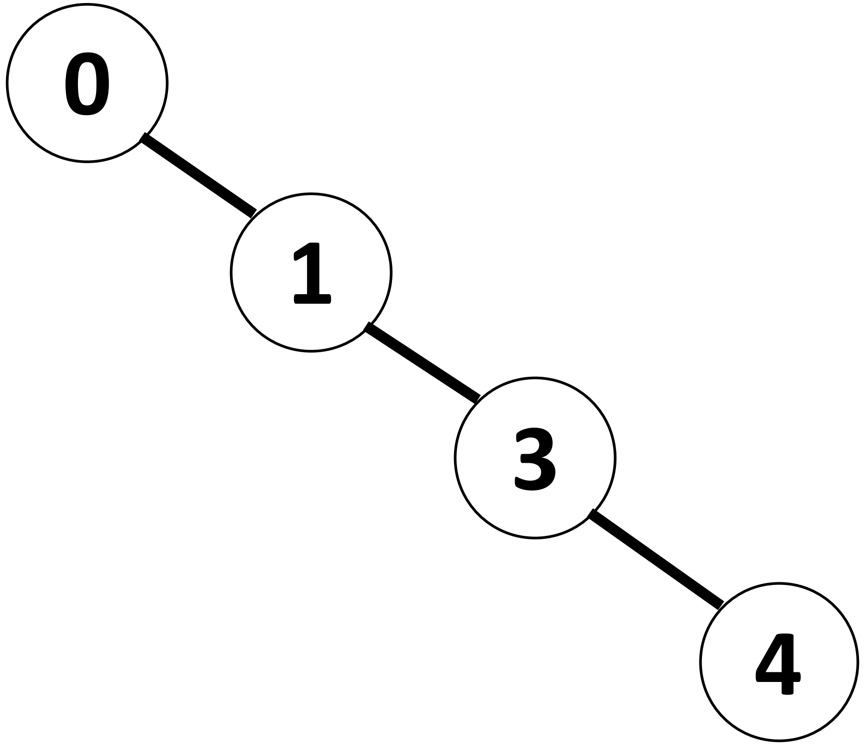
Building binary search tree

- However, order of insertion matters!
- Insert 0, 1, 3, 4, 5, 6, 9.



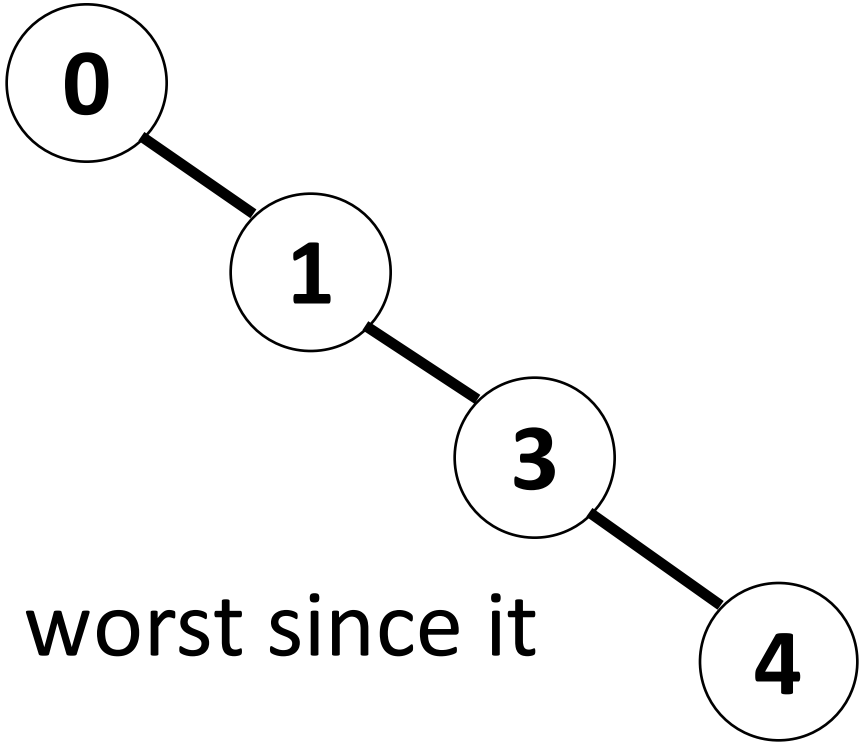
Building binary search tree

- However, order of insertion matters!
- Insert 0, 1, 3, 4, 5, 6, 9.



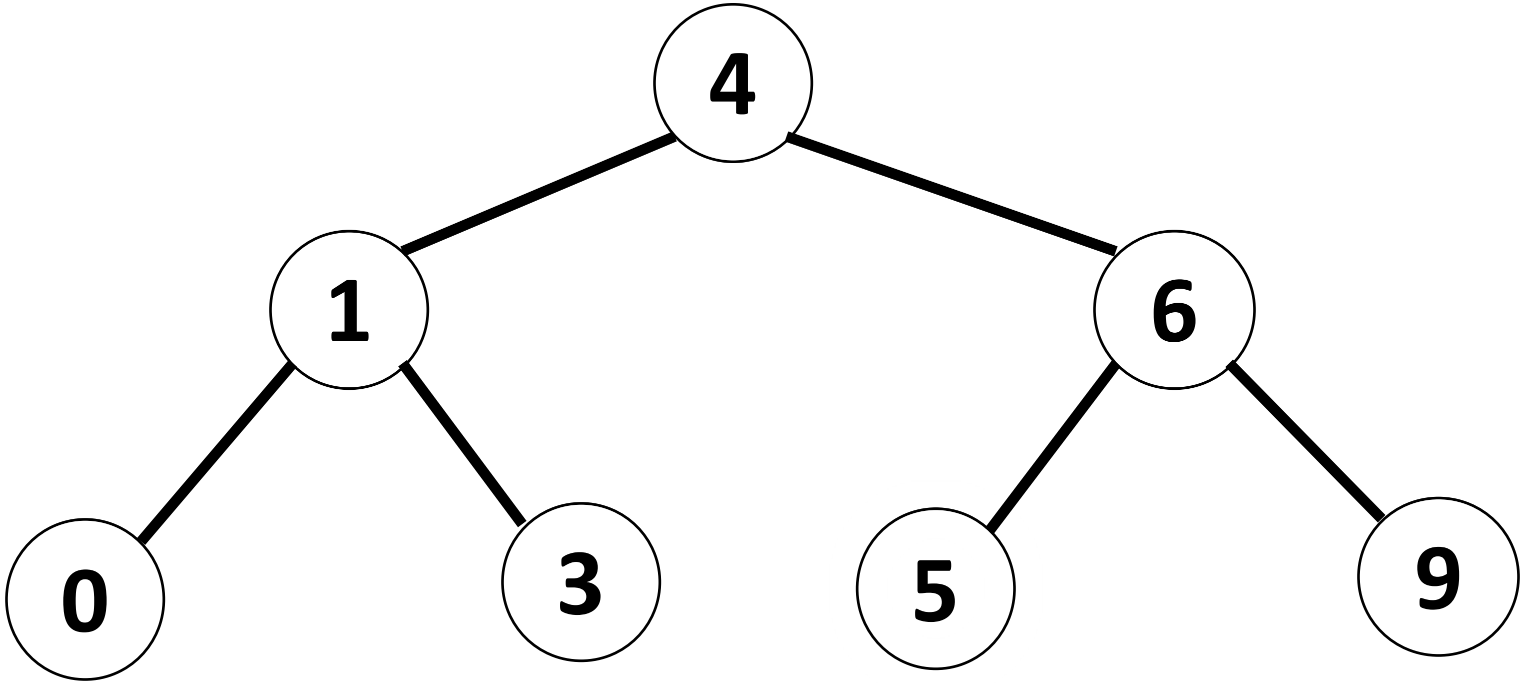
Building binary search tree

- However, order of insertion matters!
- Insert 0, 1, 3, 4, 5, 6, 9.



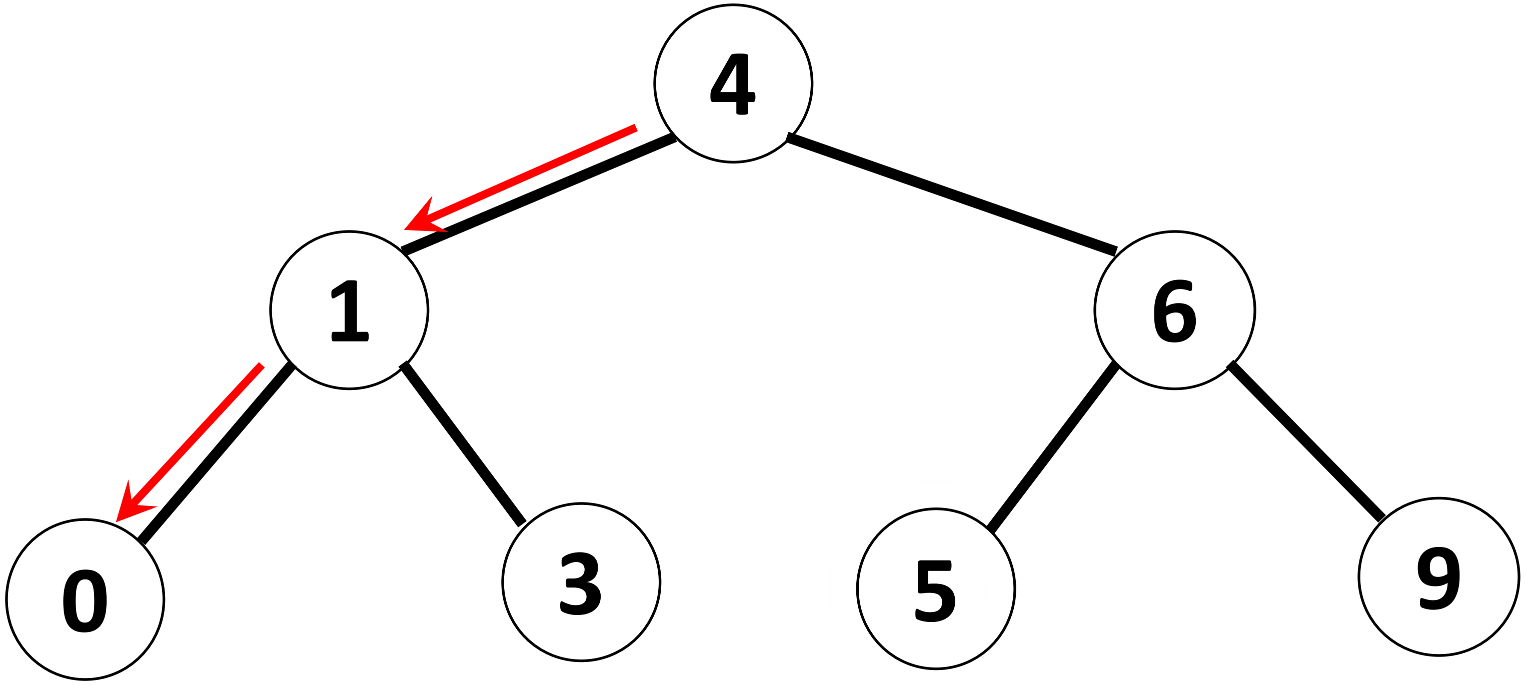
Sorted or reverse sorted is worst since it makes the height $\Theta(n)$

Minimum



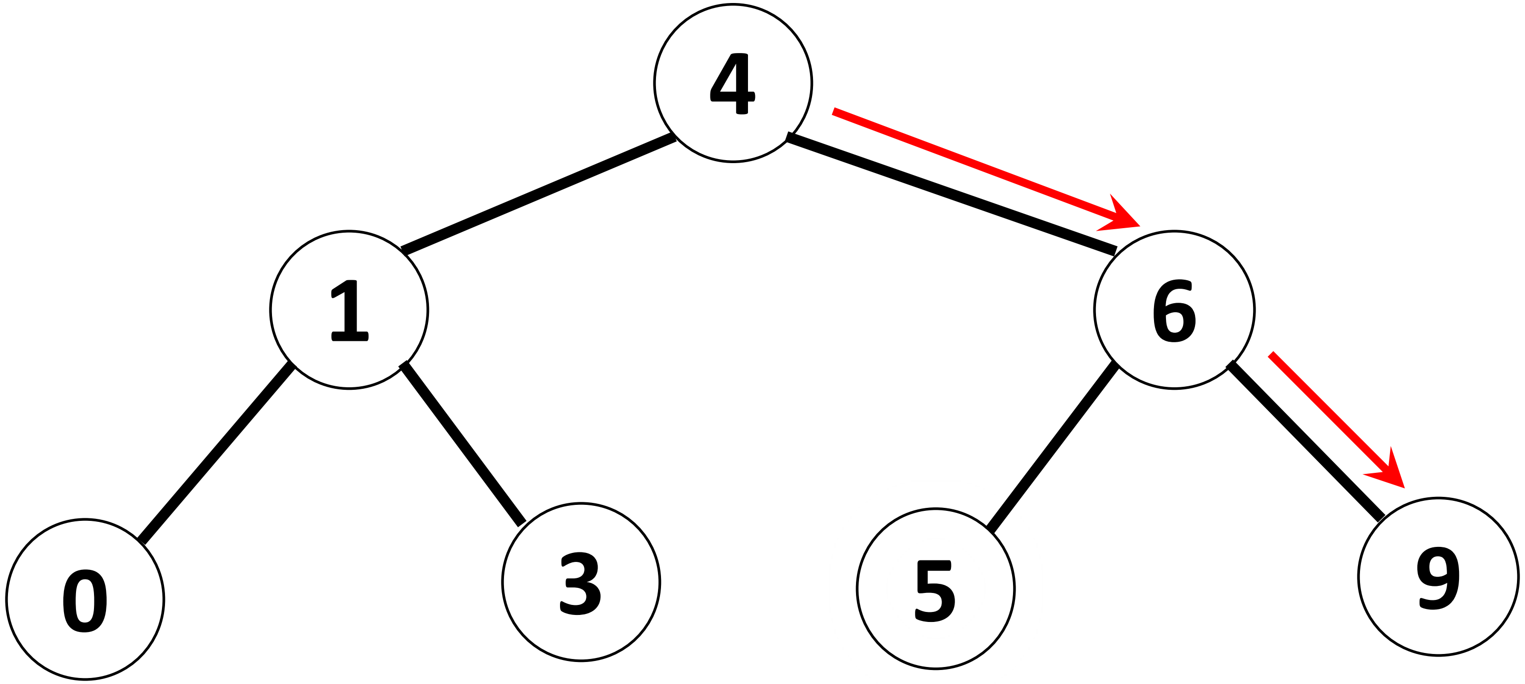
Minimum

- We keep going left until there is no left child

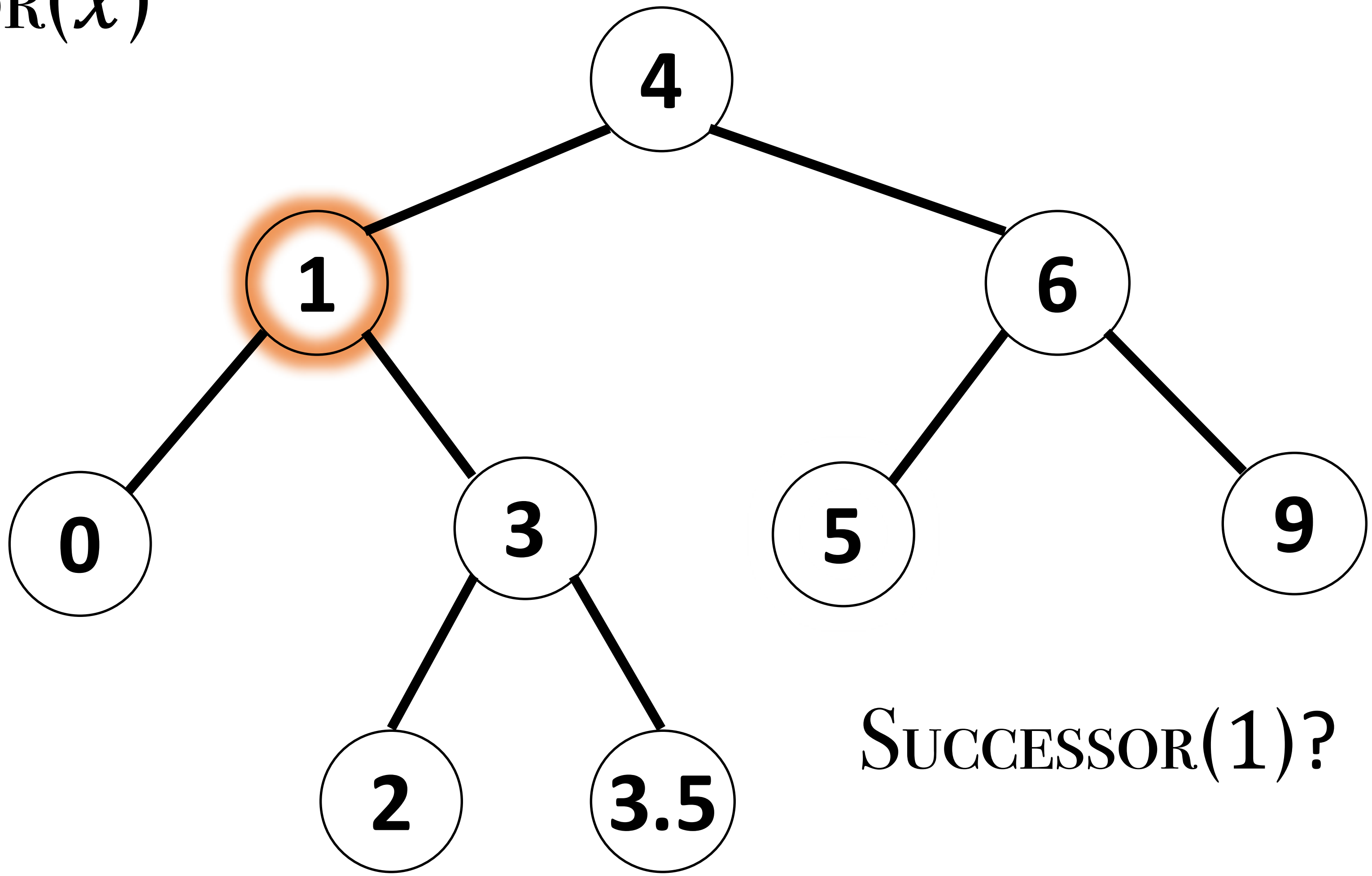


Maximum

- We keep going right until there is no right child

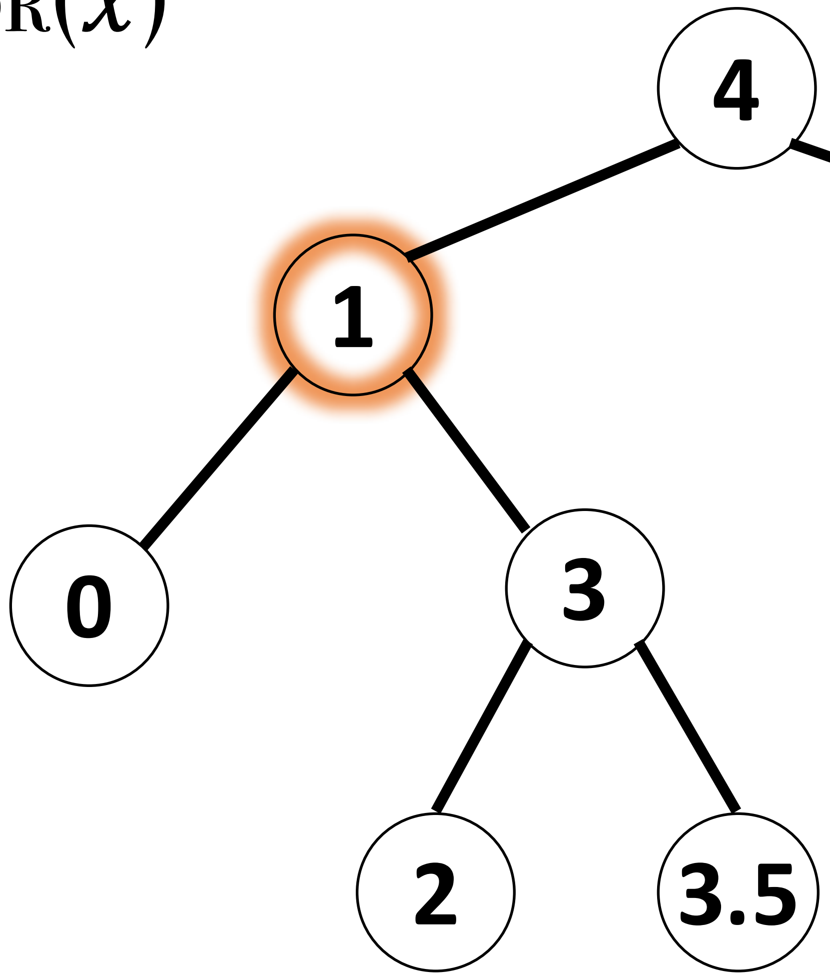


Successor(x)

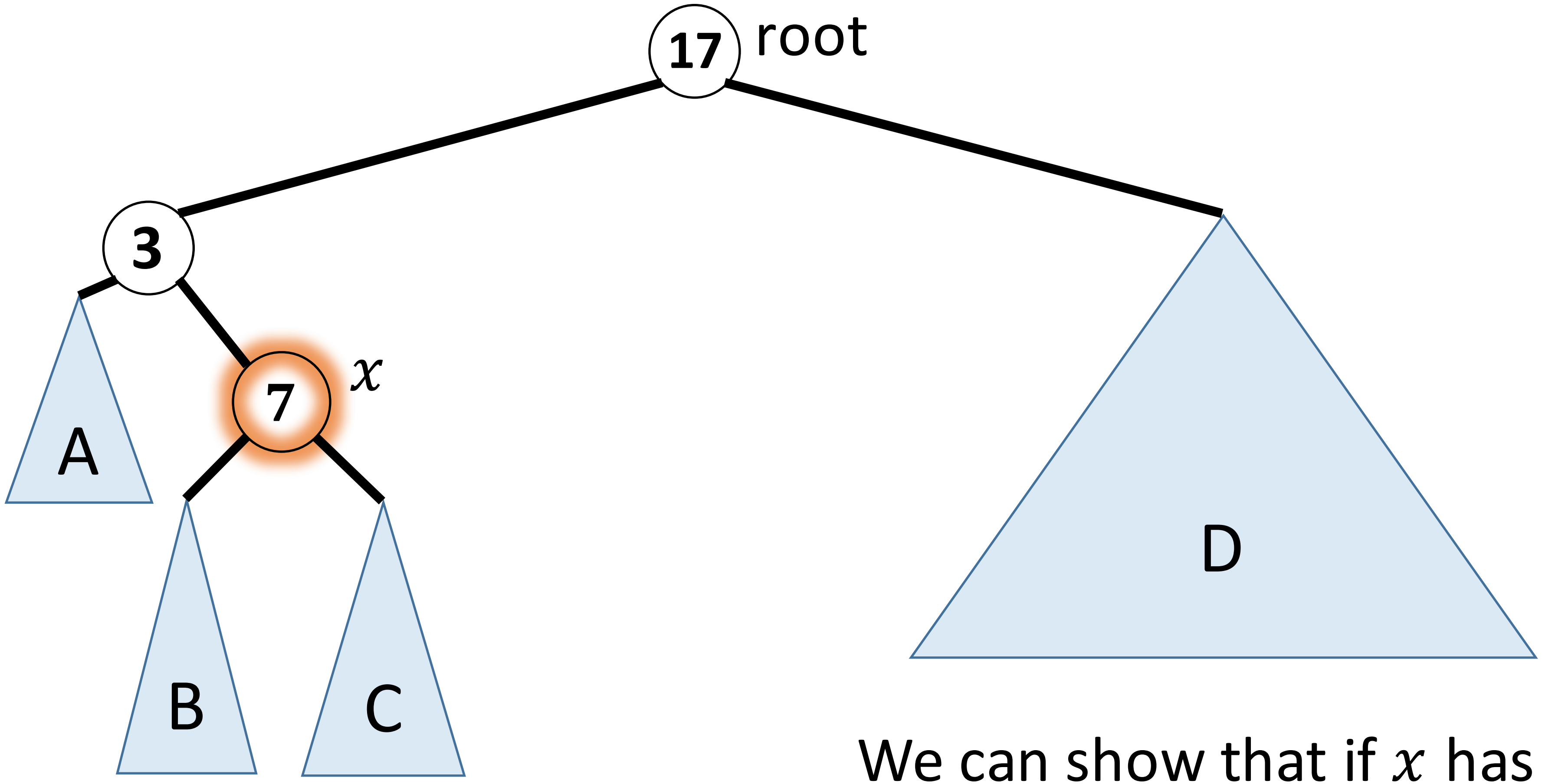


Successor(1)?

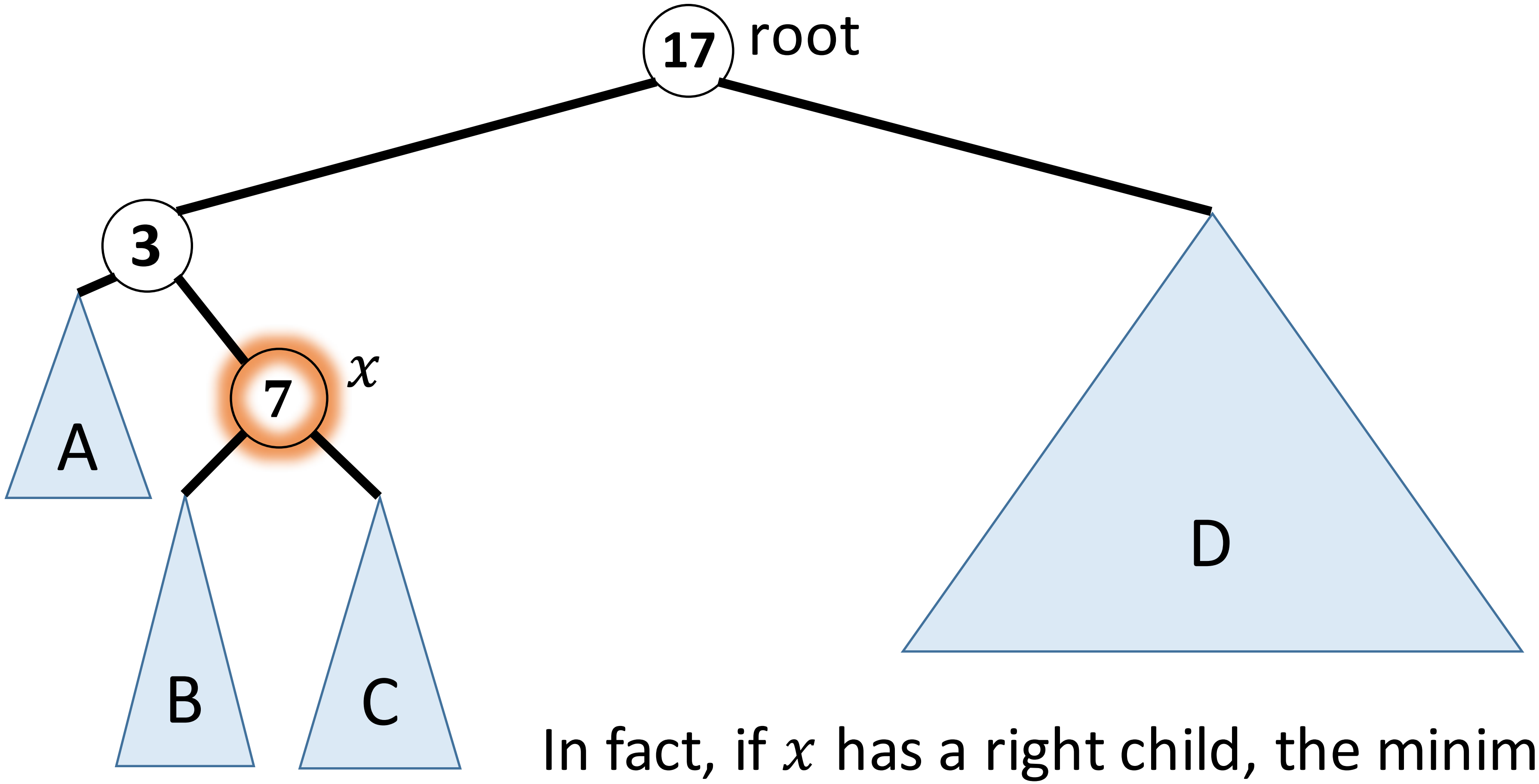
$\textsc{Successor}(x)$



$\textsc{Successor}(1)$?
It's 2, which is the minimum of 1's right subtree.



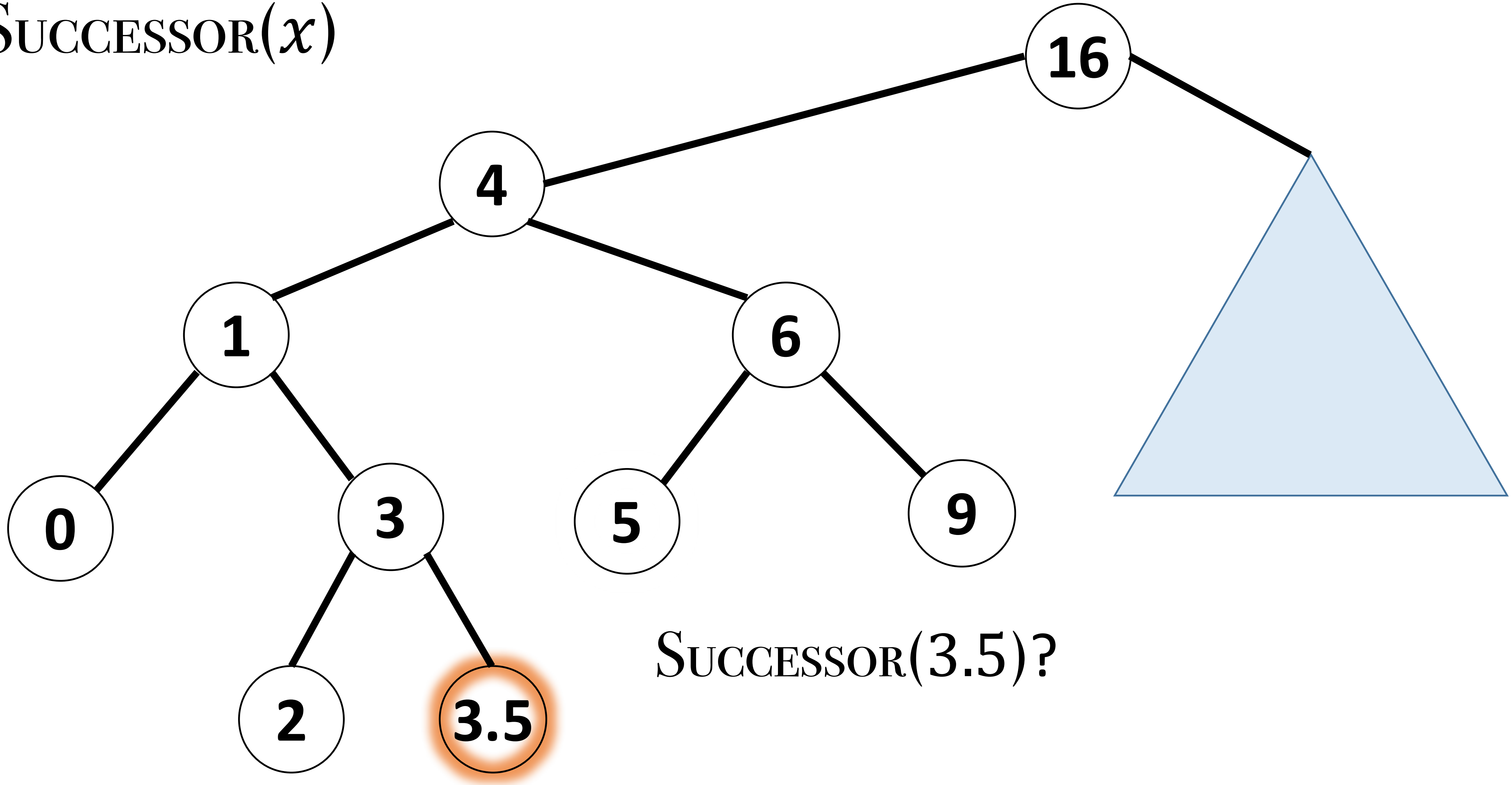
We can show that if x has a right child, the minimum of the right subtree is the successor.



In fact, if x has a right child, the minimum of the right subtree is first node that we visit after x in the in-order walk so it must be the successor.

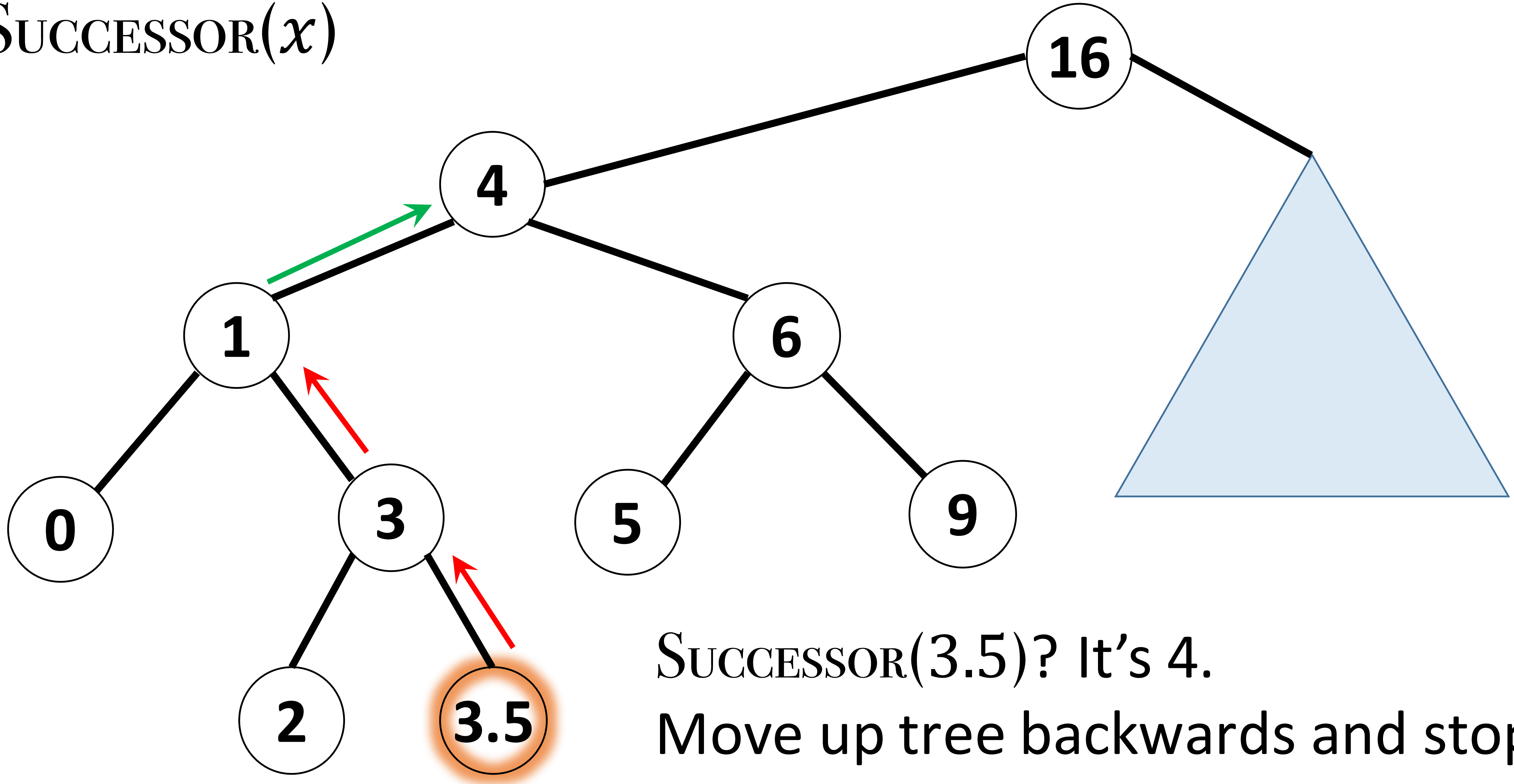
In-order walk: **A** 3 **B** 7 **C** 17 **D**

SUCCESSOR(x)



SUCCESSOR(3.5)?

SUCCESSOR(x)



SUCCESSOR(3.5)? It's 4.
Move up tree backwards and stop as soon as we go right.

SUCCESSOR(x)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

SUCCESSOR(x)

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

PREDECESSOR(x) can be implemented in a **symmetric** way. We just have to switch *right* with *left*, and TREE-MINIMUM with TREE-MAXIMUM.

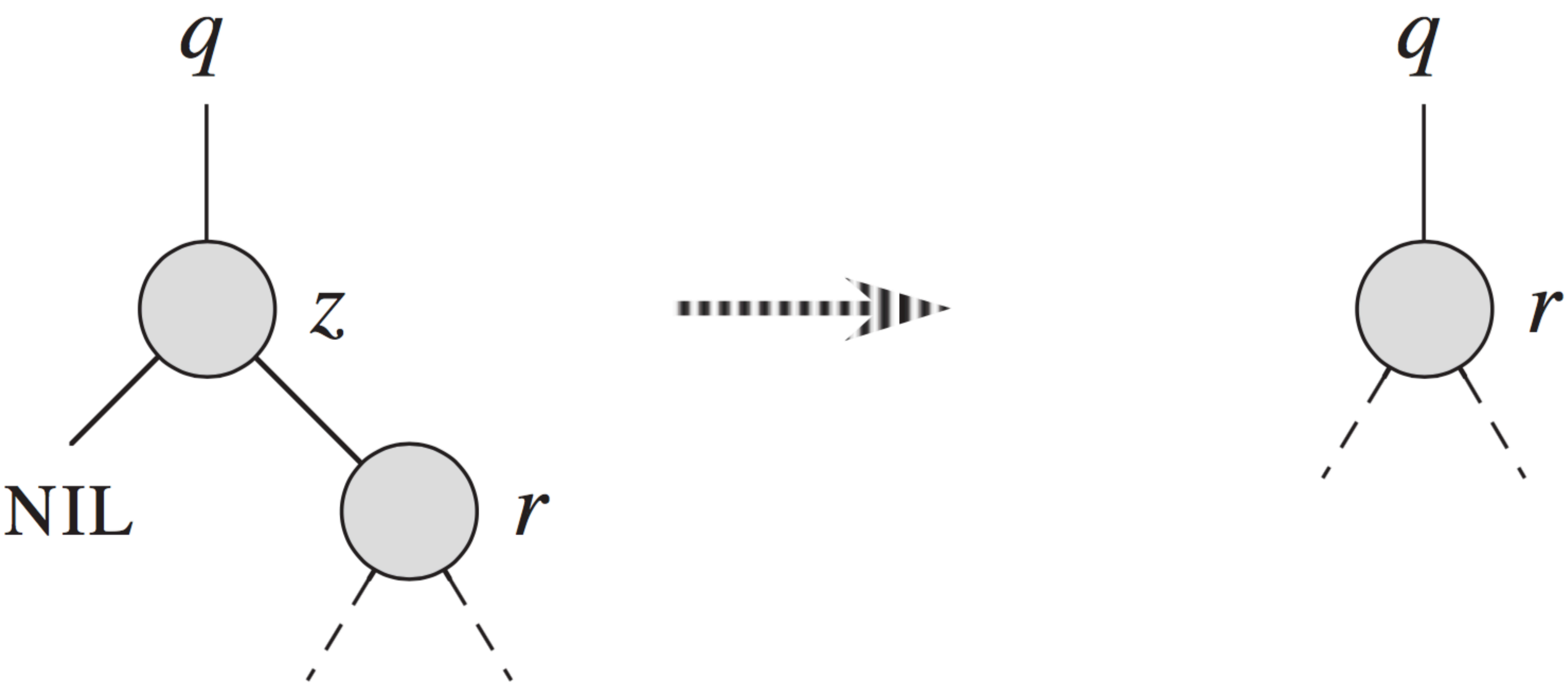
DELETE(z)

- There are **4 cases** that should be handled when deleting a node z.

- These cases depend on z's right and left subtrees

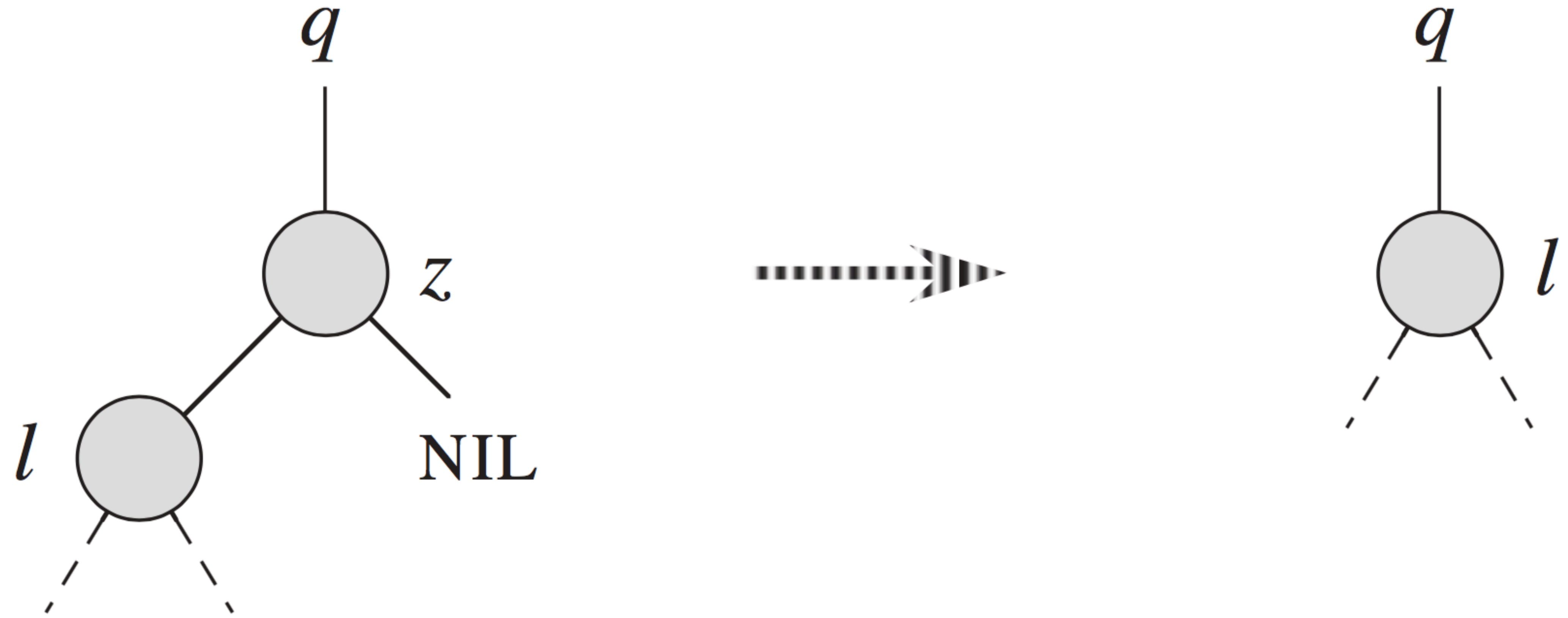
DELETE(z)

- **Case 1: if** $z.left$ == NIL



DELETE(z)

- **Case 2: elseif** $z.right$ == NIL



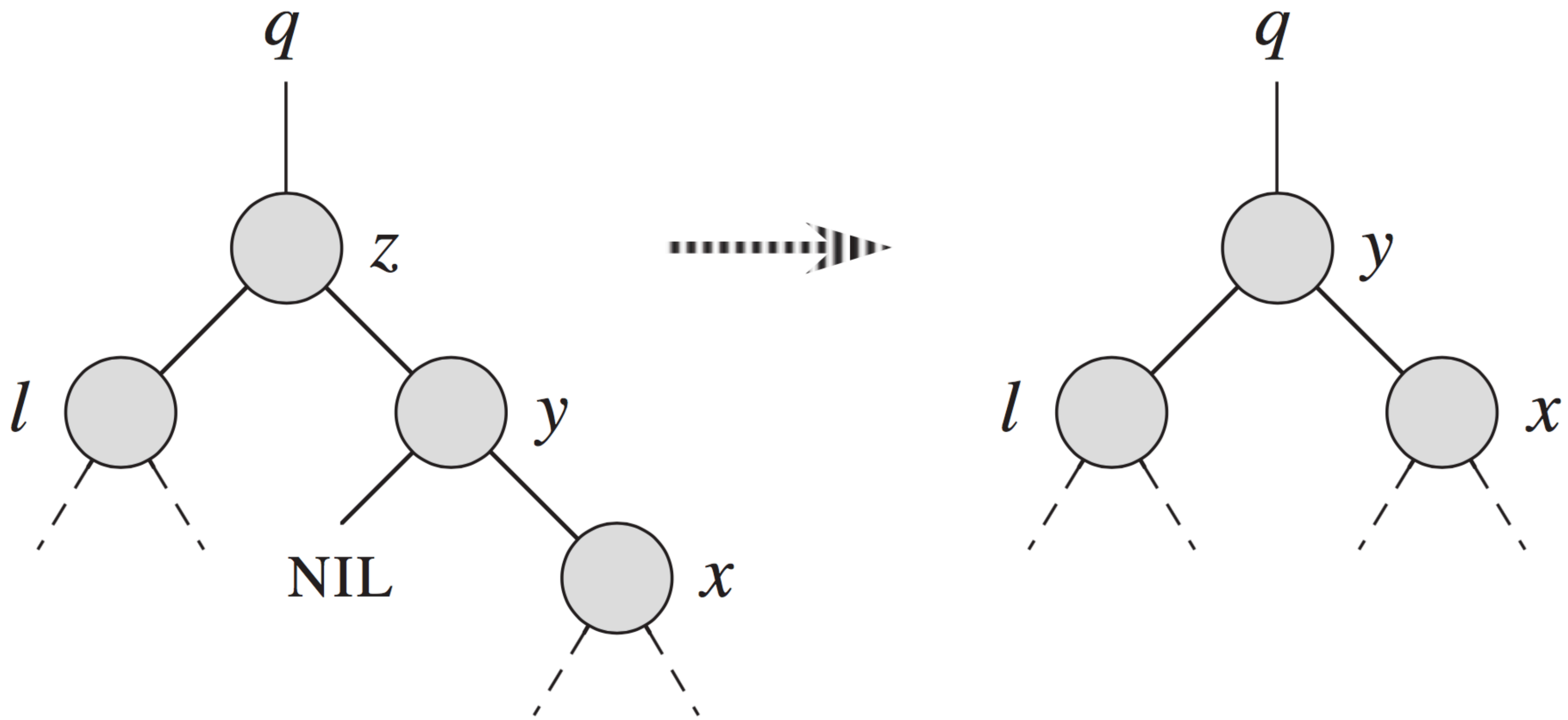
$\textsc{Delete}(z)$

- Else, both children of z are not NIL.

- For the next 2 cases, the idea is to **replace z with its successor y**

DELETE(z)

- **case 3:** $y.parent == z$



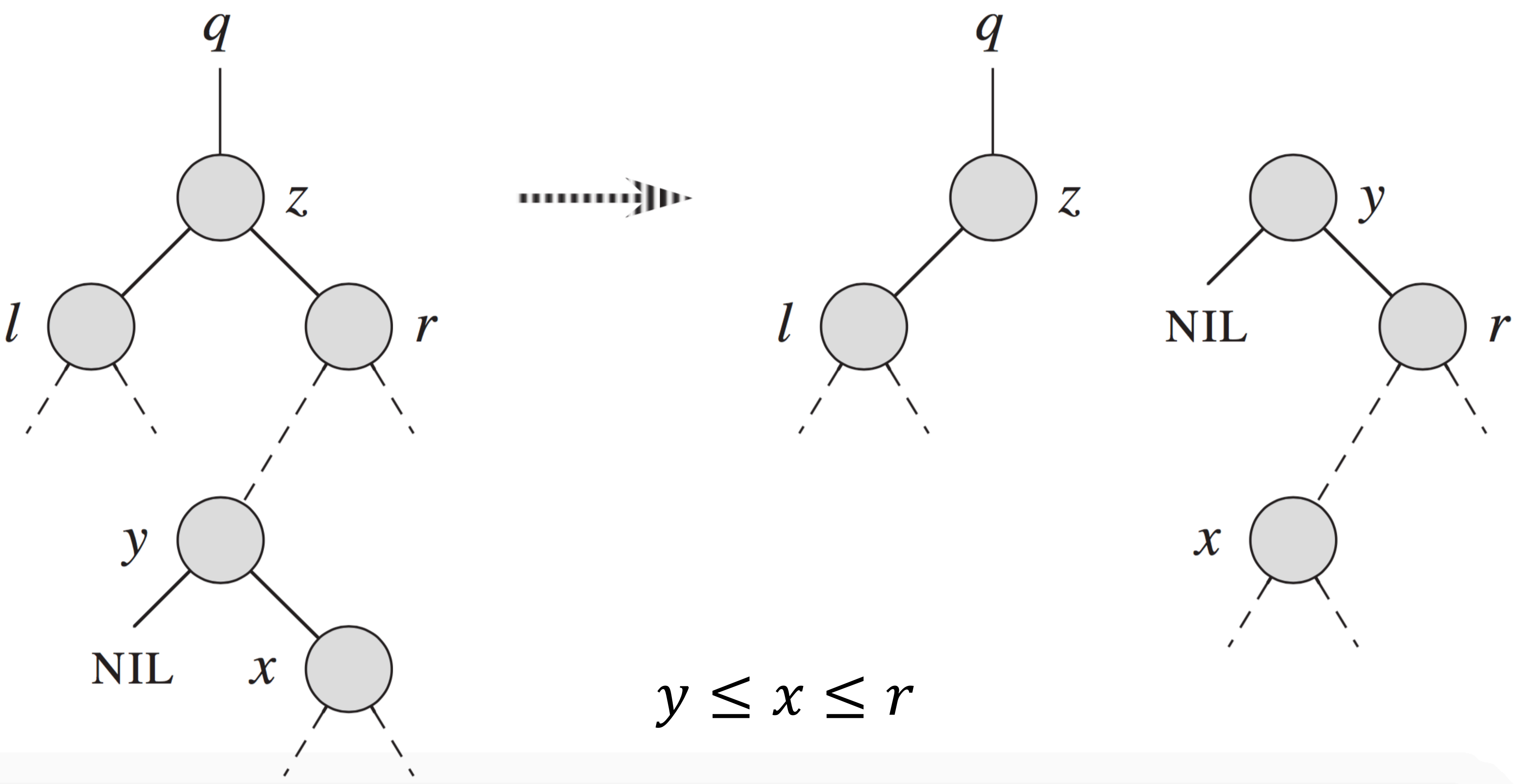
$l \le y \le x$

$\textsc{Delete}(z)$

- **A note on case 3 and 4:** It is guaranteed that y doesn’t have a left child because y is the successor of z which means y is the minimum in z’s right subtree, and the minimum node in a subtree cannot have a left child.

DELETE(z)

- **Case 4:** $y.parent \neq z$

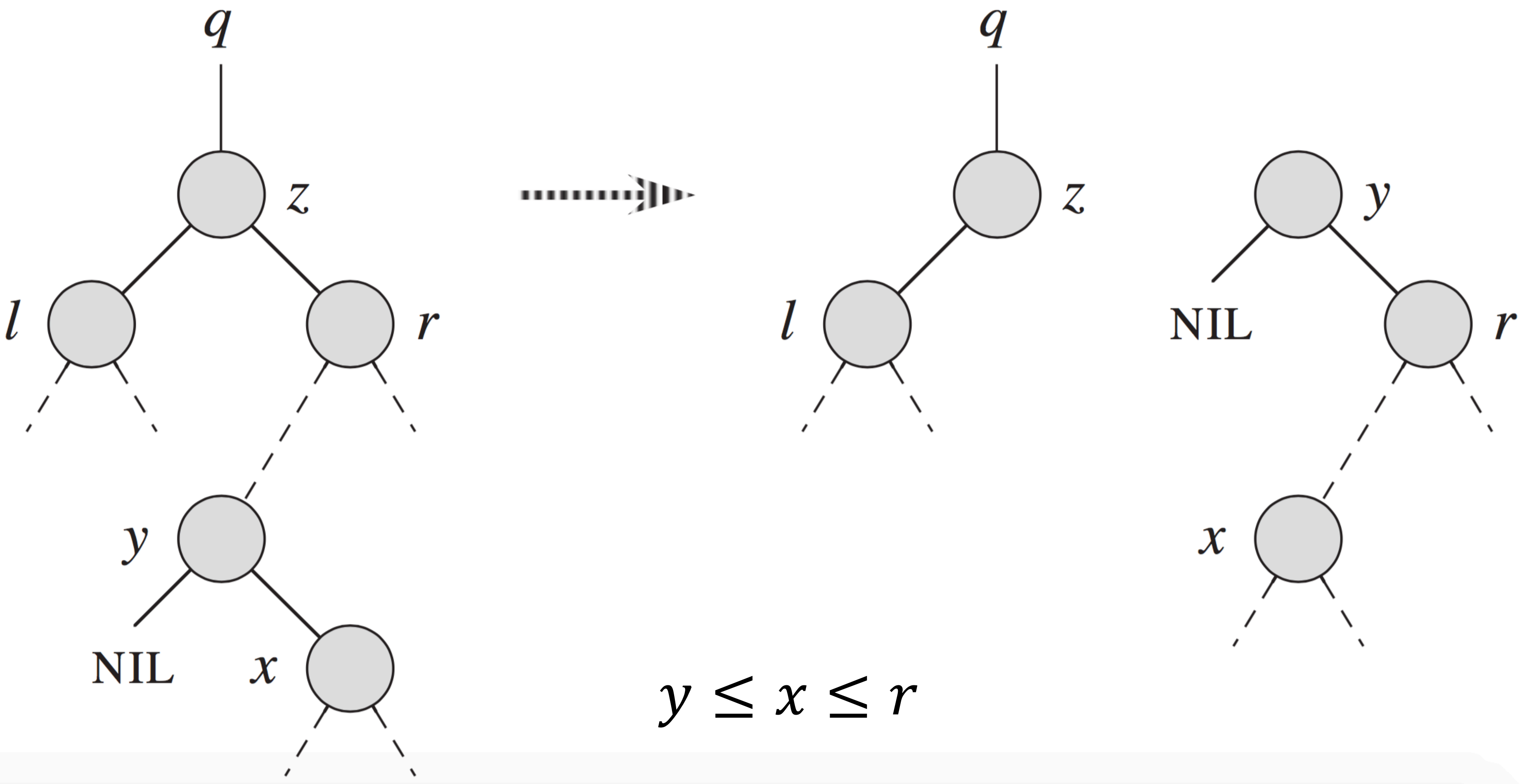


$y \leq x \leq r$

DELETE(z)

- **Case 4:** $y.parent \neq z$

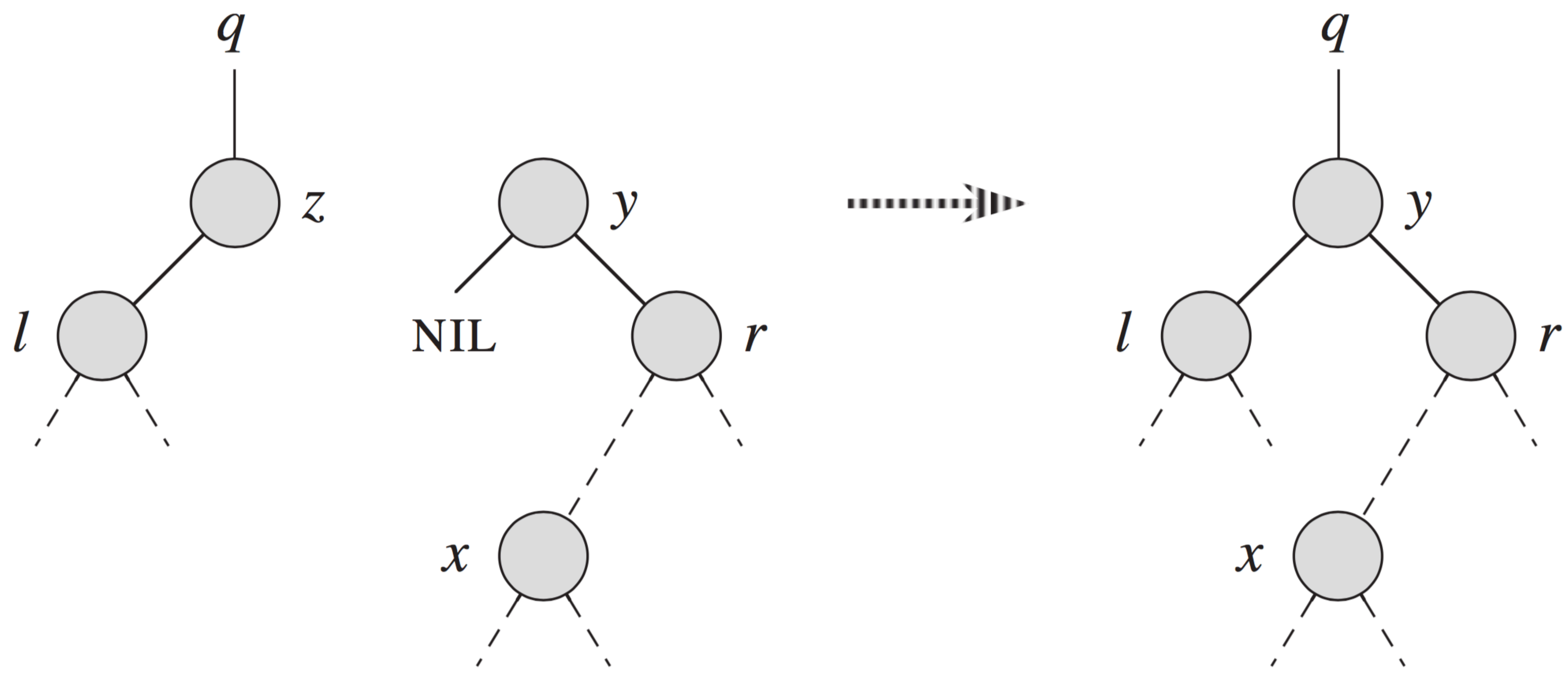
After this step, y doesn't have any parent; so, we can treat this like case 3.



$y \leq x \leq r$

DELETE(z)

- **Case 4:** $y.parent \neq z$



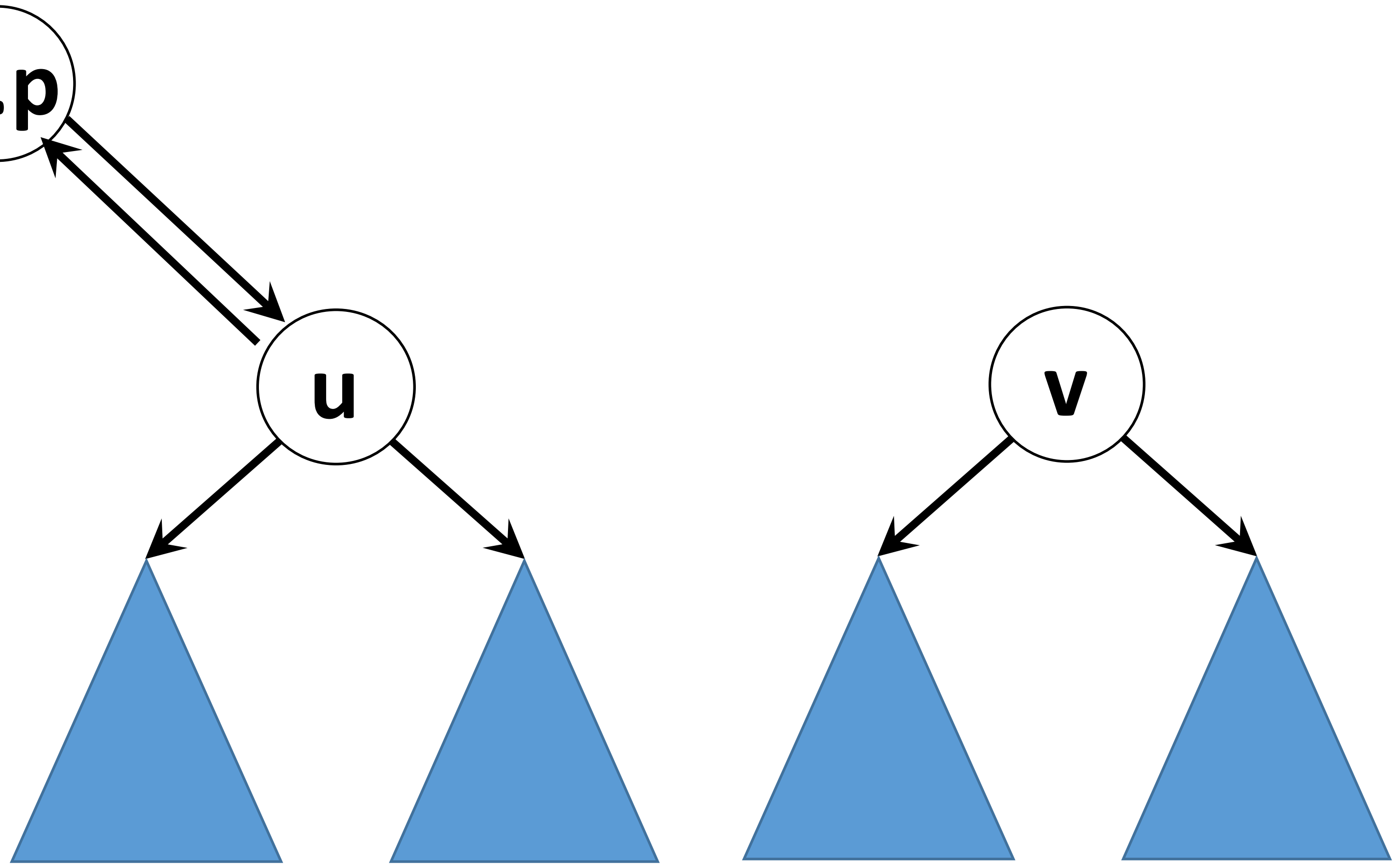
```
TREE-DELETE(T, z)
if z.left == NIL                                  case 1
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL                             case 2
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y                             case 4
    TRANSPLANT(T, z, y)
    y.left = z.left                               case 3
    y.left.p = y
```

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

- For more details on the pseudocode see section 12.3 in CLRS.

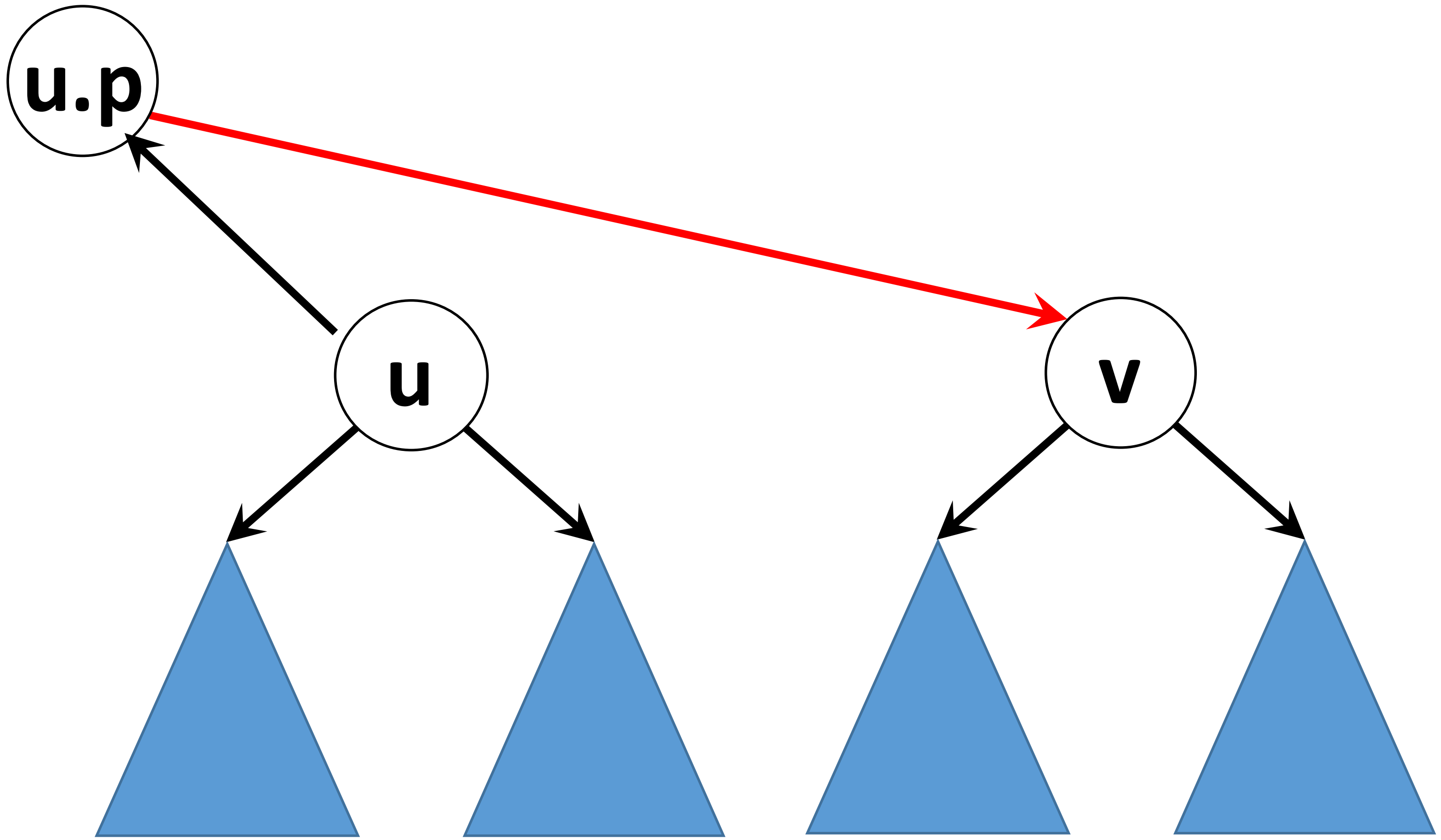
$\textsc{Transplant}(T, u, v)$

- Transplant replaces subtree rooted at u, with another subtree rooted at v, as a child of u's parent.



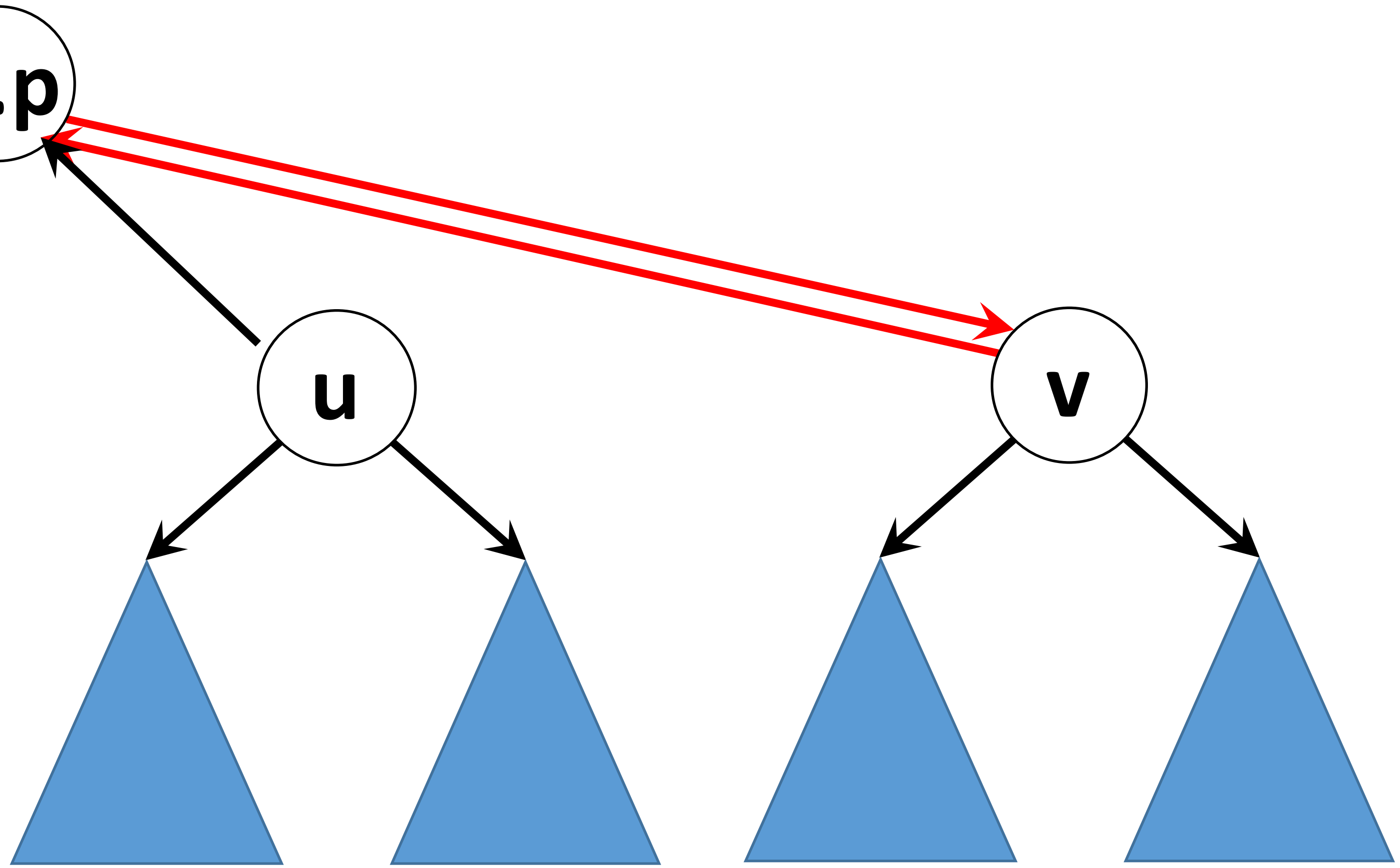
$\textsc{Transplant}(T, u, v)$

- Transplant replaces subtree rooted at u, with another subtree rooted at v, as a child of u's parent.



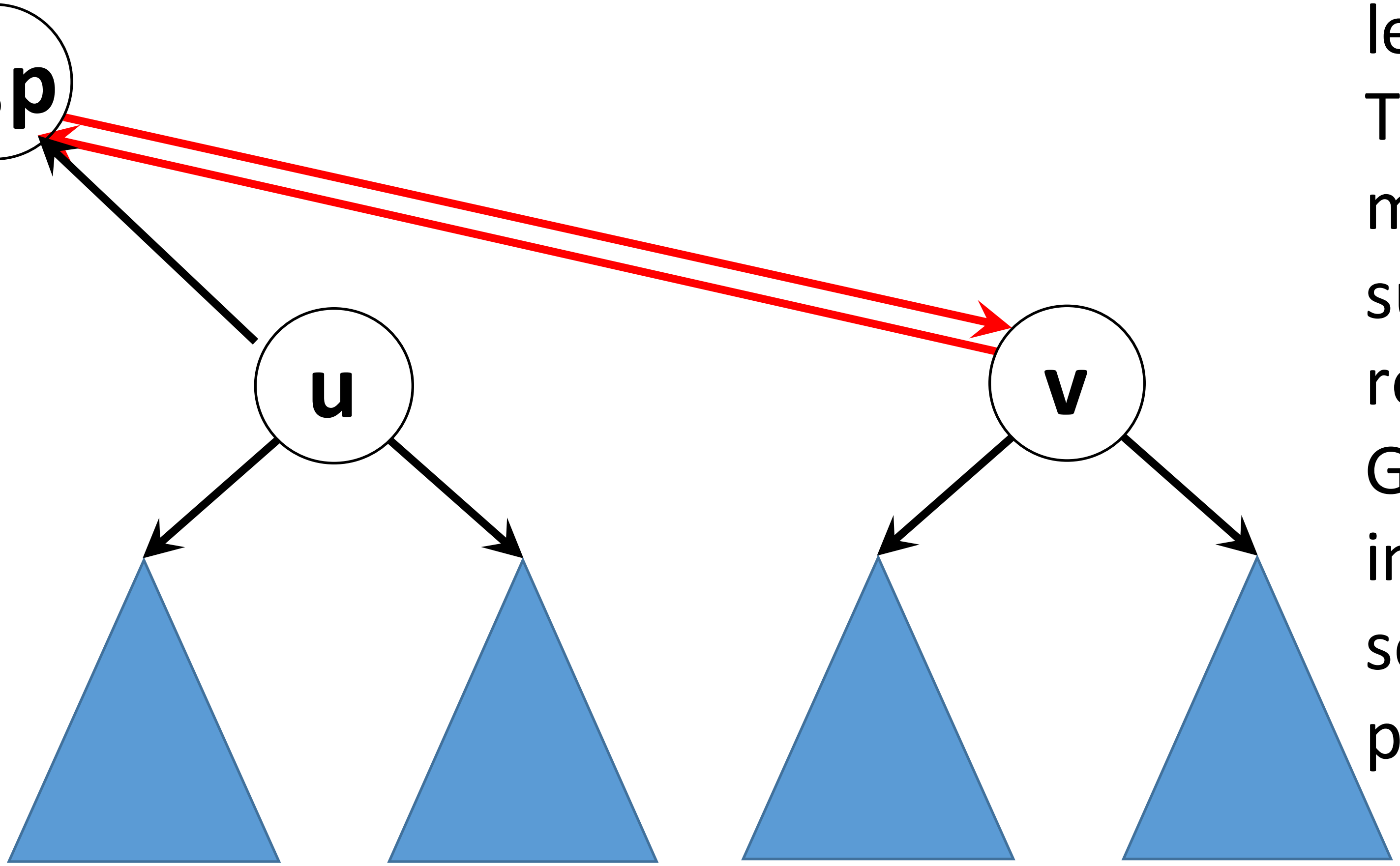
TRANSPLANT(T, u, v)

- Transplant replaces subtree rooted at u, with another subtree rooted at v, as a child of u's parent.



TRANSPLANT(T, u, v)

- Transplant replaces subtree rooted at u, with another subtree rooted at v, as a child of u's parent.



As soon as we leave the Transplant method, u and its subtrees will be removed (e.g. by Garbage Collector in Java), **unless** someone is still pointing to u

TREE-DELETE(T, z)

```
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Since variable y is still available, we can manipulate its pointers to subtrees and parent