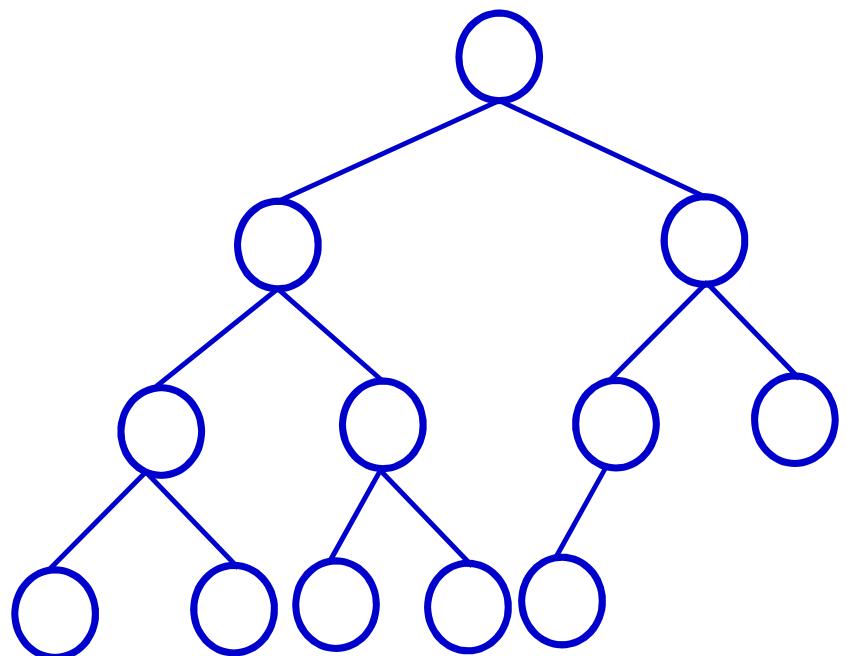


# CSE 105: Data Structures and Algorithms-I (Part 2)

Instructor

Dr Md Monirul Islam

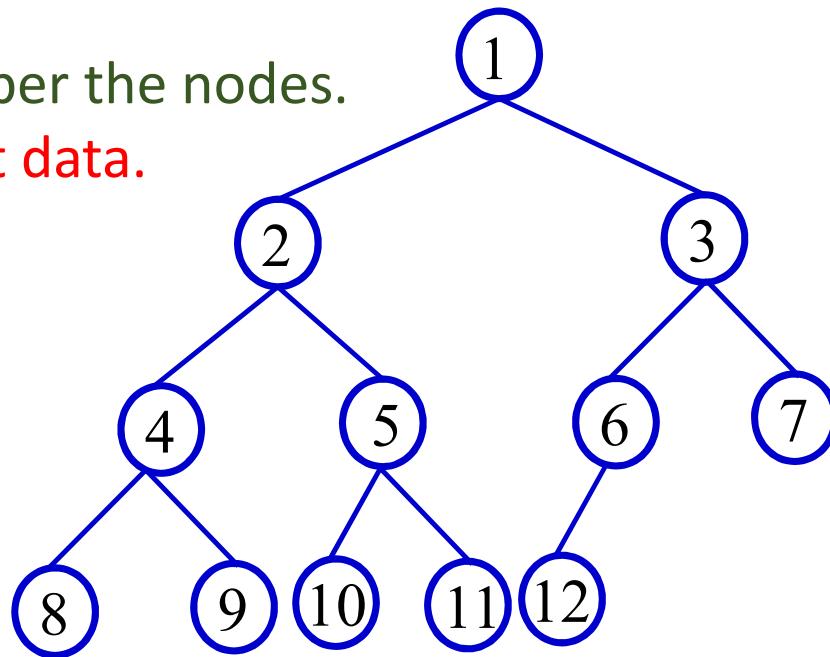
# Binary Tree Implementation: Complete Binary Tree



Complete Binary Tree

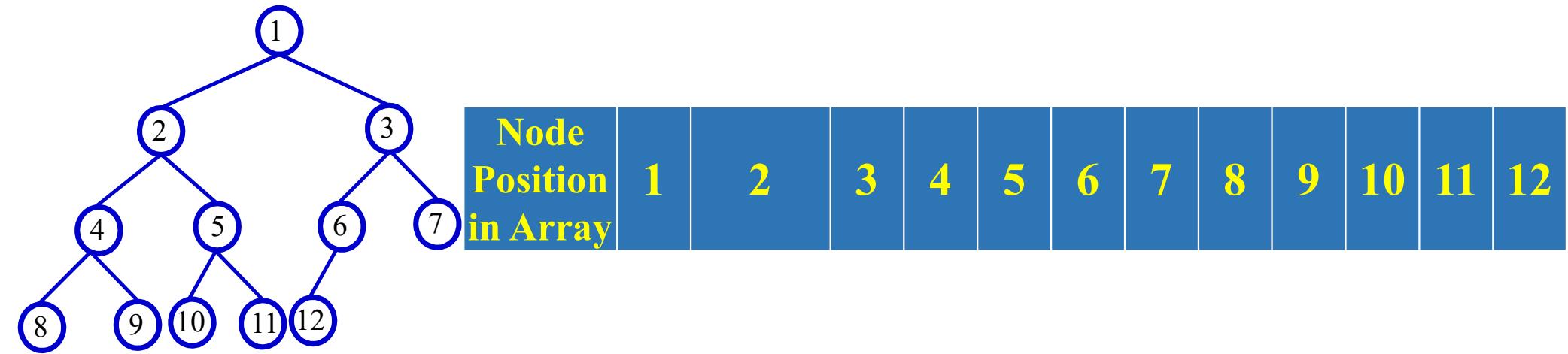
# Binary Tree Implementation: Complete Binary Tree

Let we number the nodes.  
They are not data.

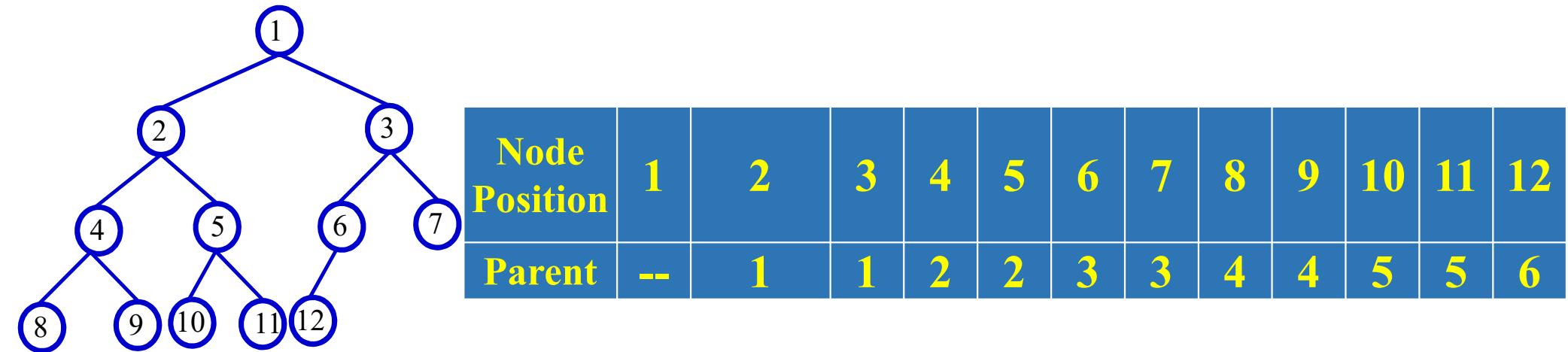


Complete Binary Tree

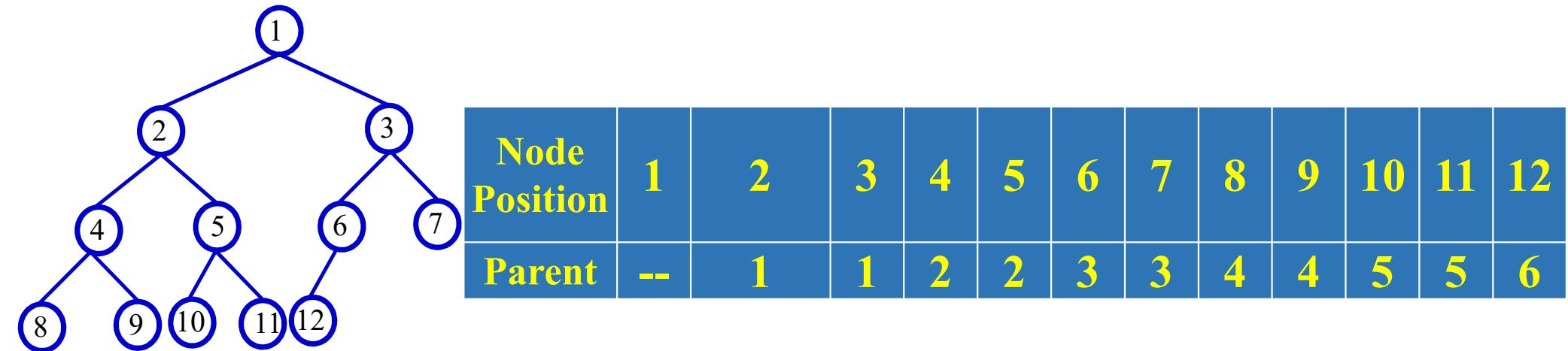
# Binary Tree Implementation: Complete Binary Tree



# Binary Tree Implementation: Complete Binary Tree

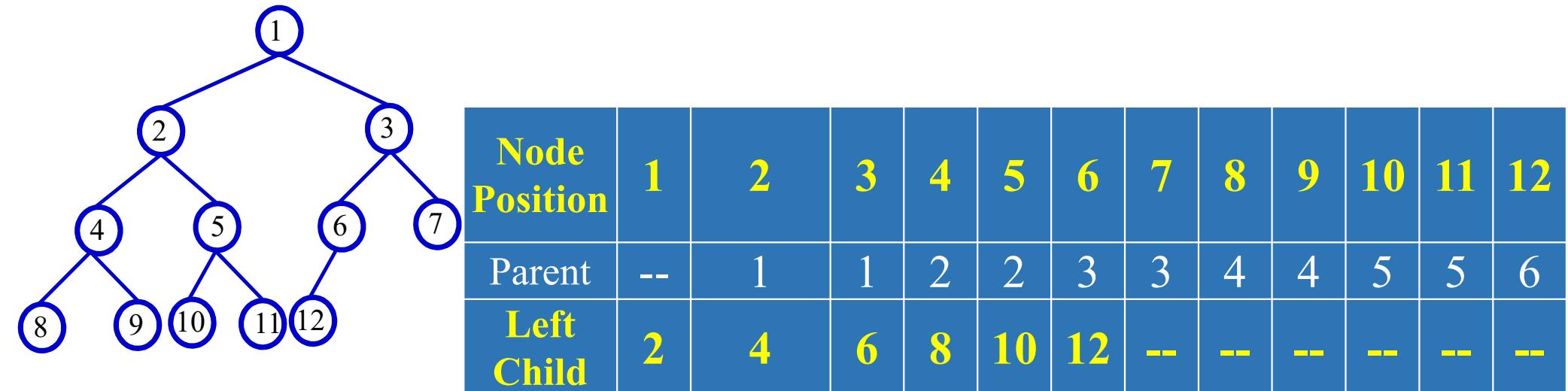


# Binary Tree Implementation: Complete Binary Tree



$\text{parent}(i) = \text{floor}(i/2);$

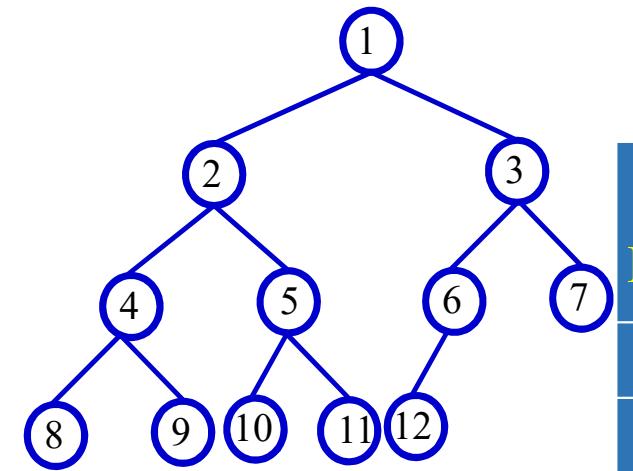
# Binary Tree Implementation: Complete Binary Tree



$\text{parent}(i) = \text{floor}(i/2);$

$\text{left}(i) = 2*i;$

# Binary Tree Implementation: Complete Binary Tree



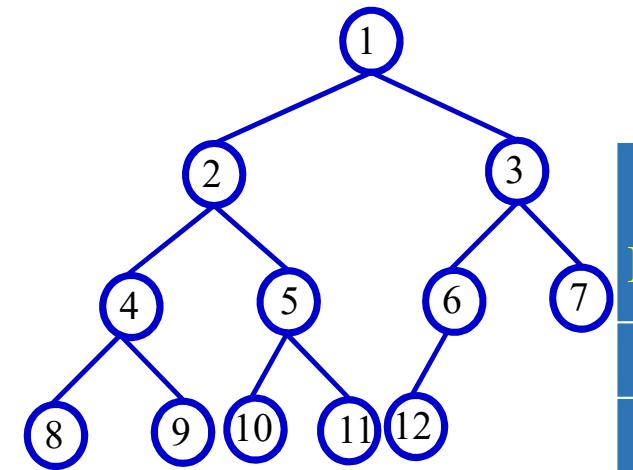
$\text{parent}(i) = \text{floor}(i/2);$

$\text{left}(i) = 2*i;$

$\text{right}(i) = 2*i + 1;$

Node Position	1	2	3	4	5	6	7	8	9	10	11	12
Parent	--	1	1	2	2	3	3	4	4	5	5	6
Left Child	2	4	6	8	10	12	--	--	--	--	--	--
Right Child	3	5	7	9	11	--	--	--	--	--	--	--

# Binary Tree Implementation: Complete Binary Tree



$\text{parent}(i) = \text{floor}(i/2);$

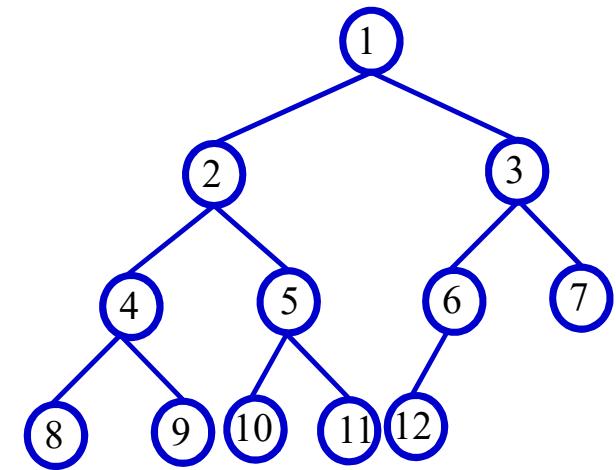
$\text{left}(i) = 2*i;$

$\text{right}(i) = 2*i + 1;$

$\text{leftSibling}(i) = i-1$ , if  $i$  is odd;

Node Position	1	2	3	4	5	6	7	8	9	10	11	12
Parent	--	1	1	2	2	3	3	4	4	5	5	6
Left Child	2	4	6	8	10	12	--	--	--	--	--	--
Right Child	3	5	7	9	11	--	--	--	--	--	--	--
Left Sibling	--	--	2	--	4	--	6	--	8	--	10	--

# Binary Tree Implementation: Complete Binary Tree



$\text{parent}(i) = \text{floor}(i/2);$

$\text{left}(i) = 2*i;$

$\text{right}(i) = 2*i + 1;$

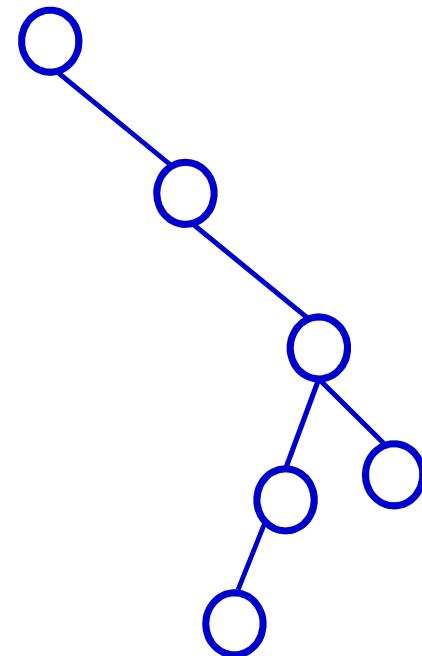
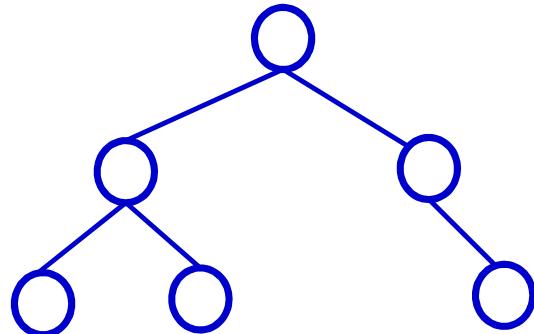
$\text{leftSibling}(i) = i-1$ , if  $i$  is odd;

$\text{rightSibling}(i) = i+1$ , if  $i$  is even;

Node Position	1	2	3	4	5	6	7	8	9	10	11	12
Parent	--	1	1	2	2	3	3	4	4	5	5	6
Left Child	2	4	6	8	10	12	--	--	--	--	--	--
Right Child	3	5	7	9	11	--	--	--	--	--	--	--
Left Sibling	--	--	2	--	4	--	6	--	8	--	10	--
Right Sibling	--	3	--	5	--	7	--	9	--	11	--	--

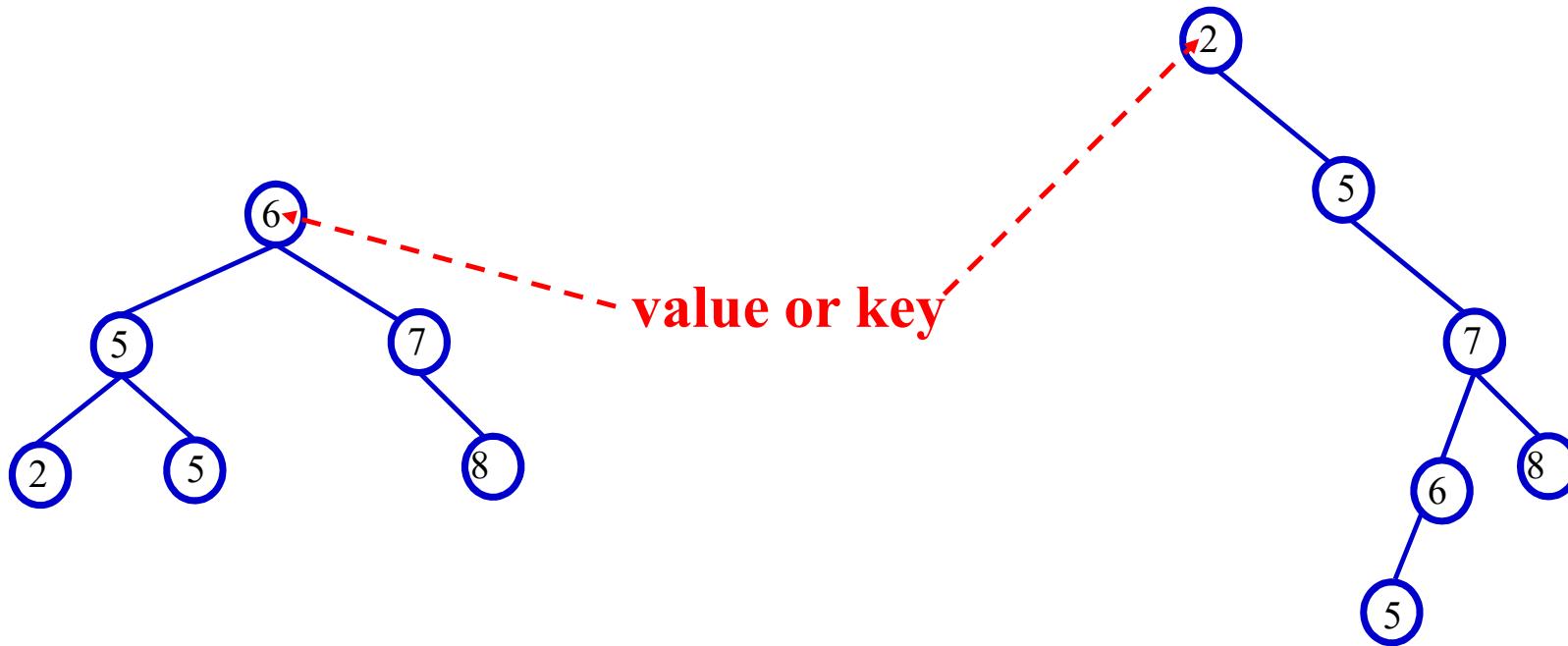
# Binary Search Tree

- A Binary tree
- Three pointers in each node: **left**, **right**, **parent**
- **Stores value or key**



# Binary Search Tree

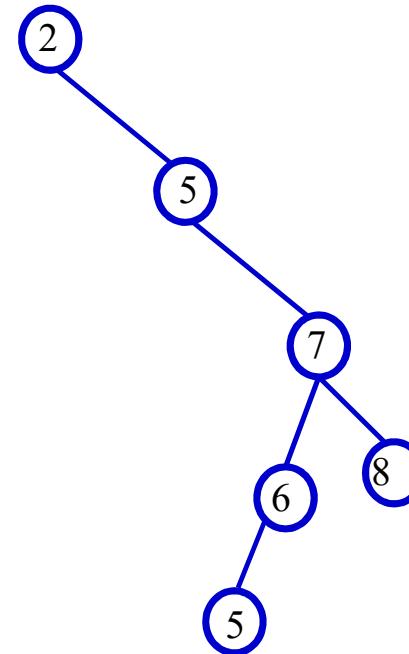
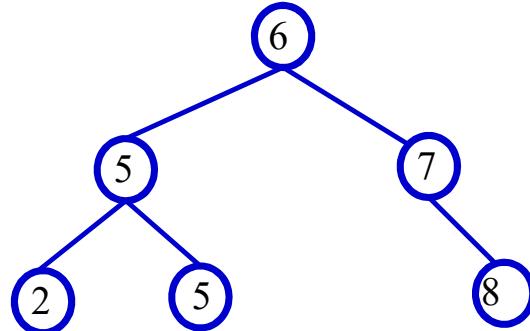
- A Binary tree
- Three pointers in each node: **left**, **right**, **parent**
- Maintains a special property for each node **Binary Search Tree property**



# Binary Search Tree

## BST property

All elements stored in the left subtree of a node with value  $K$  have values  $\leq K$ .  
All elements stored in the right subtree of a node with value  $K$  have values  $\geq K$ .



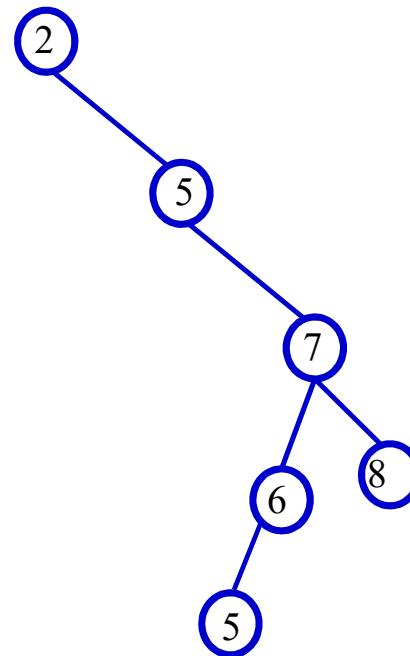
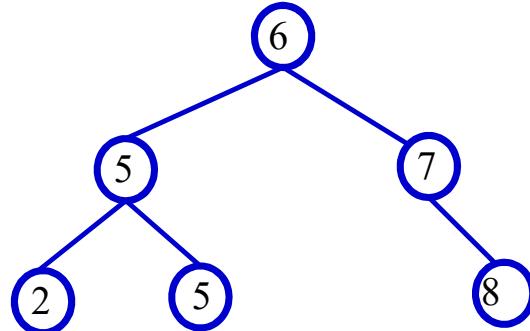
# Binary Search Tree

## BST property

Let  $x$  be a node in a binary search tree.

If  $y$  is a node in the left subtree of  $x$ , then  $y.key \leq x.key$

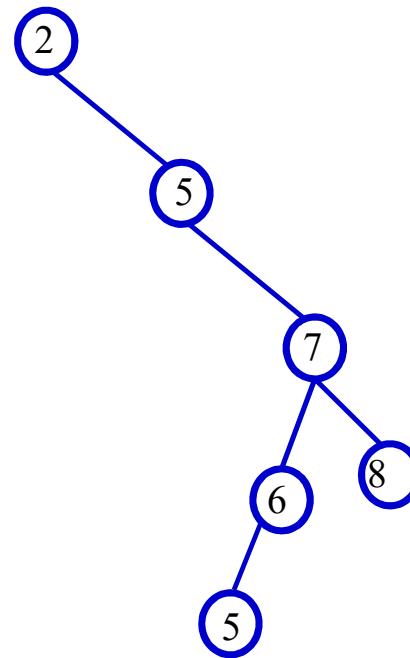
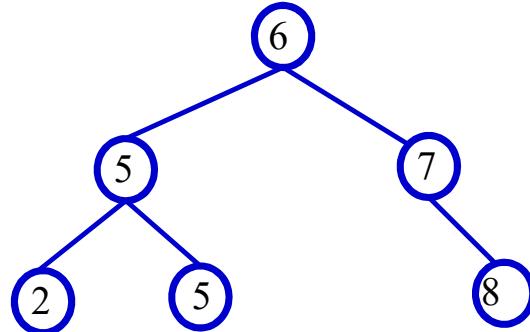
If  $y$  is a node in the right subtree of  $x$ , then  $y.key \geq x.key$ .



# Binary Search Tree

Inorder traversal of a BST

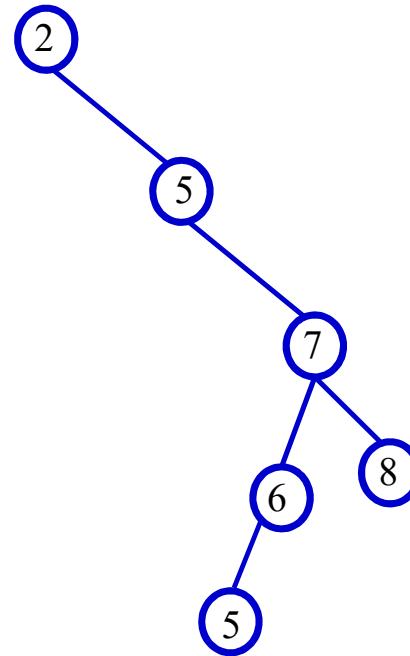
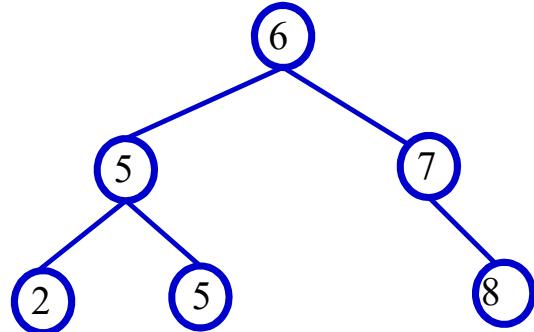
Traversal Outcome:?



# Binary Search Tree

Inorder traversal of a BST

Traversal Outcome: 2 5 5 6 7 8

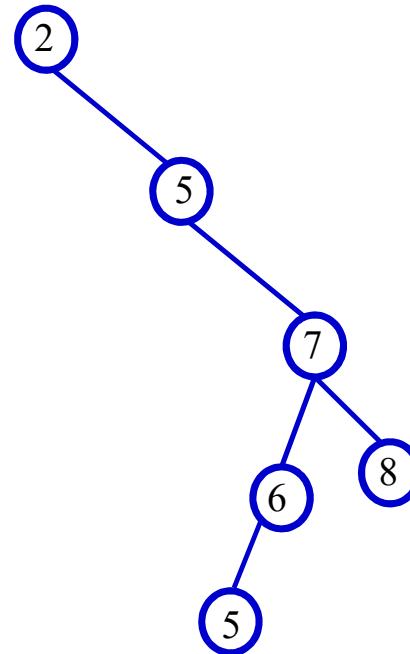
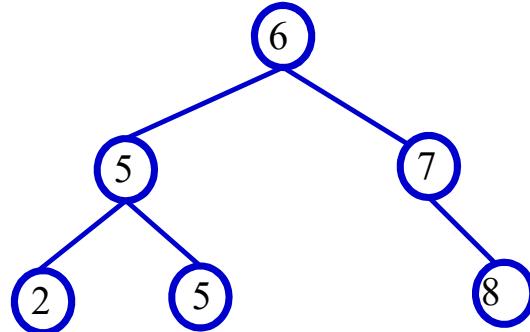


# Binary Search Tree

## Inorder traversal of a BST

Traversal Outcome: 2 5 5 6 7 8

Same list of keys but different BST shape.

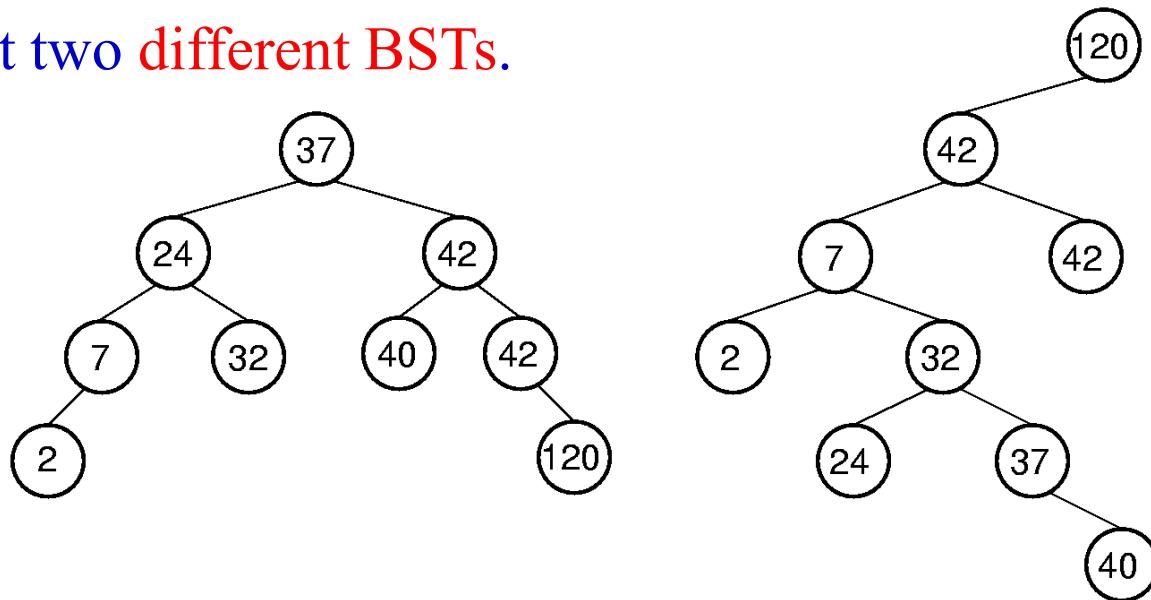


# Binary Search Tree

## Another Example

**Traversal Outcome:** 2, 7, 24, 32, 37, 40, 42, 42, 120

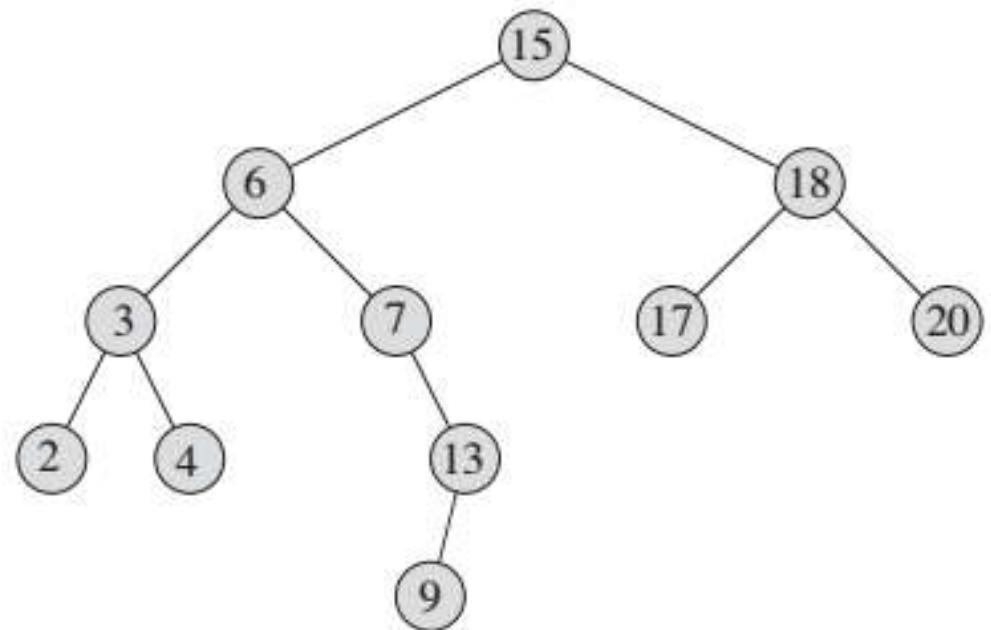
We also get two different BSTs.



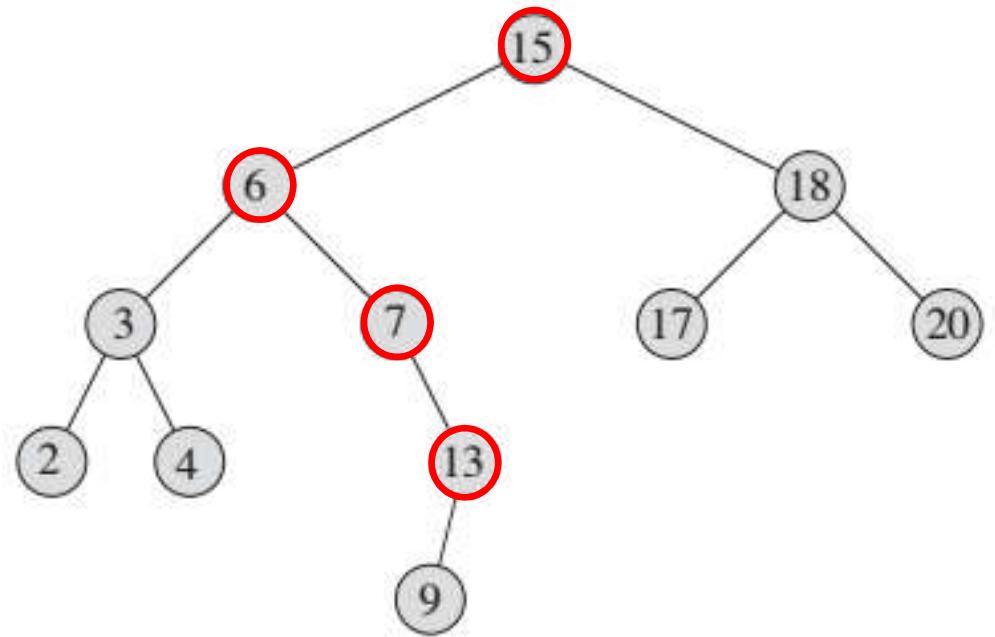
# BST Operations

- Search for a key
- Minimum
- Maximum
- Successor
- Predecessor
- Insert
- Delete

# BST Operation: Search

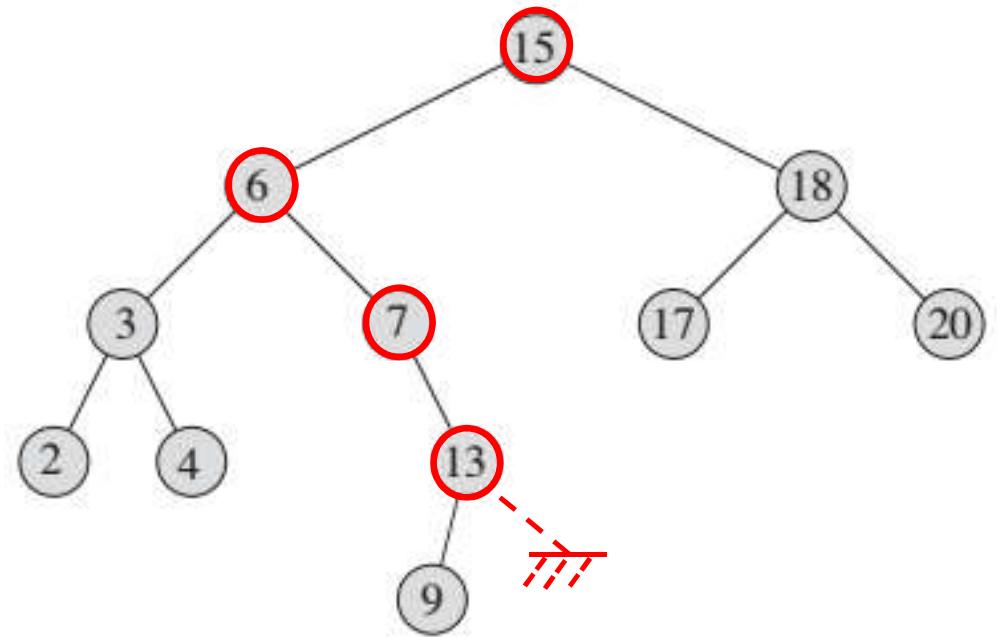


# BST Operation: Search



Search for 13

# BST Operation: Search



Search for 14

# BST Operation: Search

```
TREE_SEARCH ( $x, k$ )
```

```
1 if  $x == \text{NULL}$  or  $k == x->\text{key}$ 
```

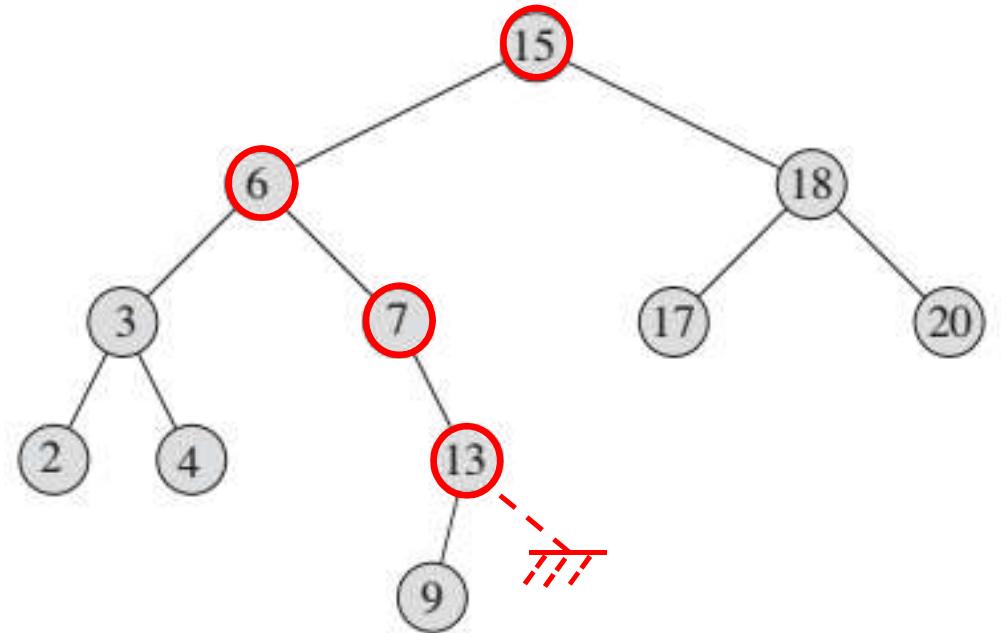
```
2 return  $x$ 
```

```
3 if  $k < x->\text{key}$ 
```

```
4 return TREE_SEARCH( $x->\text{left}, k$ )
```

```
5 else return TREE_SEARCH( $x->\text{right}, k$ )
```

**Search for 14**



# BST Operation: Search

TREE\_SEARCH ( $x, k$ )

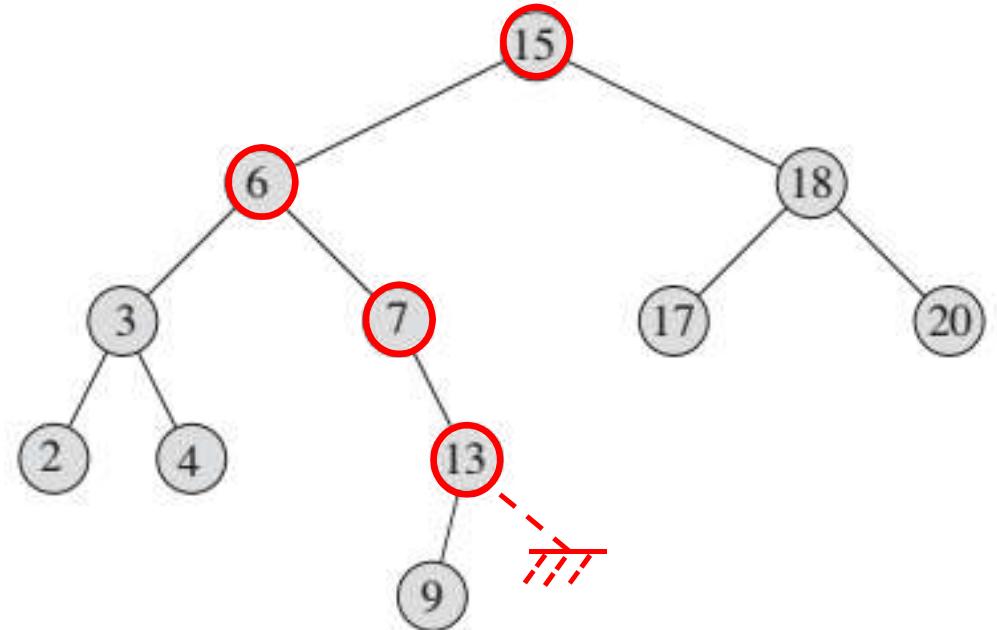
1 **if**  $x == \text{NULL}$  **or**  $k == x->\text{key}$

2 **return**  $x$

3 **if**  $k < x->\text{key}$

4 **return** TREE\_SEARCH( $x->\text{left}, k$ )

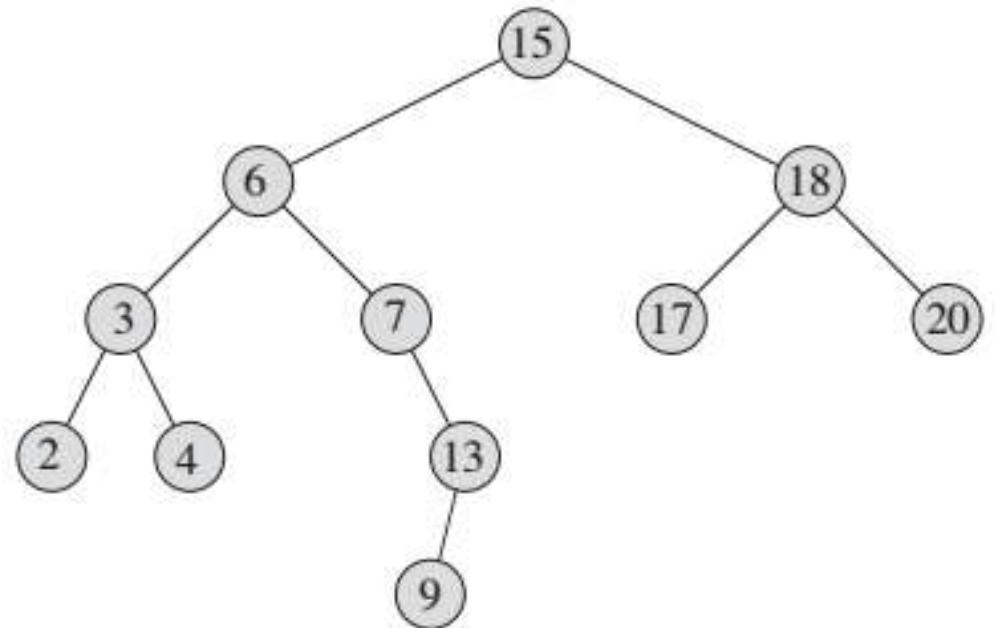
5 **else return** TREE\_SEARCH( $x->\text{right}, k$ )



**Complexity:**  $O(h)$

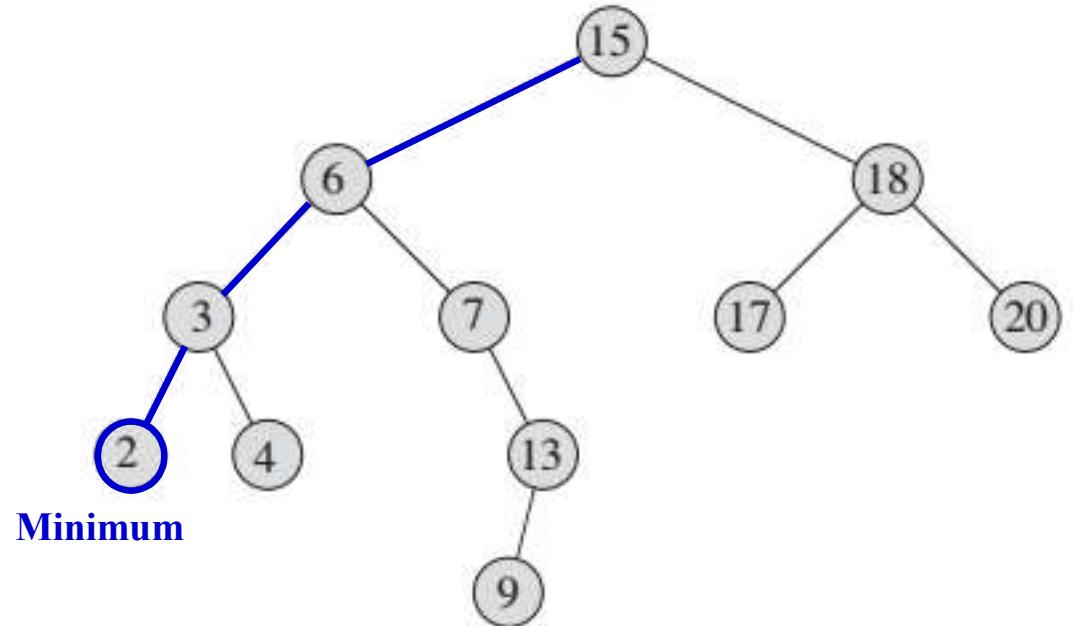
# BST Operation: Minimum

Where?



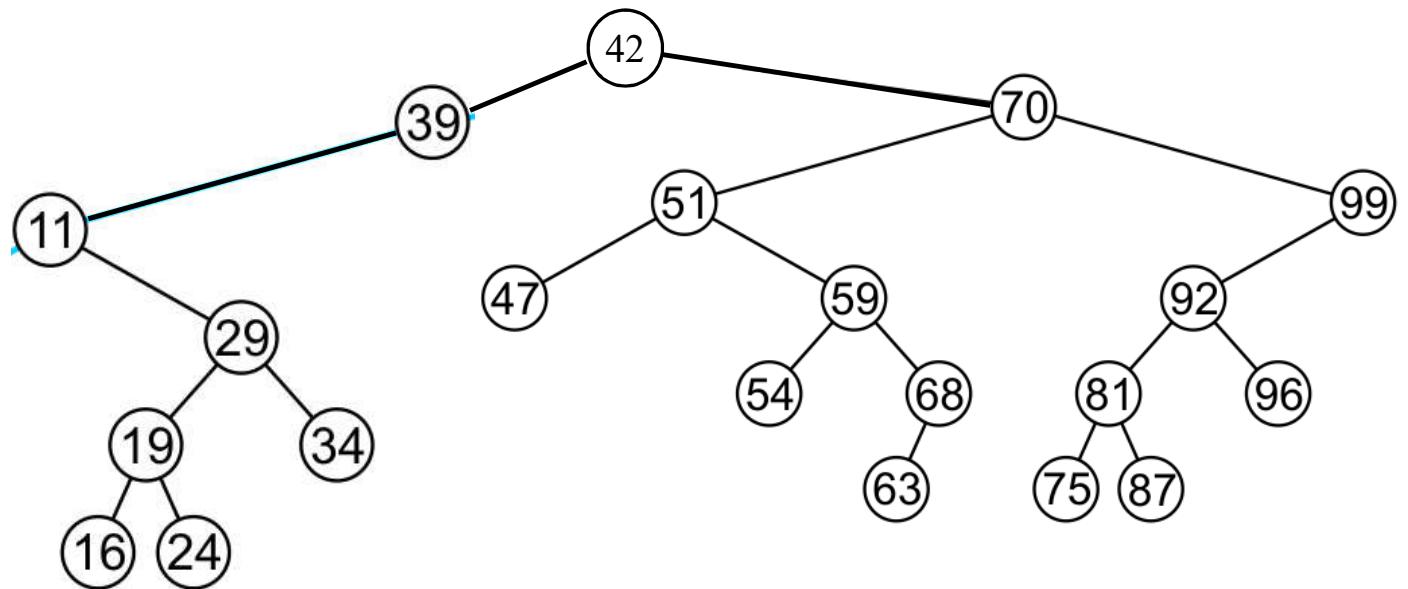
# BST Operation: Minimum

Must be in the left subtree



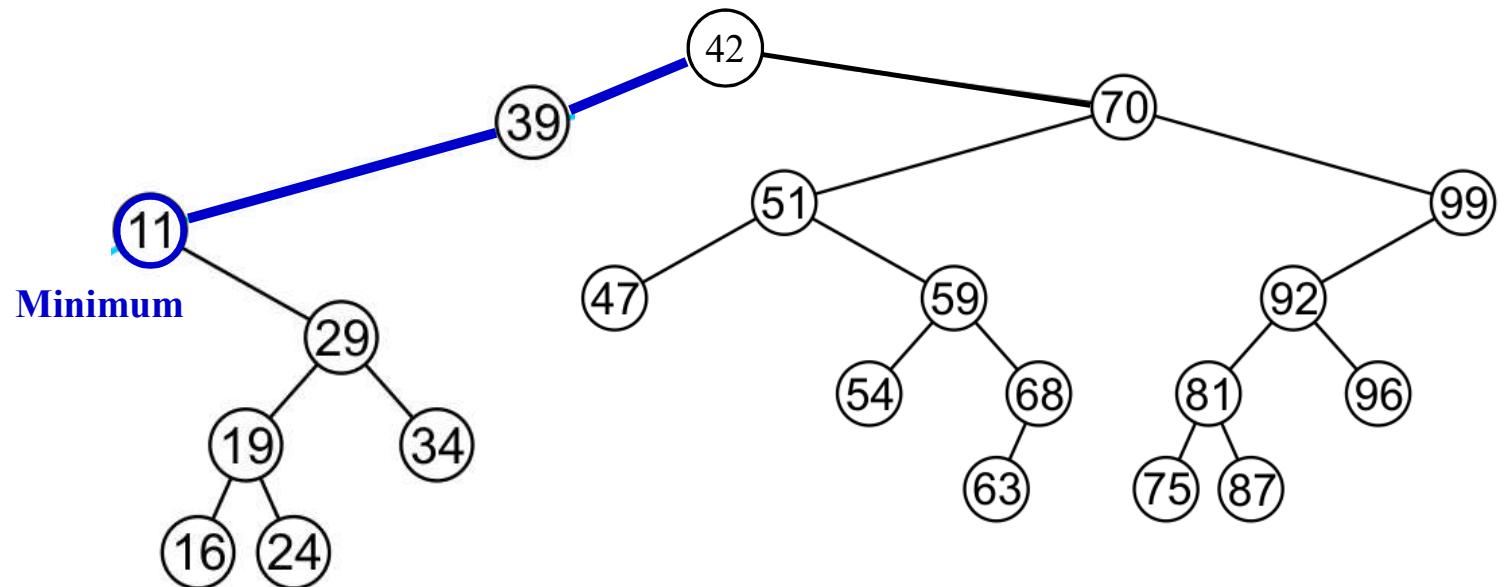
# BST Operation: Minimum

Not necessarily in the leaf node



# BST Operation: Minimum

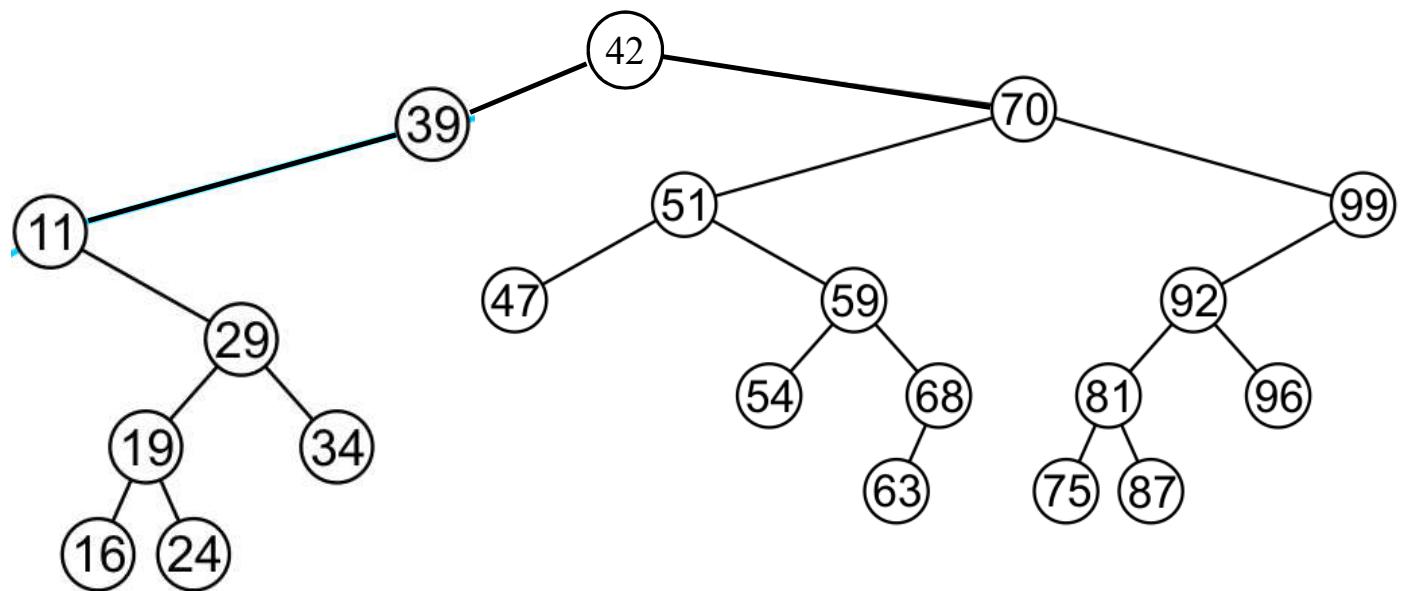
Not necessarily in the leaf node



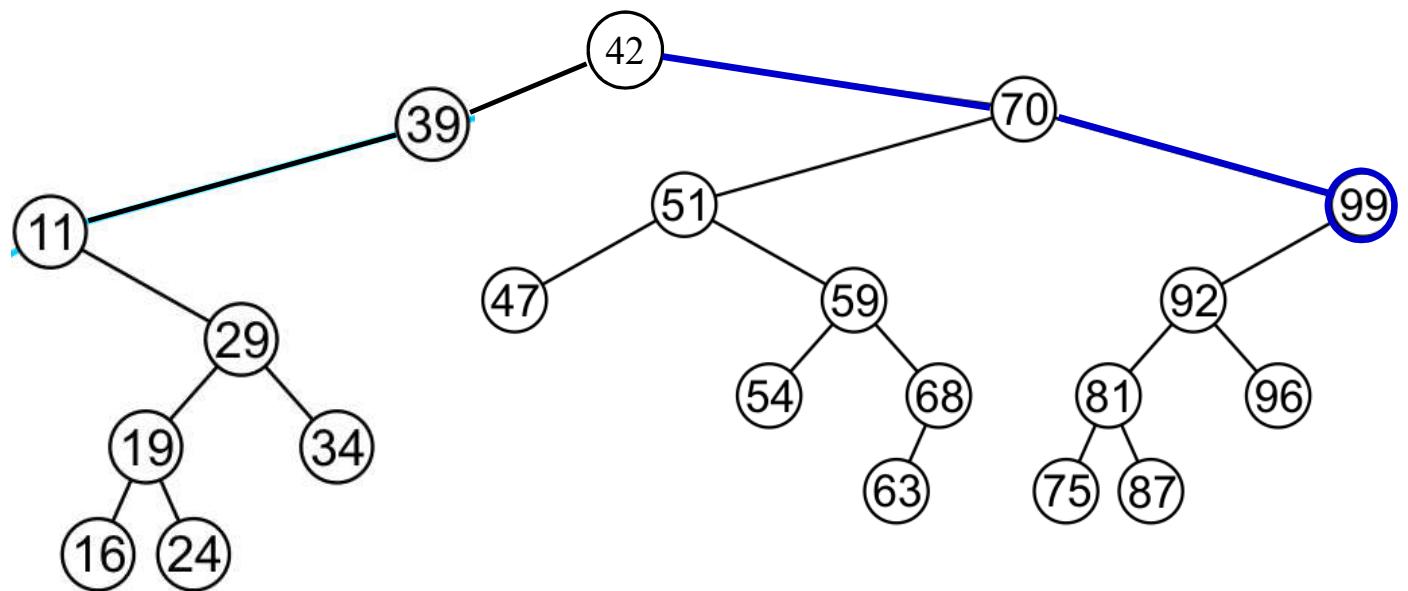
# BST Operation: Minimum

TREE\_MINIMUM ( $x$ )

```
1 if  $x == \text{NULL}$  return  $\text{NULL}$ 
2 while  $x->left \neq \text{NULL}$ 
3    $x = x->left$ 
4 return  $x$ 
```



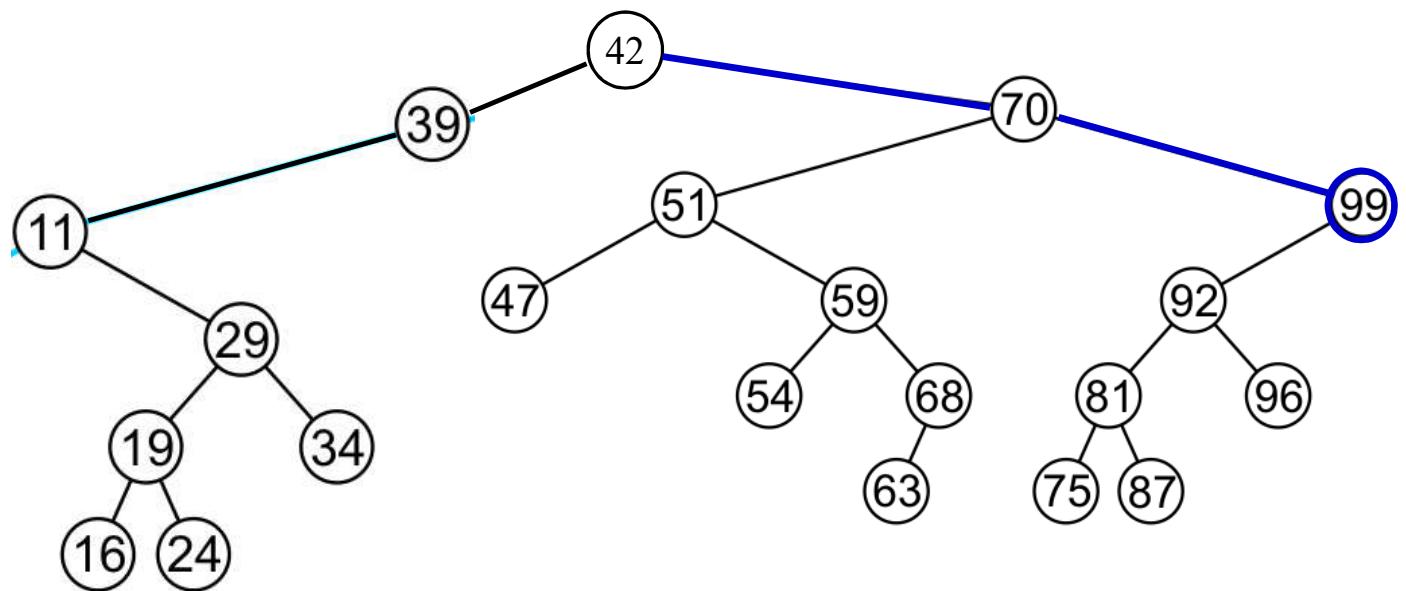
# BST Operation: Maximum



# BST Operation: Maximum

TREE\_MAXIMUM ( $x$ )

```
1 if  $x == \text{NULL}$  return  $\text{NULL}$ 
2 while  $x->right \neq \text{NULL}$ 
3    $x = x->right$ 
4 return  $x$ 
```

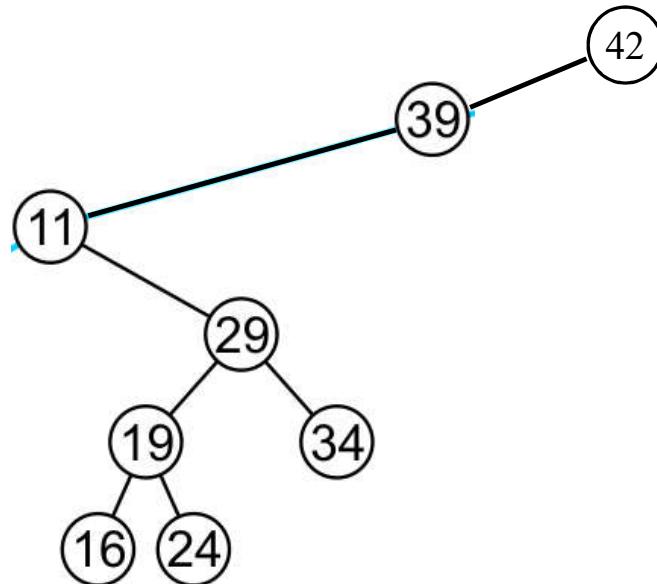


# BST Operation: Maximum

TREE\_MAXIMUM ( $x$ )

```
1 if  $x == \text{NULL}$  return  $\text{NULL}$ 
2 while  $x->right \neq \text{NULL}$ 
3    $x = x->right$ 
4 return  $x$ 
```

The algorithm  
works in this case  
too.



# BST Operation: Minimum and Maximum

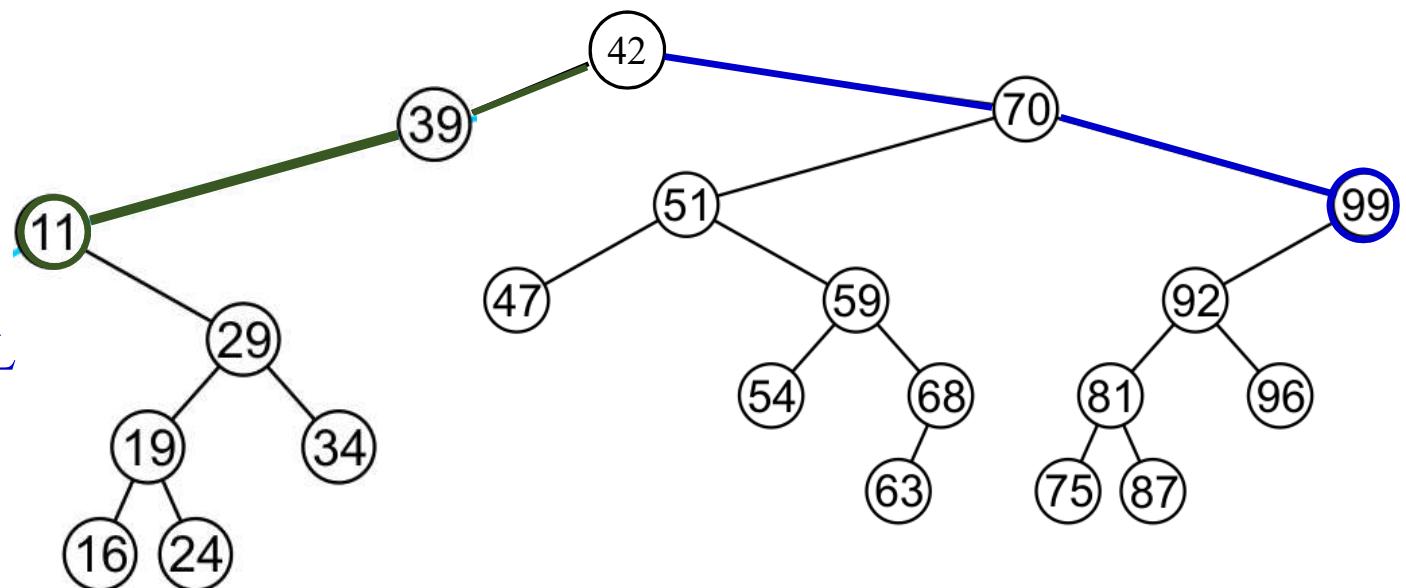
```
TREE_MINIMUM ( $x$ )
```

```
1 if  $x == \text{NULL}$  return  $\text{NULL}$ 
2 while  $x->left \neq \text{NULL}$ 
3    $x = x->left$ 
4 return  $x$ 
```

```
TREE_MAXIMUM ( $x$ )
```

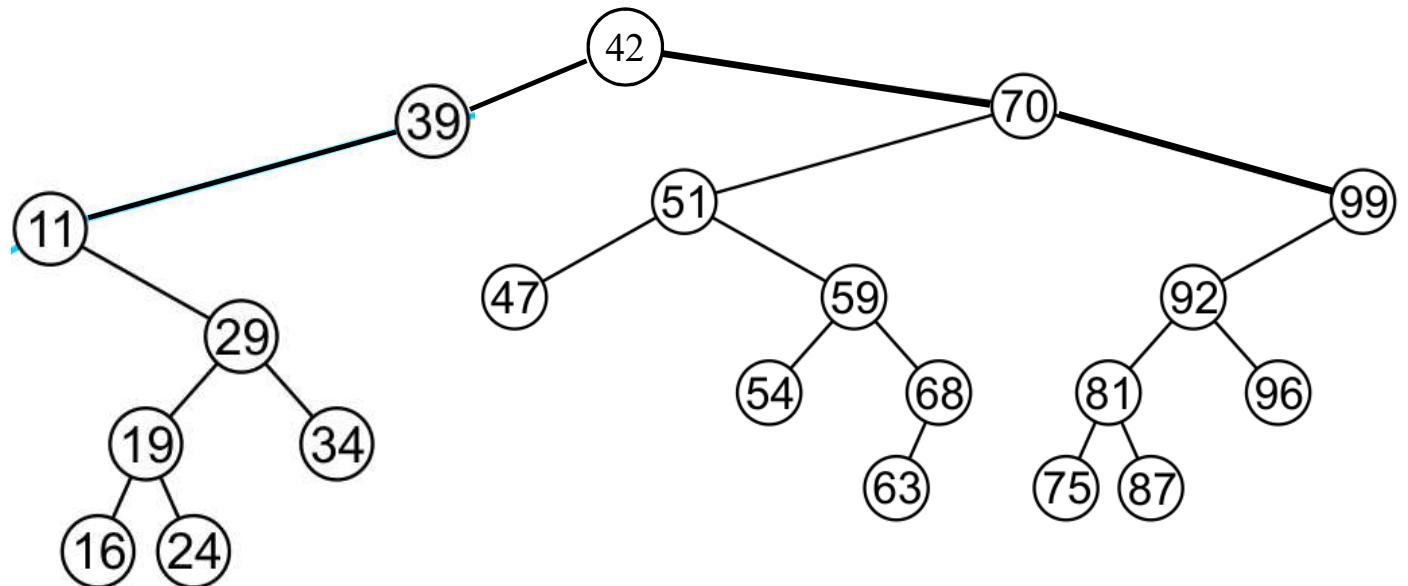
```
1 if  $x == \text{NULL}$  return  $\text{NULL}$ 
2 while  $x->right \neq \text{NULL}$ 
3    $x = x->right$ 
4 return  $x$ 
```

Complexity:  $O(h)$



# BST Operation: Successor

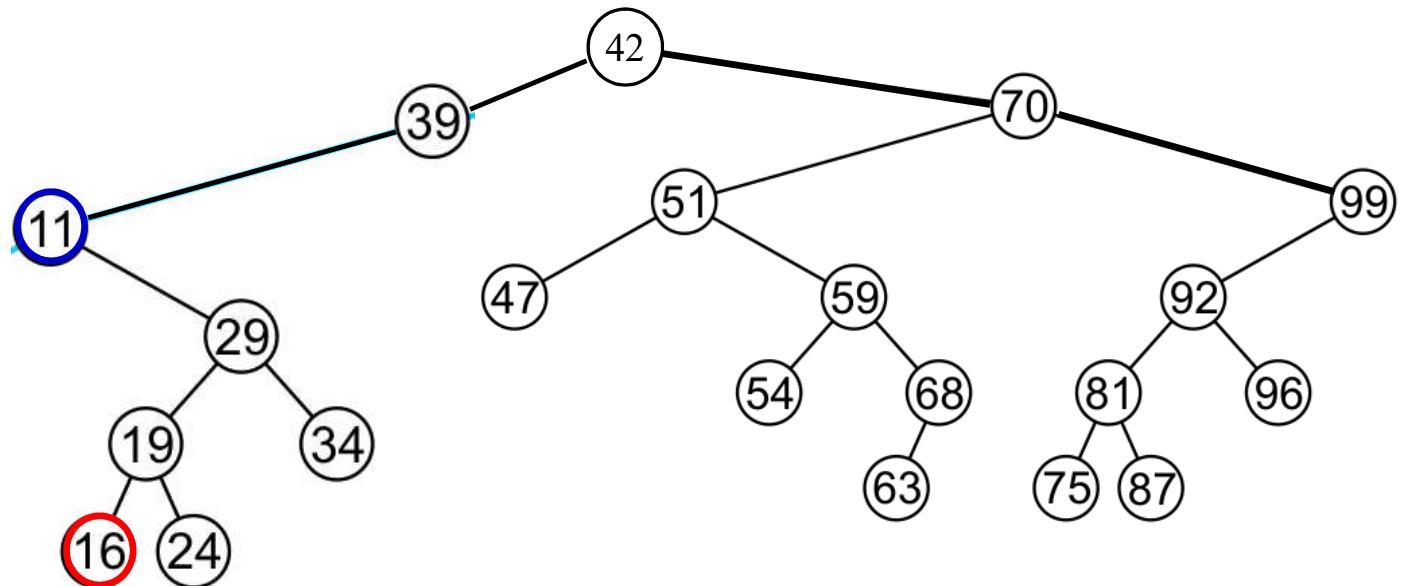
**successor of a node  $x$**  : the node with the **smallest key** greater than  $x.key$



# BST Operation: Successor

successor of a node  $x$  : the node with the smallest key greater than  $x.key$

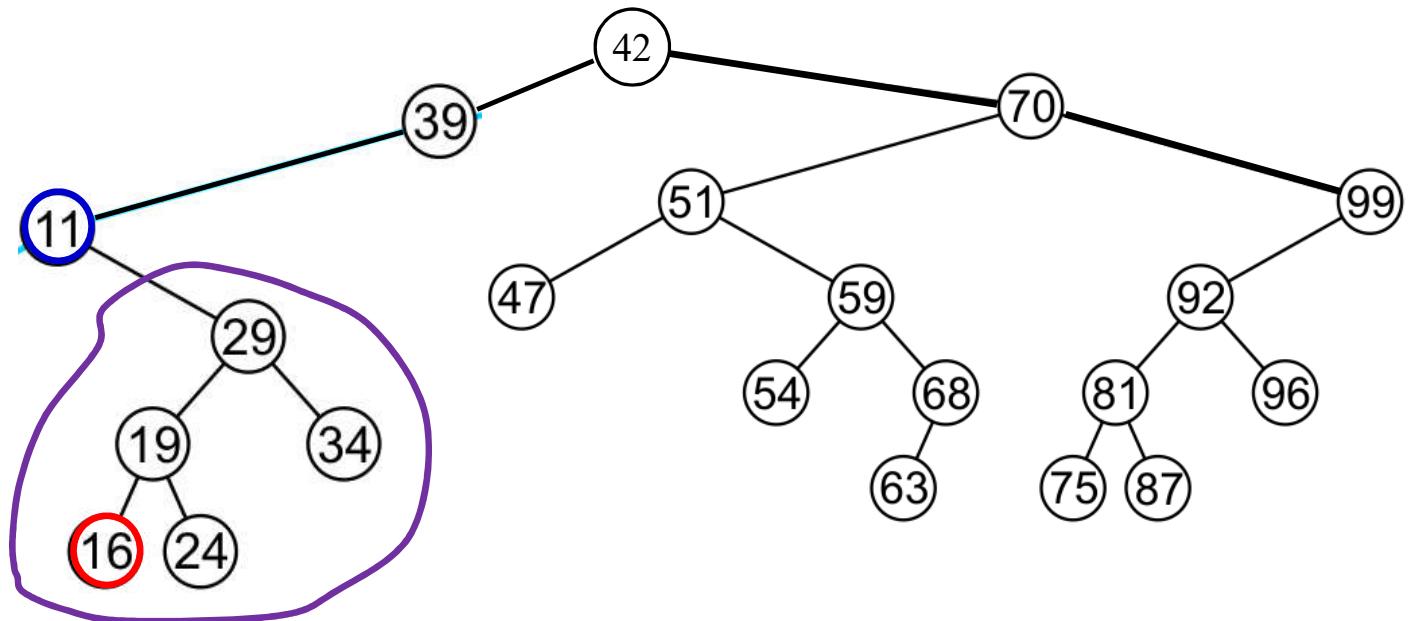
successor of the node with 11 : the node with 16



# BST Operation: Successor

successor of a node  $x$  : the node with the smallest key greater than  $x.key$

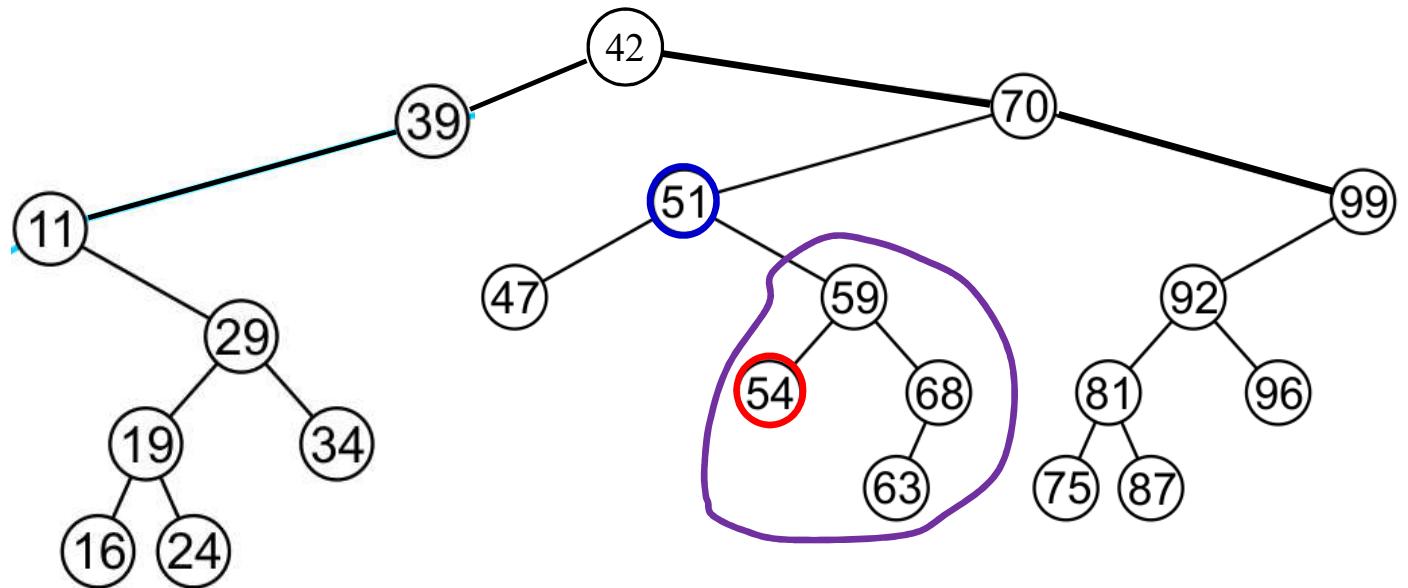
successor of the node with 11 : the node with 16 (minimum of right subtree)



# BST Operation: Successor

successor of a node  $x$  : the node with the smallest key greater than  $x.key$

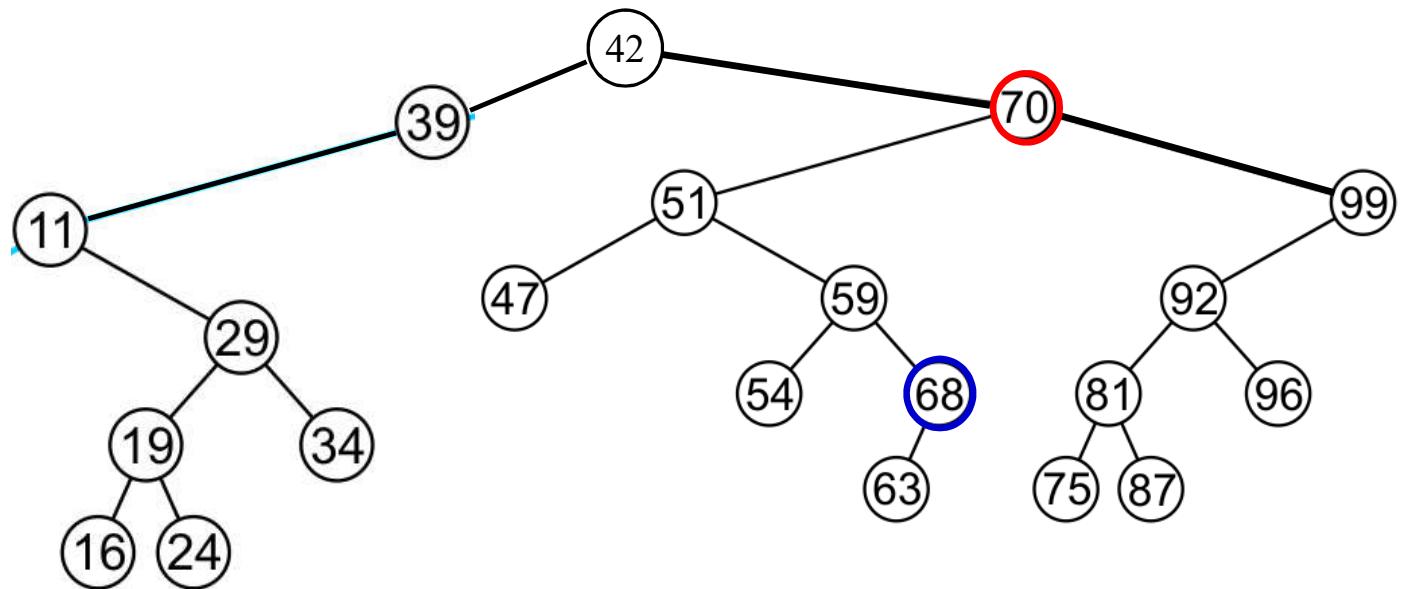
successor of the node with 51 : the node with 54 (minimum of right subtree)



# BST Operation: Successor

successor of a node  $x$  : the node with the smallest key greater than  $x.key$

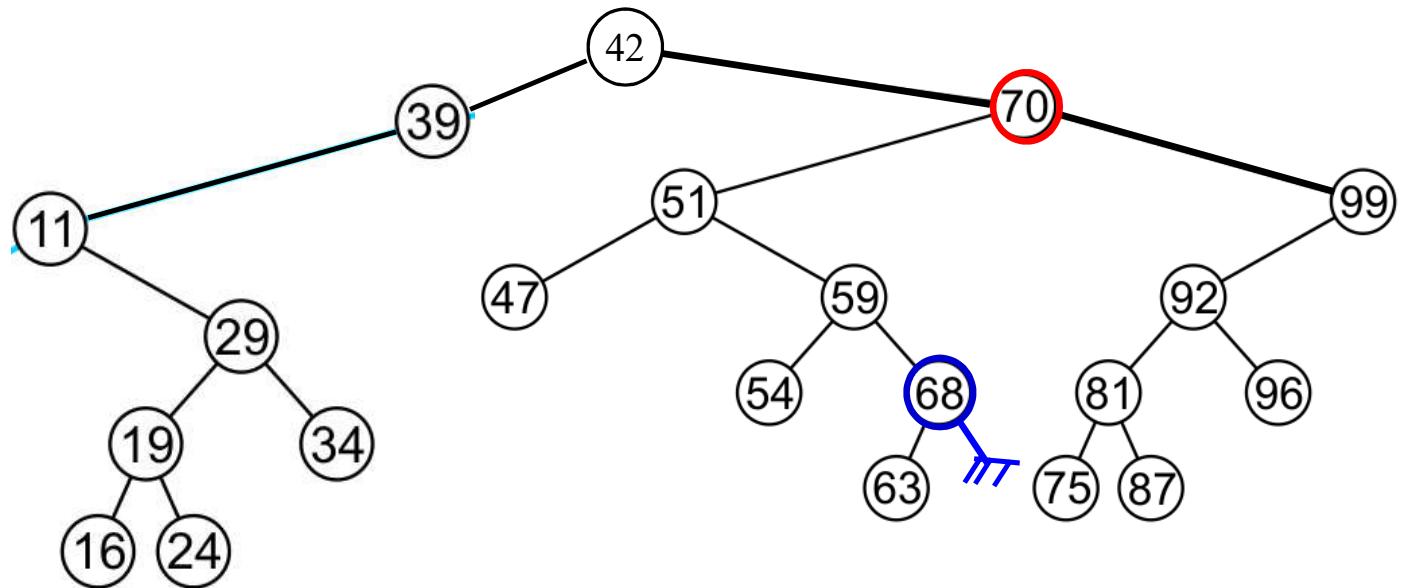
successor of the node with 68 : the node with 70



# BST Operation: Successor

successor of a node  $x$  : the node with the smallest key greater than  $x.key$

successor of the node with 68 : the node with 70 (right subtree is NULL)



# BST Operation: Successor

successor of a node  $x$  : the node with the smallest key greater than  $x.key$

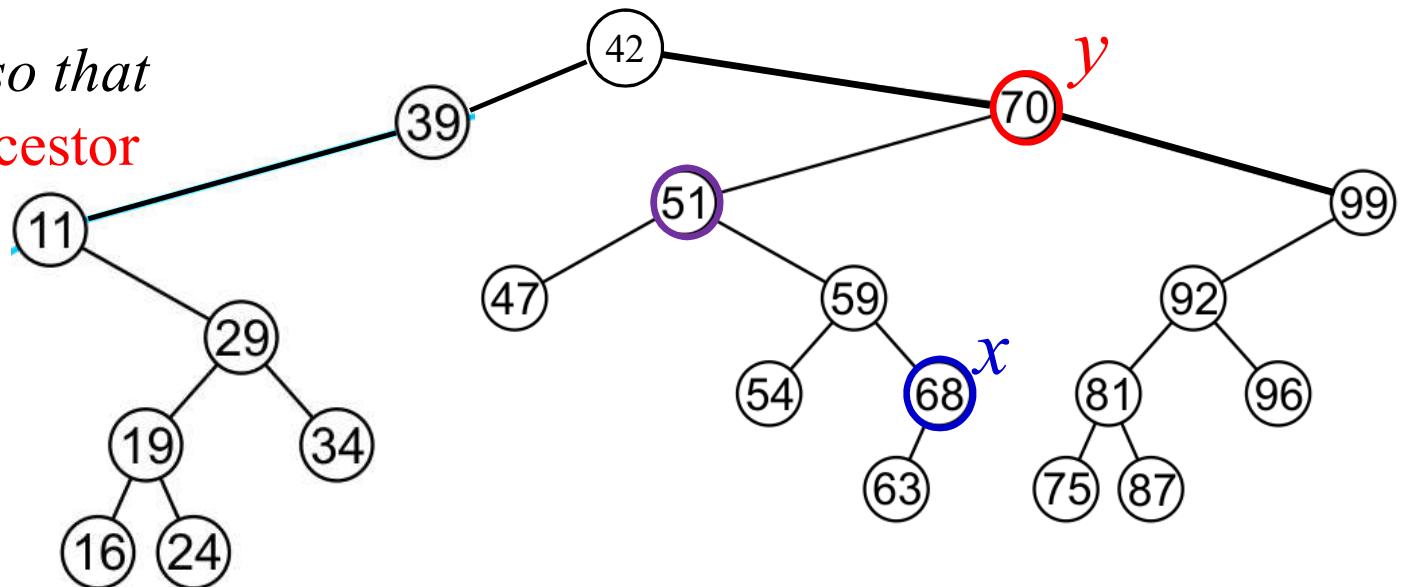
successor of the node with 68 : the node with 70 (right subtree is NULL)

$y$  is successor of  $x$

$y$  is lowest ancestor of  $x$  so that

$y$ 's left child is also an ancestor

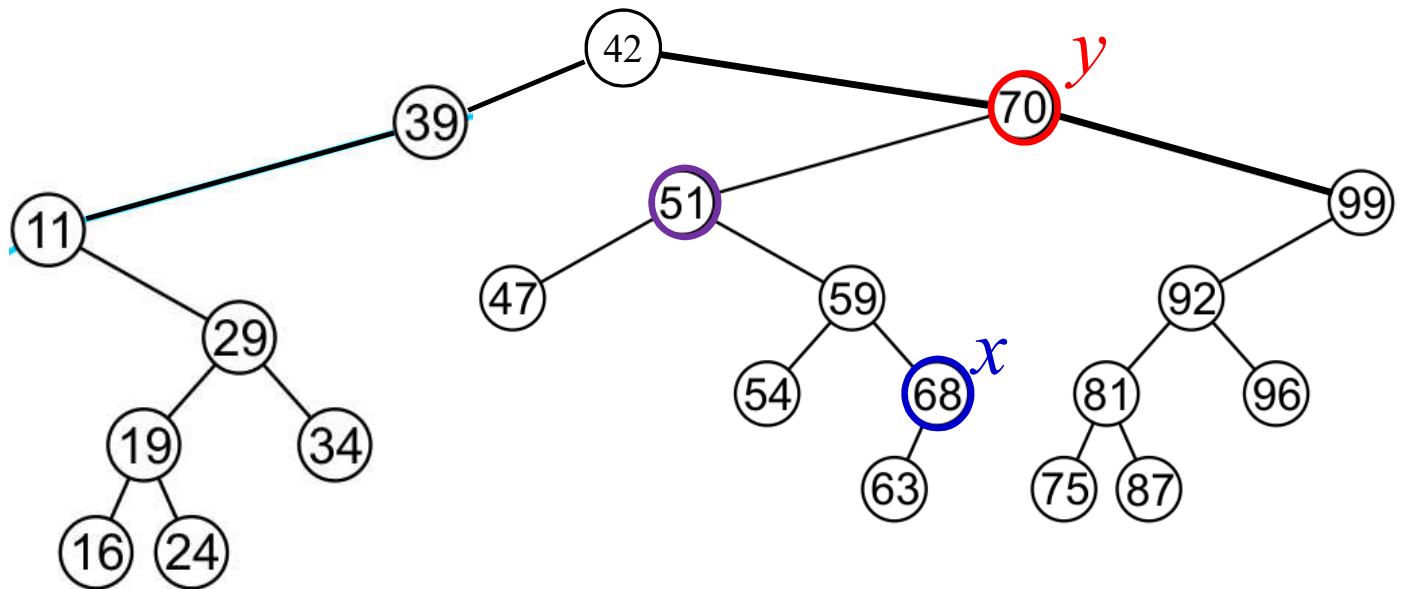
of  $x$



# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

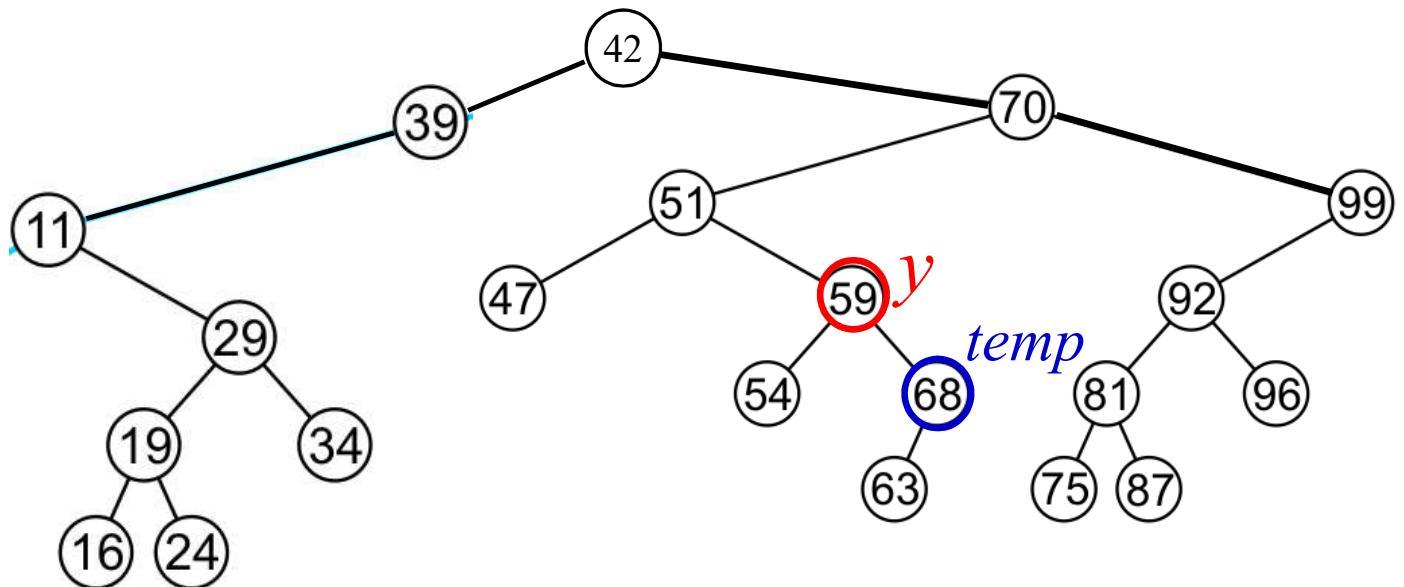
```
1 if x->right ≠ NULL  
2     return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5     temp = y  
6     y = y->parent  
7 return y
```



# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

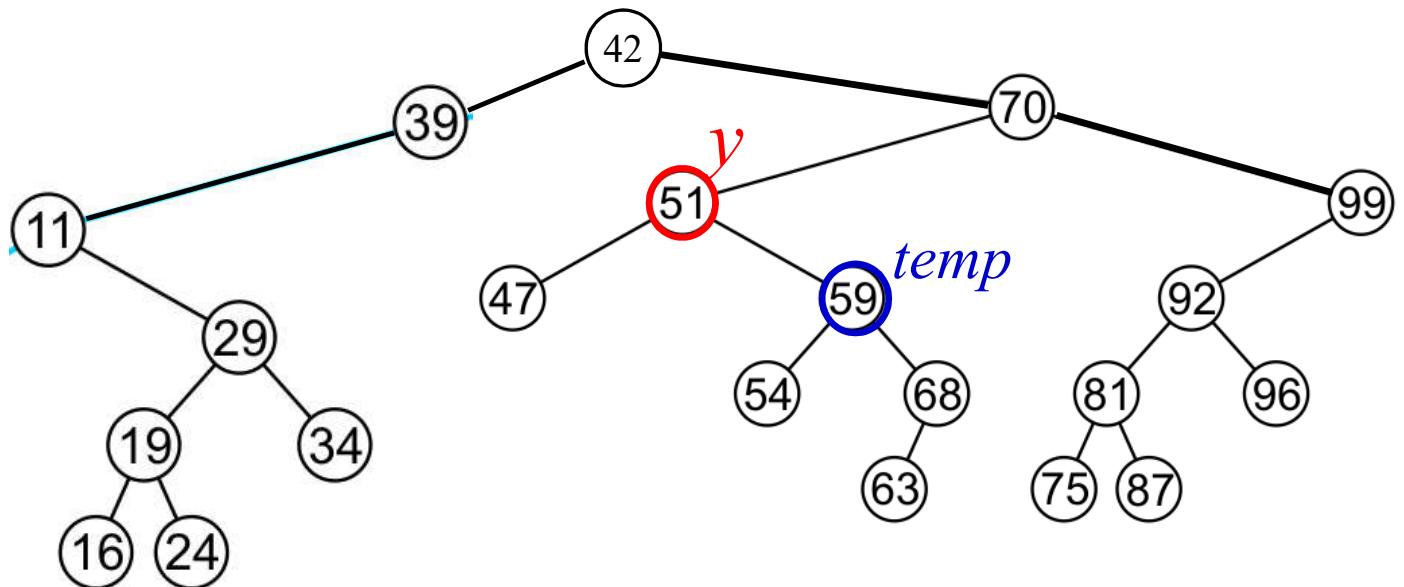
```
1 if x->right ≠ NULL  
2     return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5     temp = y  
6     y = y->parent  
7 return y
```



# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

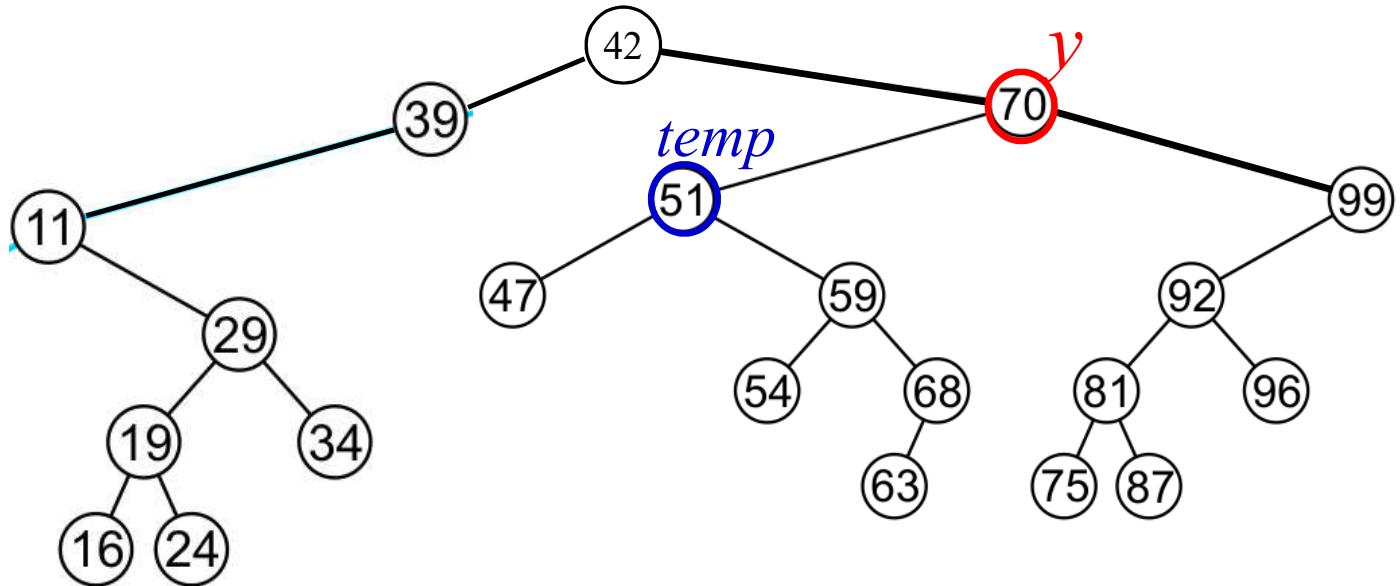
```
1 if x->right ≠ NULL  
2     return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5     temp = y  
6     y = y->parent  
7 return y
```



# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

```
1 if x->right ≠ NULL  
2     return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5     temp = y  
6     y = y->parent  
7 return y
```

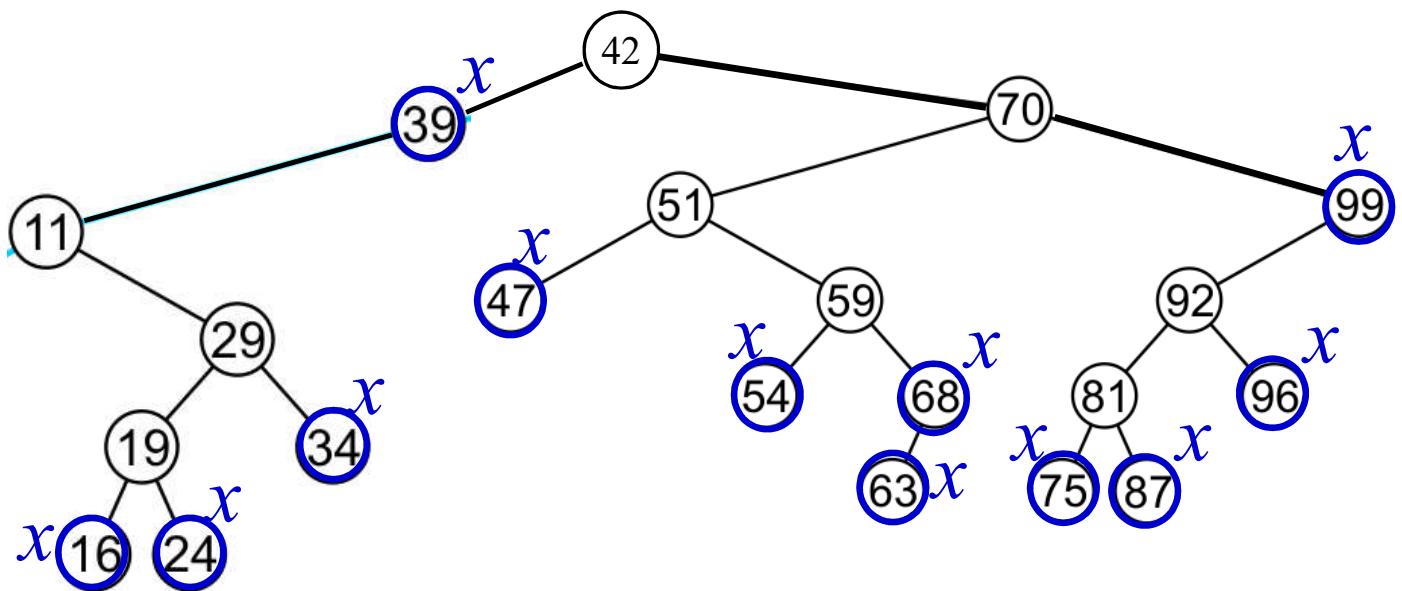


# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

```
1 if x->right ≠ NULL  
2   return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5   temp = y  
6   y = y->parent  
7 return y
```

Successor of  $x$  ?

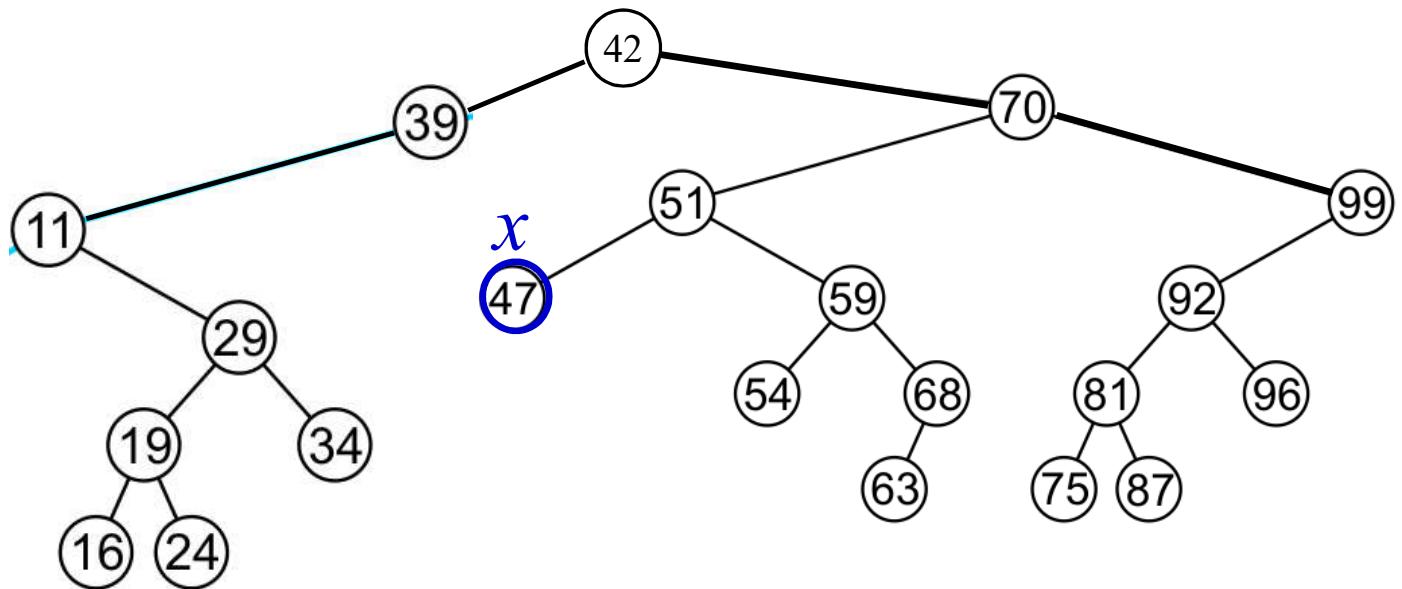


# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

```
1 if x->right ≠ NULL  
2   return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5   temp = y  
6   y = y->parent  
7 return y
```

Successor of  $x$  ?

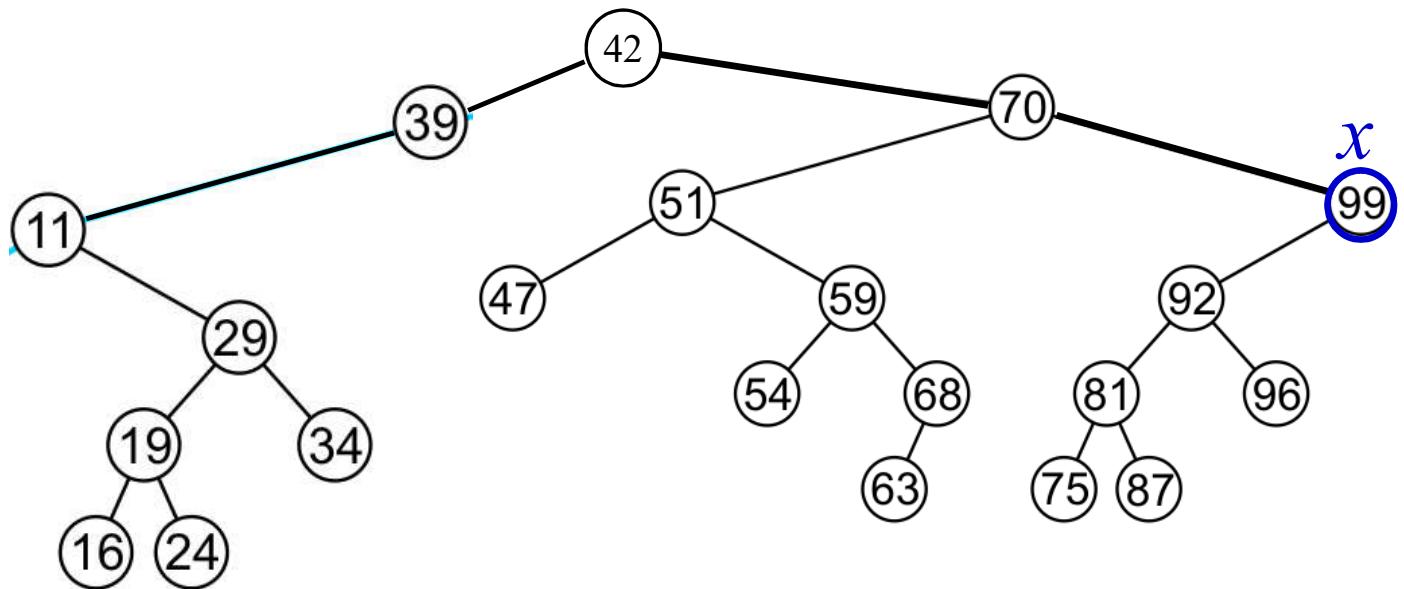


# BST Operation: Successor

```
TREE_SUCCESSOR(x)
```

```
1 if x->right ≠ NULL  
2   return TREE_MINIMUM(x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5   temp = y  
6   y = y->parent  
7 return y
```

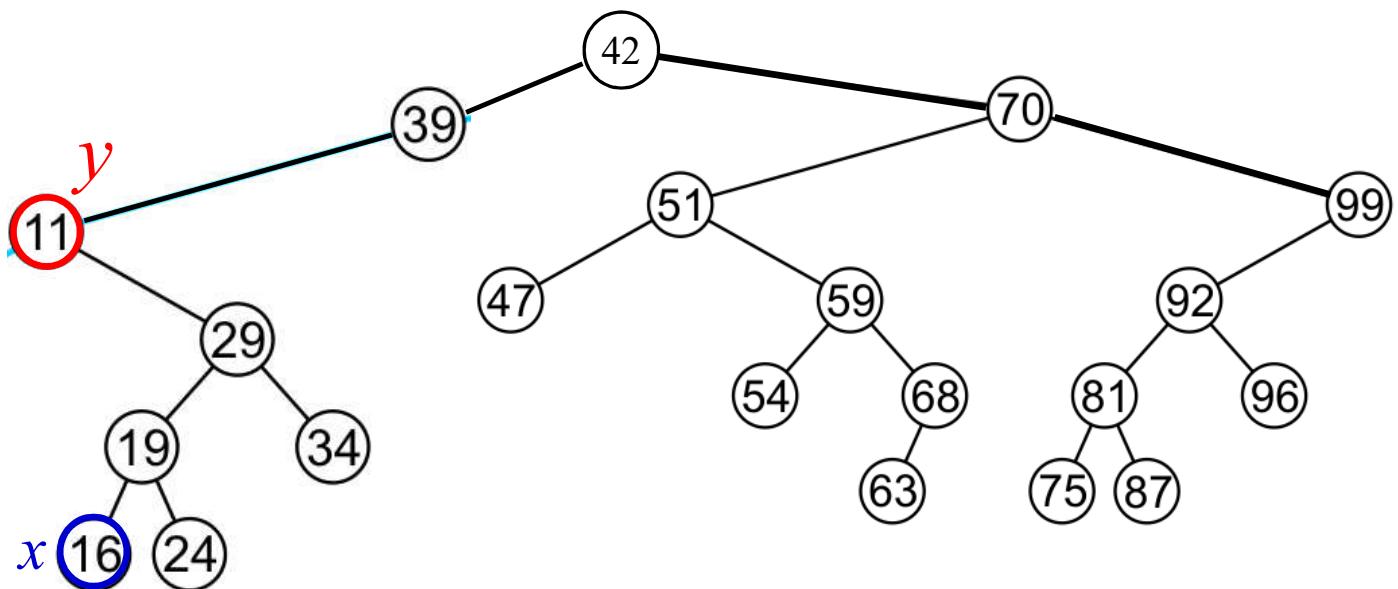
Successor of  $x$  ?



# BST Operation: Predecessor

```
TREE_PREDECESSOR (x)
```

```
1 if x->left ≠ NULL  
2     return TREE_MAXIMUM (x->left)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->left  
5     temp = y  
6     y = y->parent  
7 return y
```

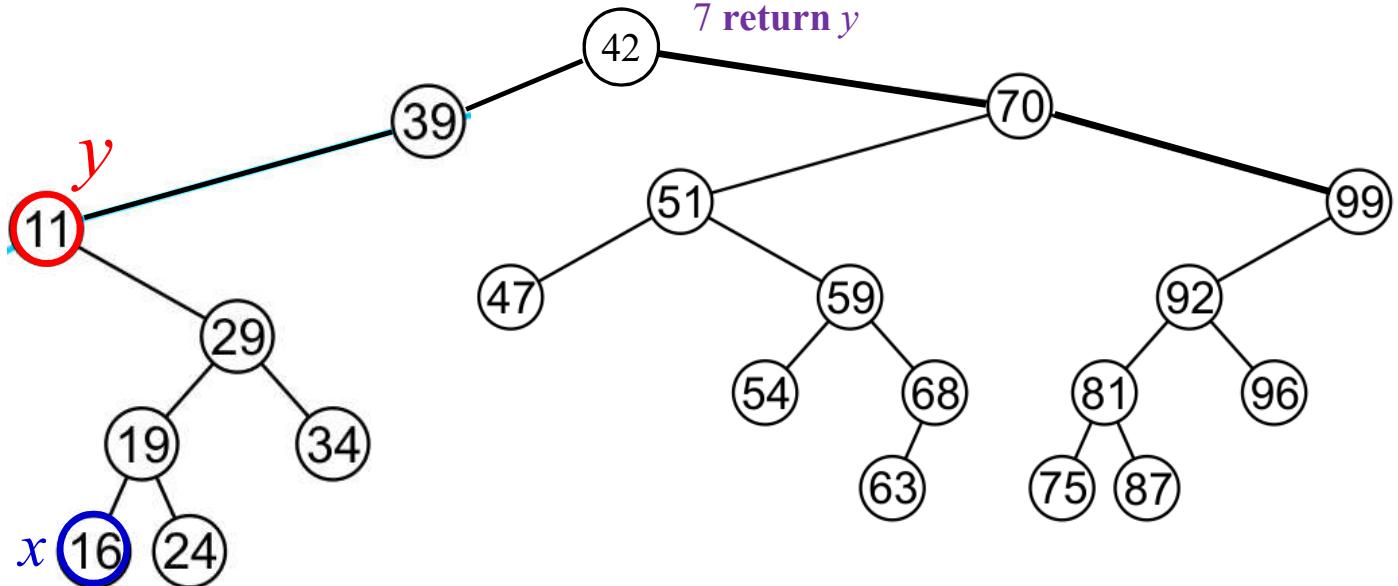


# BST Operation: Predecessor

```
TREE_PREDECESSOR (x)
```

```
1 if x->left ≠ NULL  
2     return TREE_MAXIMUM (x->left)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->left  
5     temp = y  
6     y = y->parent  
7 return y
```

```
TREE_SUCCESSOR (x)  
1 if x->right ≠ NULL  
2     return TREE_MINIMUM (x->right)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->right  
5     temp = y  
6     y = y->parent  
7 return y
```

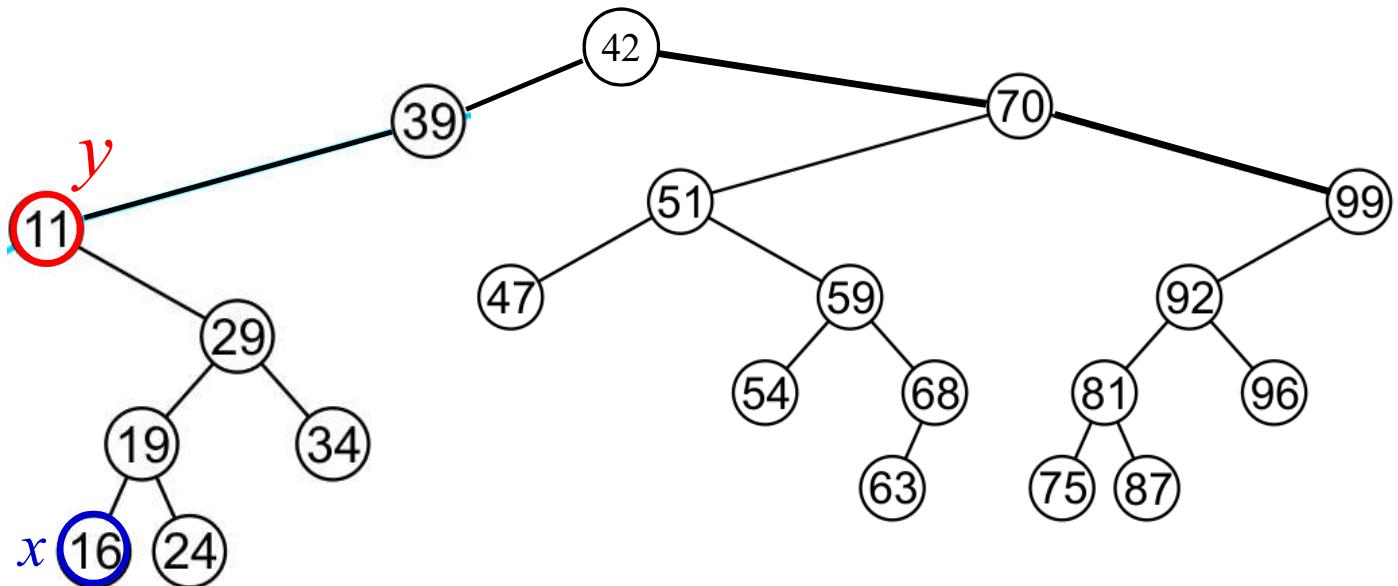


# BST Operation: Predecessor

```
TREE_PREDECESSOR (x)
```

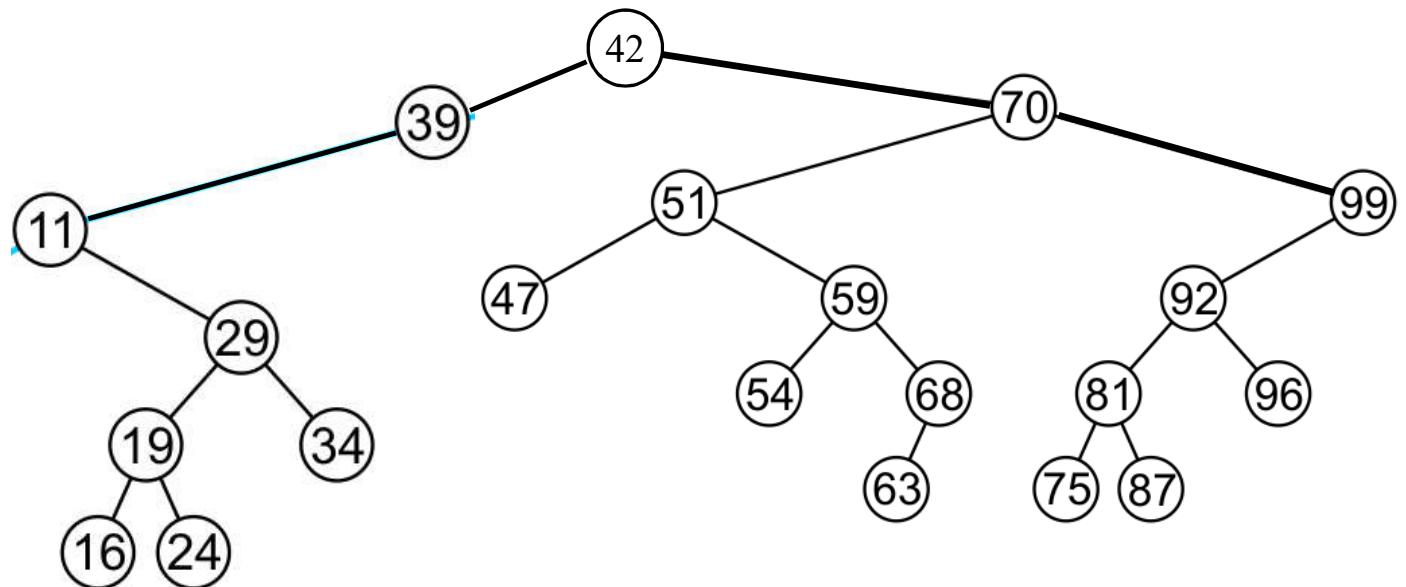
```
1 if x->left ≠ NULL  
2     return TREE_MAXIMUM (x->left)  
3 temp = x; y = temp->parent  
4 while y ≠ NULL and temp == y->left  
5     temp = y  
6     y = y->parent  
7 return y
```

Complexity  $O(h)$



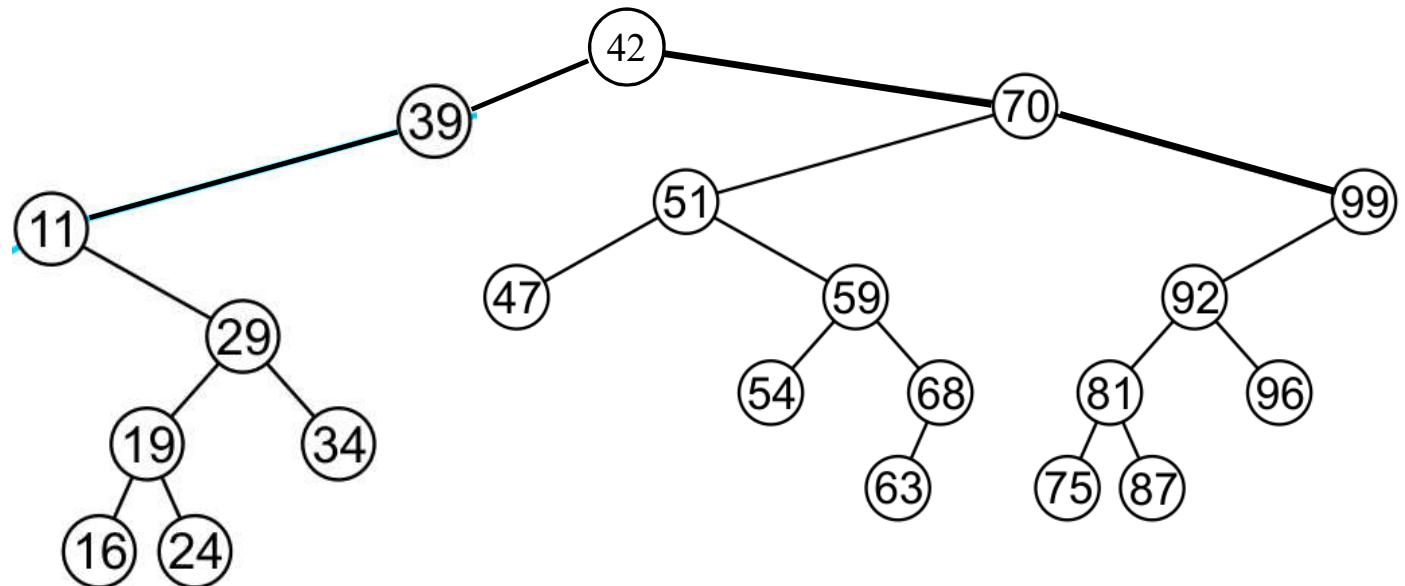
# BST Operation: Insertion

An insertion will be performed at a leaf node



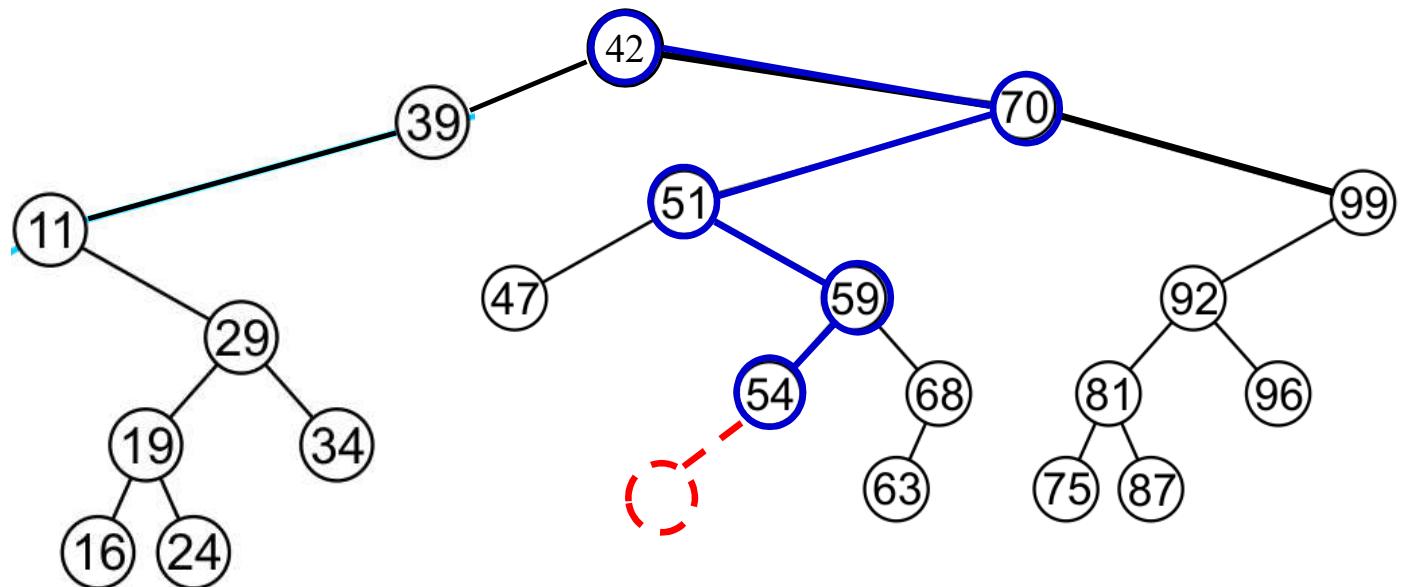
# BST Operation: Insertion

Given a *key* to insert, find the location if it were in the tree



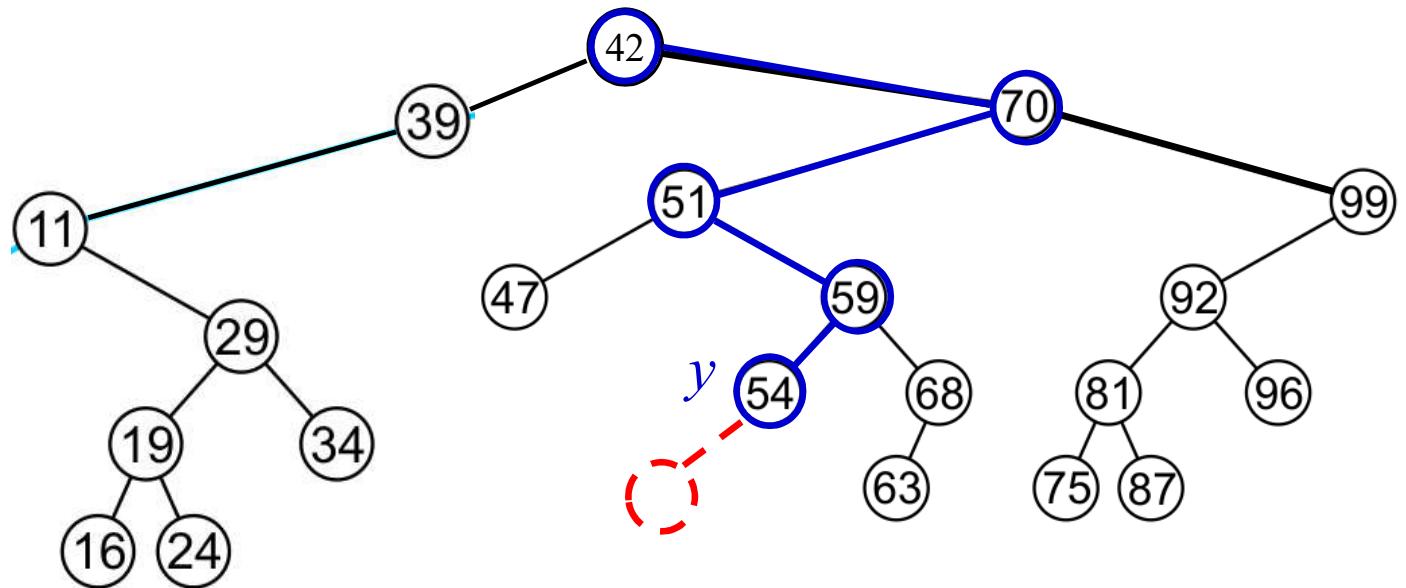
# BST Operation: Insertion

To insert a node  $z$  with key 52



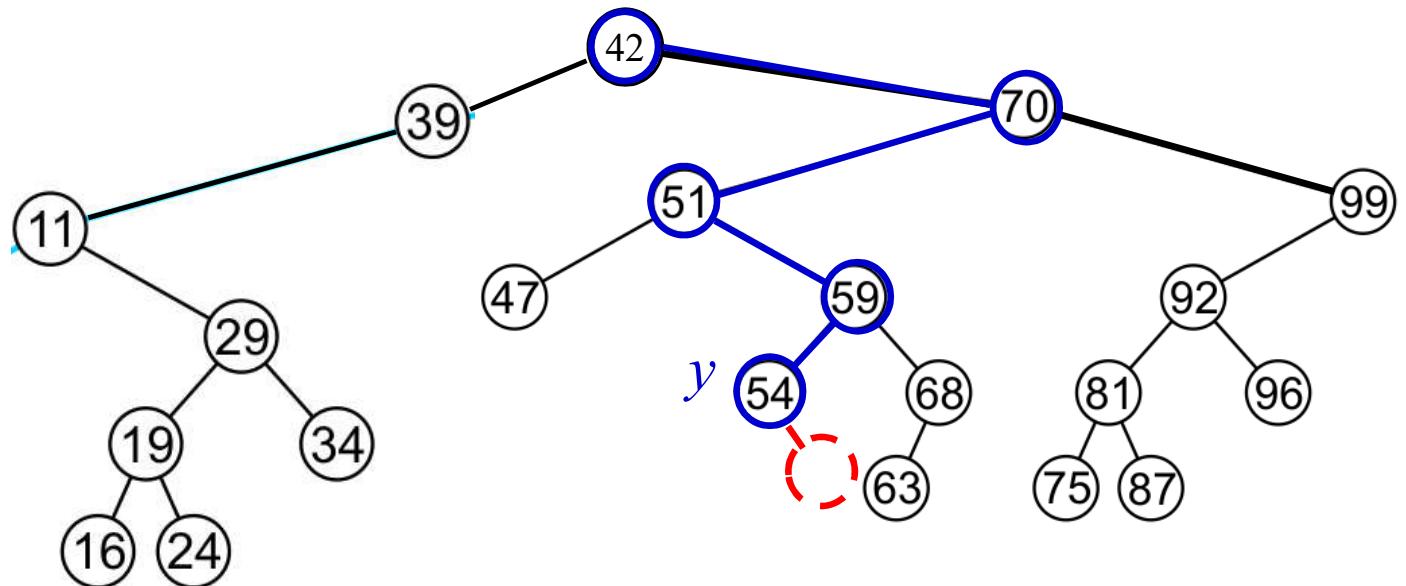
# BST Operation: Insertion

To insert a node  $z$  with key 52



# BST Operation: Insertion

To insert a node  $z$  with key 55

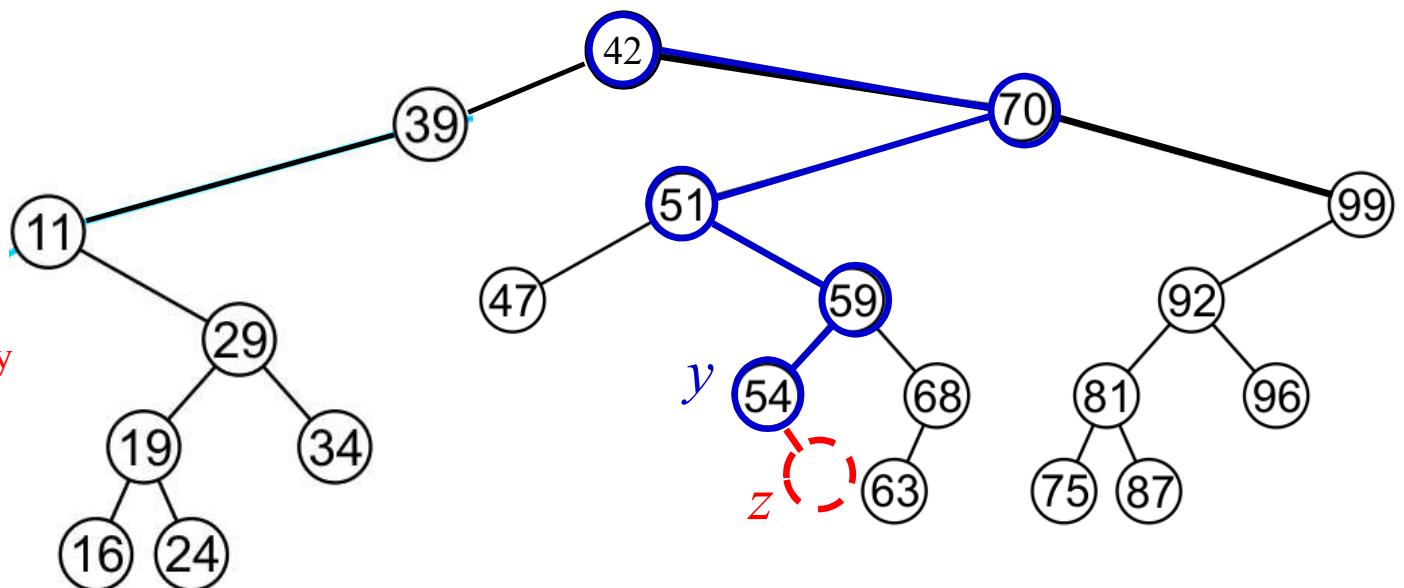


# BST Operation: Insertion

TREE\_INSERT ( $T, z$ )

```
1  $y = \text{NULL}$ 
2  $x = T \rightarrow \text{root}$ 
3 while  $x \neq \text{NULL}$ 
4    $y = x$ 
5   if  $z \rightarrow \text{key} < x \rightarrow \text{key}$ 
6      $x = x \rightarrow \text{left}$ 
7   else  $x = x \rightarrow \text{right}$ 
```

```
8  $z \rightarrow \text{parent} = y$ 
9 if  $y == \text{NULL}$ 
10    $T \rightarrow \text{root} = z$  // tree T was empty
11 elseif  $z \rightarrow \text{key} < y \rightarrow \text{key}$ 
12    $y \rightarrow \text{left} = z$ 
13 else  $y \rightarrow \text{right} = z$ 
```



# BST Operation: Insertion

TREE\_INSERT ( $T, z$ )

```
1  $y = \text{NULL}$ 
2  $x = T \rightarrow \text{root}$ 
3 while  $x \neq \text{NULL}$ 
4    $y = x$ 
5   if  $z \rightarrow \text{key} < x \rightarrow \text{key}$ 
6      $x = x \rightarrow \text{left}$ 
7   else  $x = x \rightarrow \text{right}$ 
8  $z \rightarrow \text{parent} = y$ 
9 if  $y == \text{NULL}$ 
10    $T \rightarrow \text{root} = z // \text{tree } T \text{ was empty}$ 
11 elseif  $z \rightarrow \text{key} < y \rightarrow \text{key}$ 
12    $y \rightarrow \text{left} = z$ 
13 else  $y \rightarrow \text{right} = z$ 
```

Complexity:  $O(h)$

