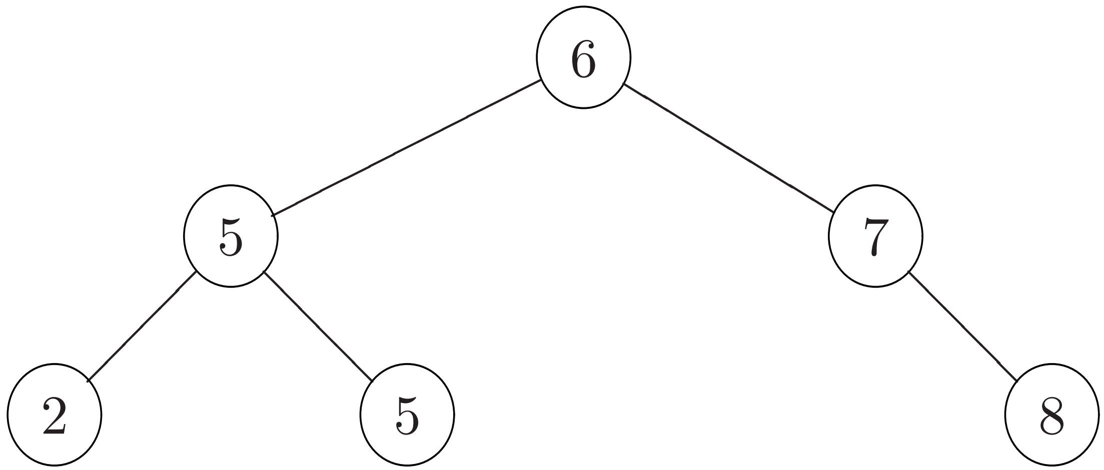
Binary Search Trees

Binary Search Trees

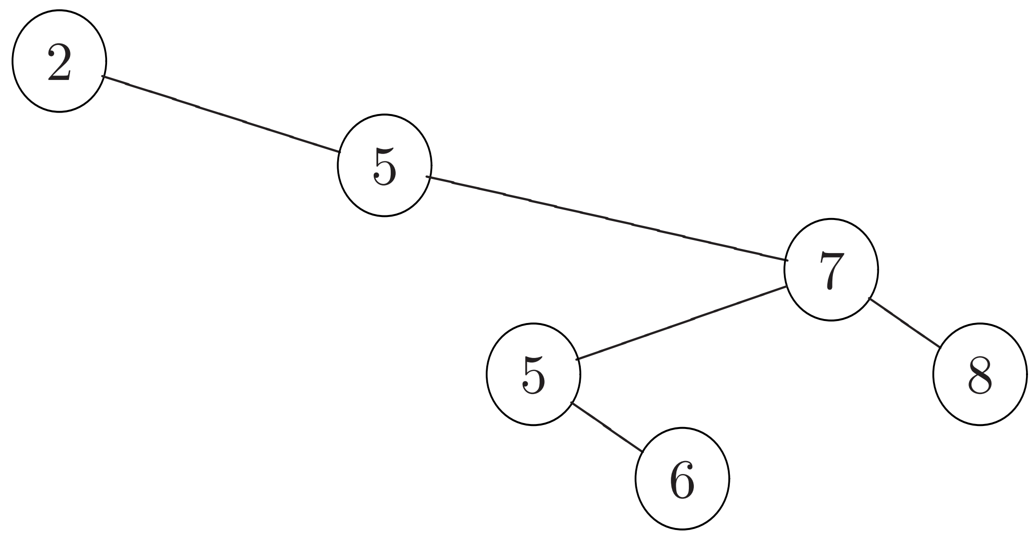


Binary search tree on $(2, 5, 5, 6, 7, 8)$.

Binary Search Tree Property. Let node y be a descendant of node x.

- If y is in the left subtree of x, then $y.\mathsf{key} \leq x.\mathsf{key}$;
- If y is in the right subtree of x, then $y.\mathsf{key} \geq x.\mathsf{key}$.

Binary Search Trees

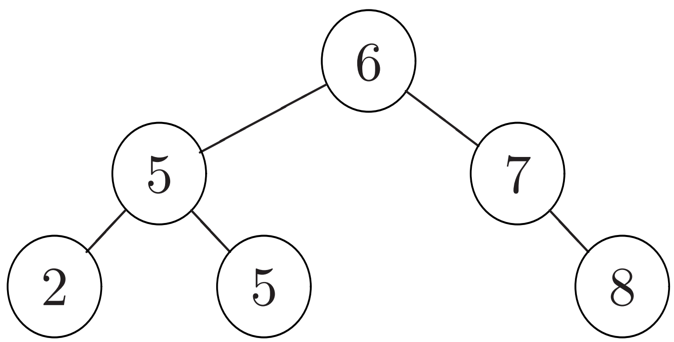


Another binary search tree on $(2, 5, 5, 6, 7, 8)$.

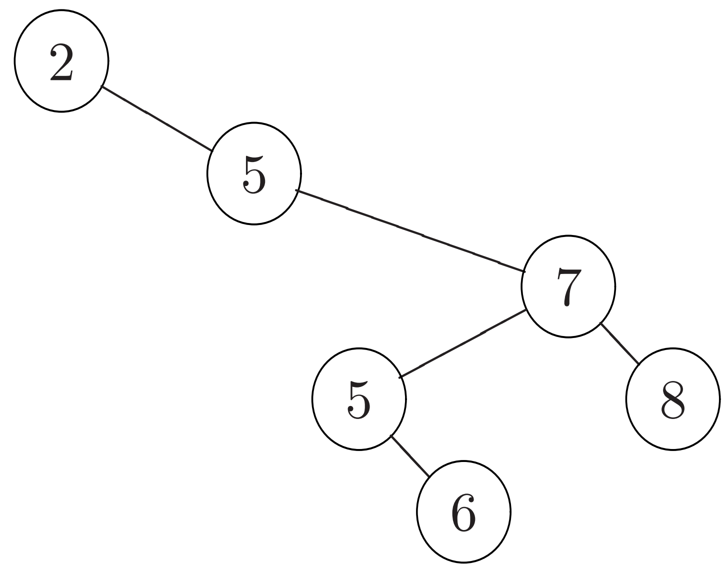
Binary Search Tree Property. Let node y be a descendant of node x.

- If y is in the left subtree of x, then $y.\mathsf{key} \leq x.\mathsf{key}$;
- If y is in the right subtree of x, then $y.\mathsf{key} \geq x.\mathsf{key}$.

Binary Search Trees



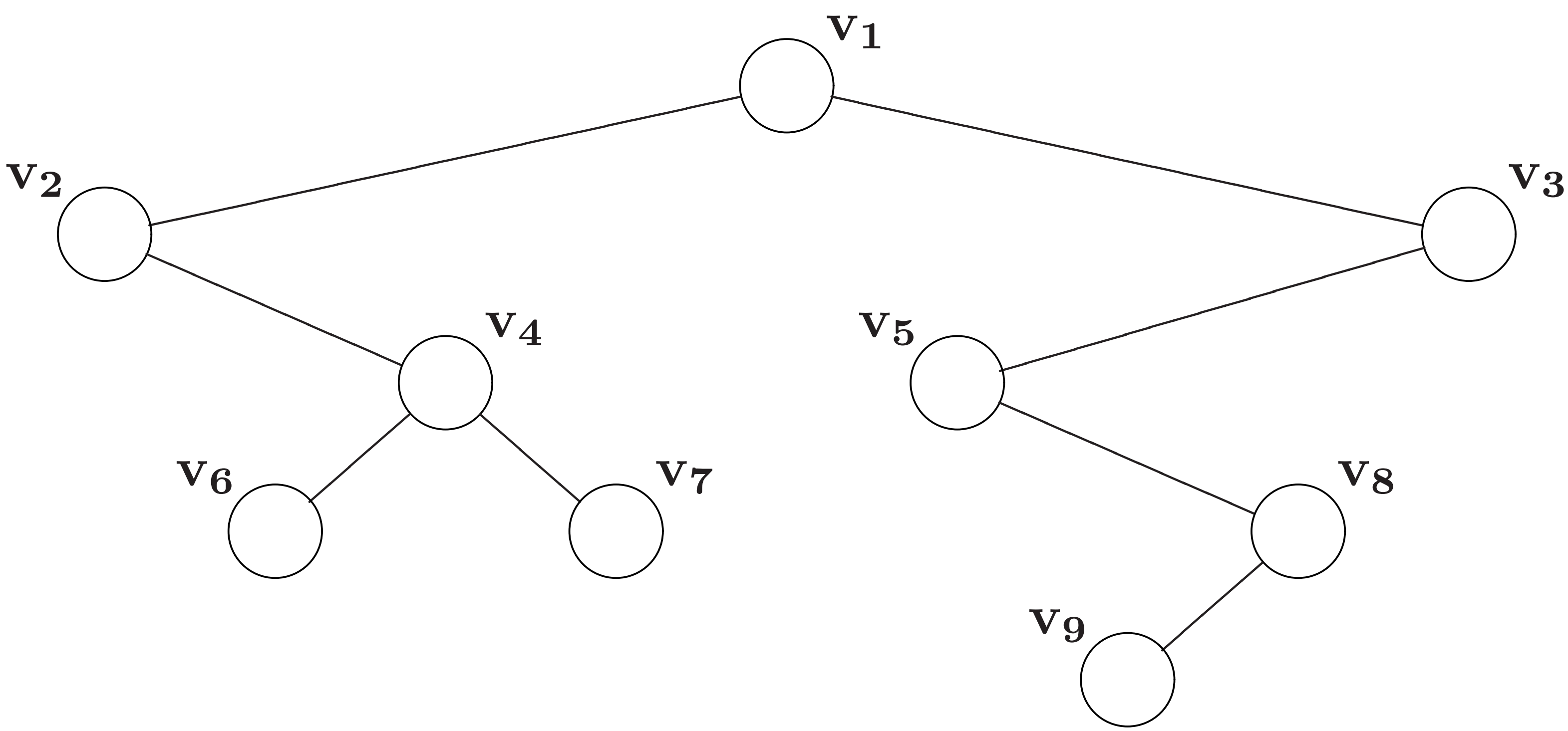
A



B

Binary search tree on $(2, 5, 5, 6, 7, 8)$.

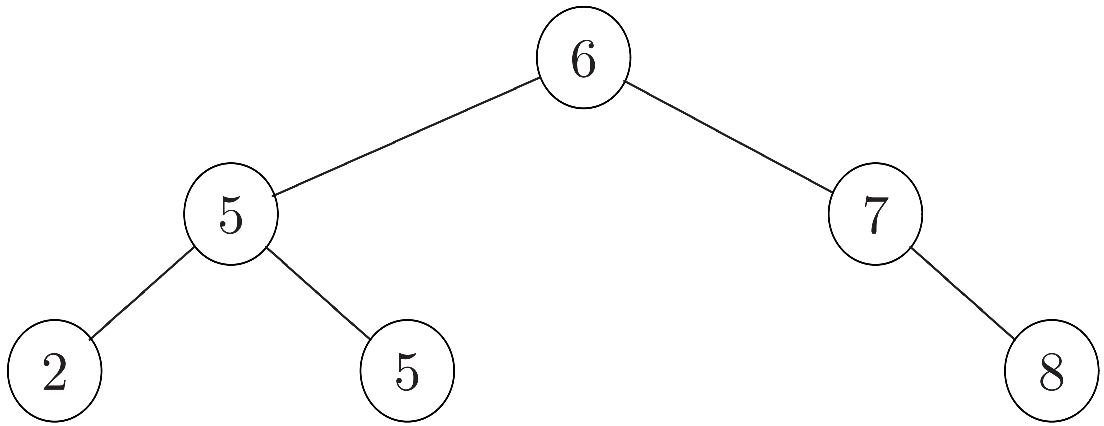
Binary Search Tree: Exercise



Assign the following values to the tree nodes so that the tree is a binary search tree:

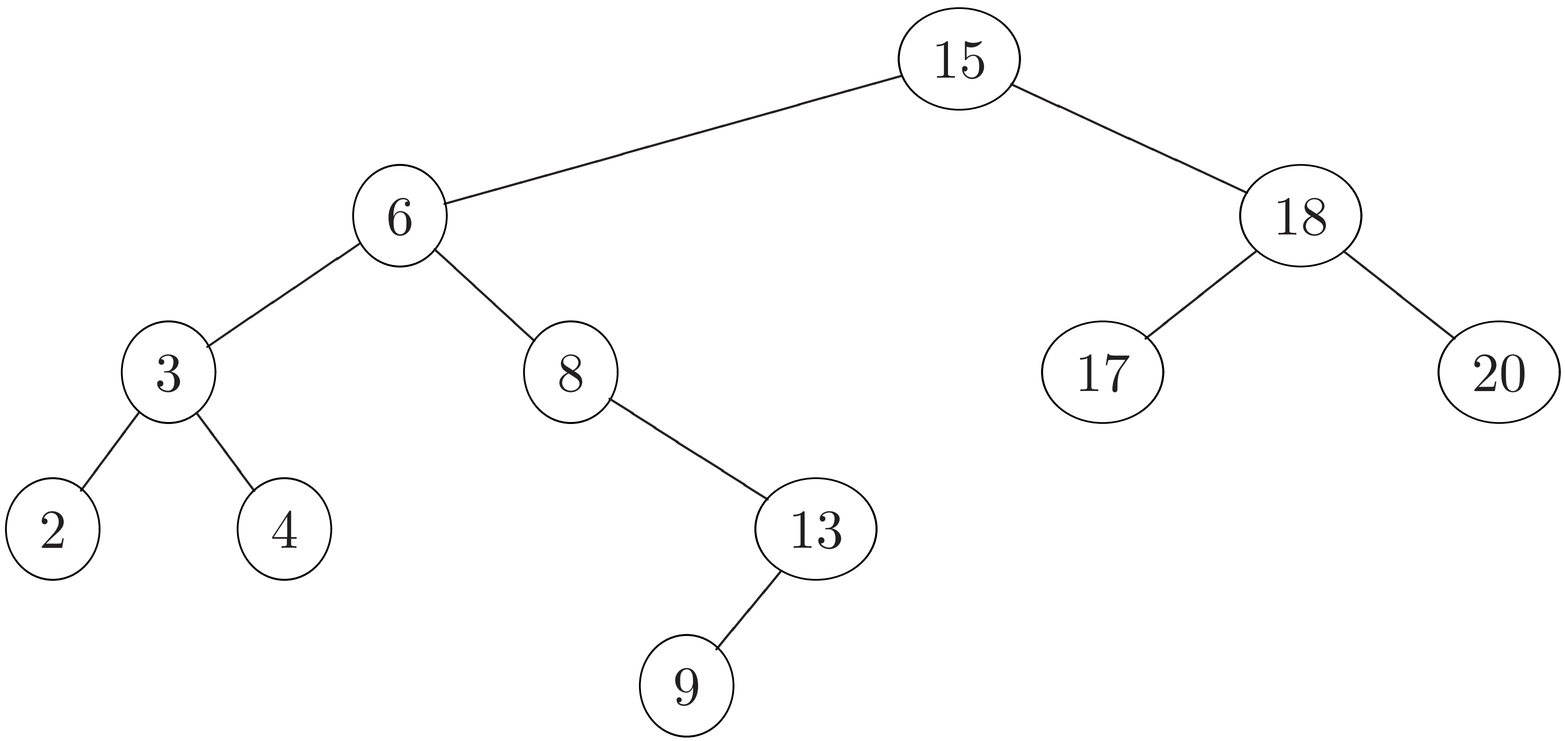
$$7, 12, 14, 15, 18, 22, 25, 27, 30$$

Inorder Tree Walk




```
procedure InorderTreeWalk(x)
if (x ≠ NIL) then
    InorderTreeWalk(x.left);
    print x.key;
    InorderTreeWalk(x.right);
end
```

Tree Search



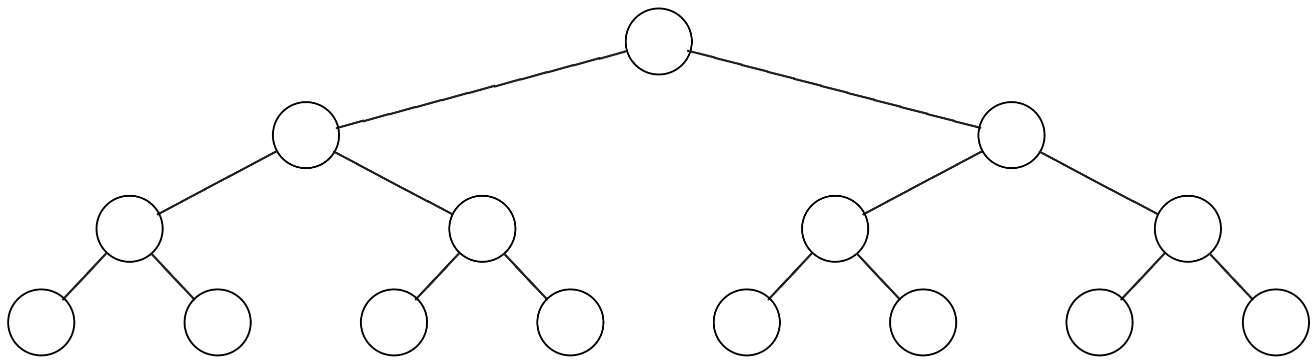
Tree Search

```
procedure TreeSearch(x,K)
if (x = NIL) or (K = x.key) then
    return (x);
else if (K < x.key) then
    TreeSearch(x.left, K);
else
    TreeSearch(x.right, K);
end
```

Iterative Tree Search

```
procedure IterativeTreeSearch(x,K)
while (x ≠ NIL) and (K ≠ x.key) do
    if (K ≤ x.key) then
        x ← x.left;
    else
        x ← x.right;
    end
end
return (x);
```

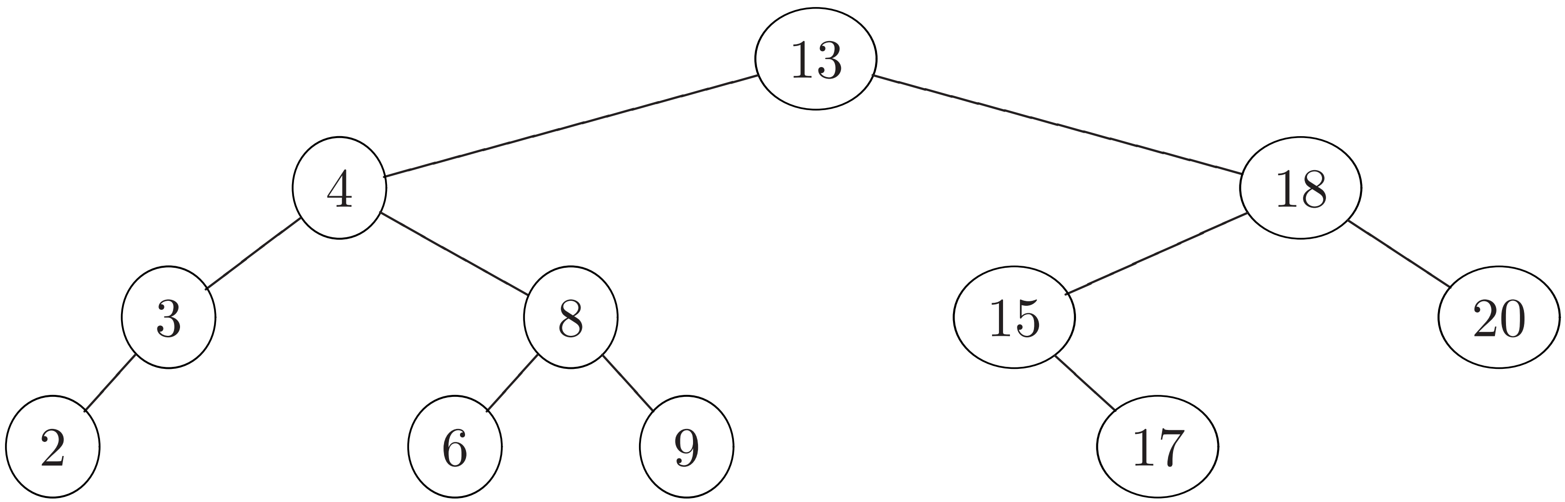
Complete Binary Tree



Definition. A **complete binary tree** is a binary tree where:

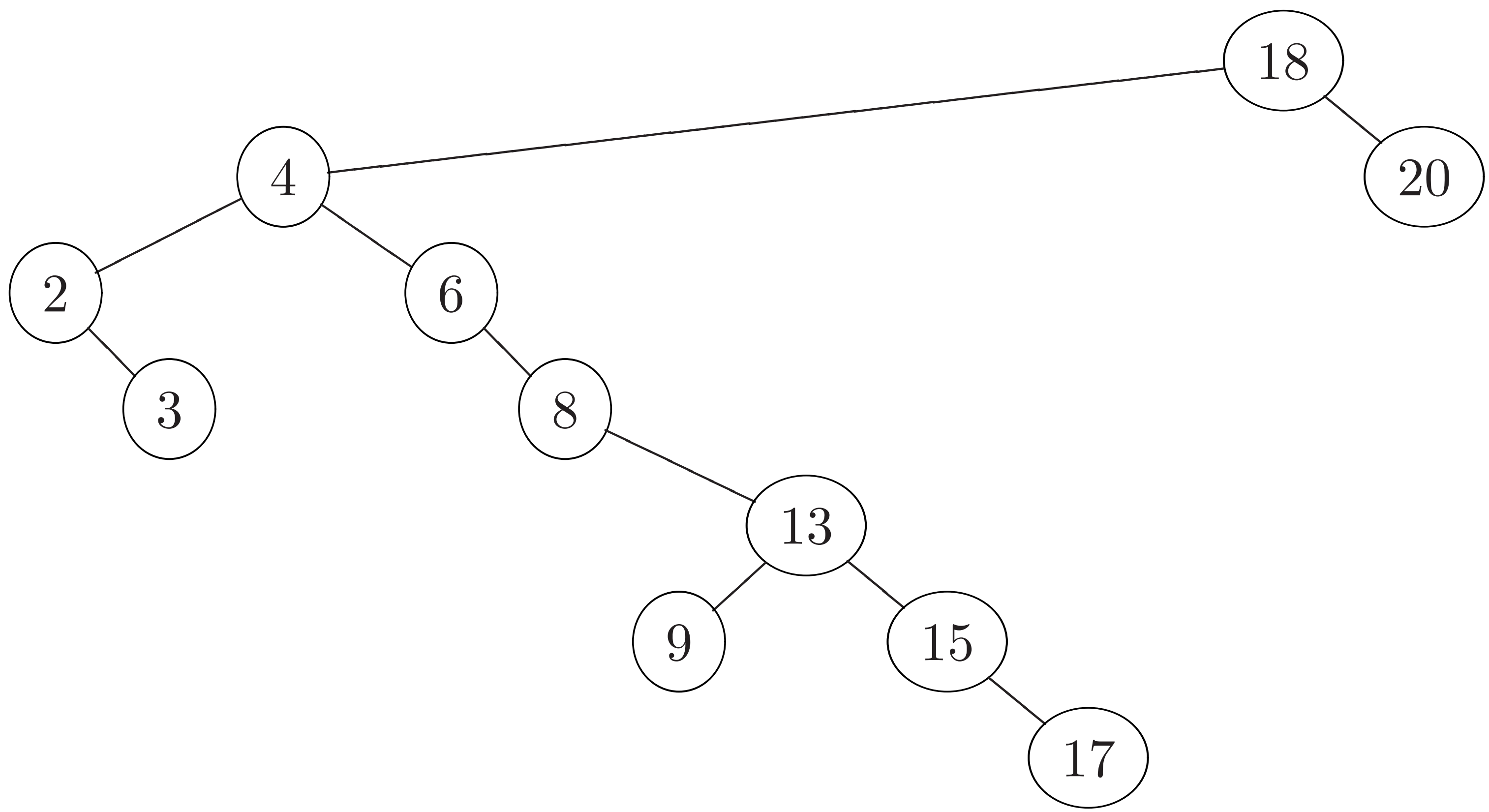
- All the internal nodes have EXACTLY two children;
- All the leaves are at the same distance from the root.

"Balanced" Binary Search Tree



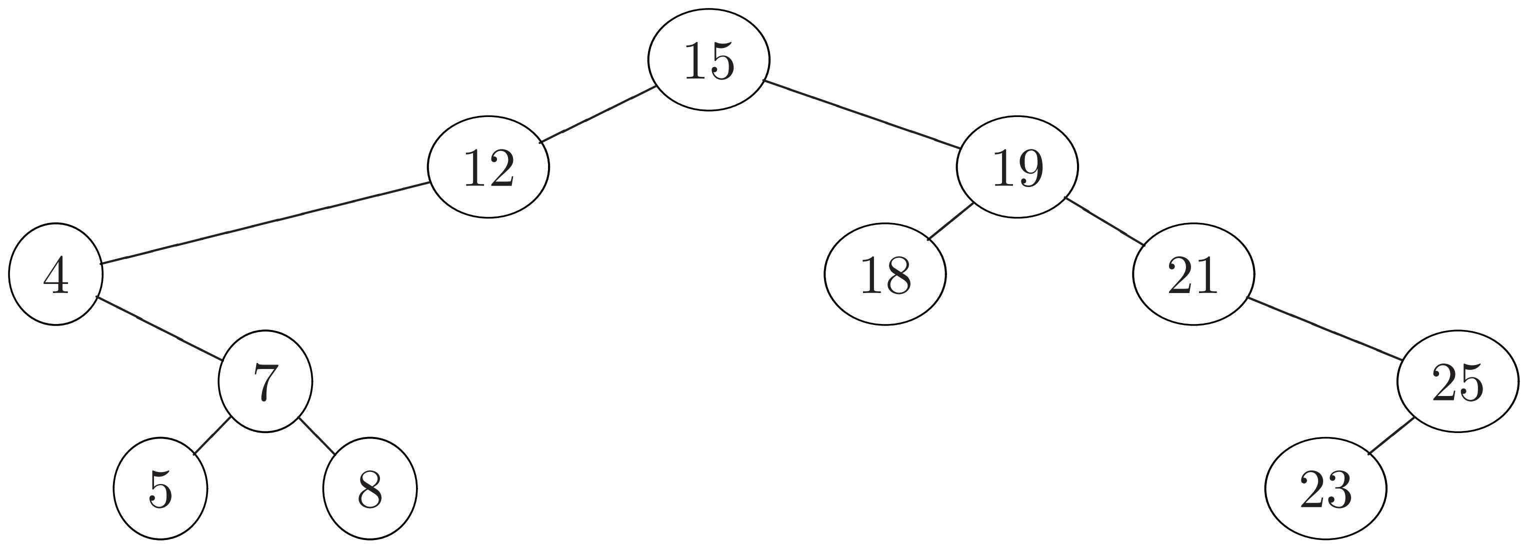
Binary search tree on: (2, 3, 4, 6, 8, 9, 13, 15, 17, 18, 20)

“Unbalanced” Binary Search Tree



Binary search tree on: (2, 3, 4, 6, 8, 9, 13, 15, 17, 18, 20)

Tree Minimum



procedure `TreeMin`(x)

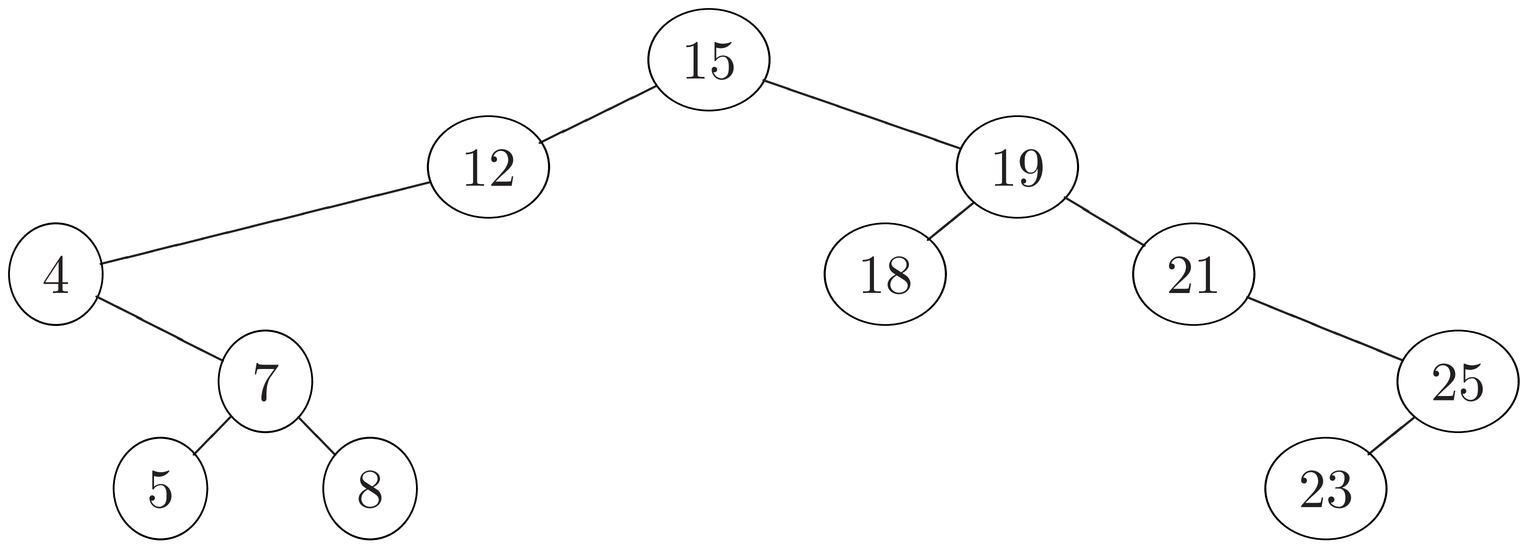
while (x.left $\neq$ **NIL**) **do**

  $x \leftarrow x$.left;

end

return (x);

Tree Maximum



procedure `TreeMax`(x)

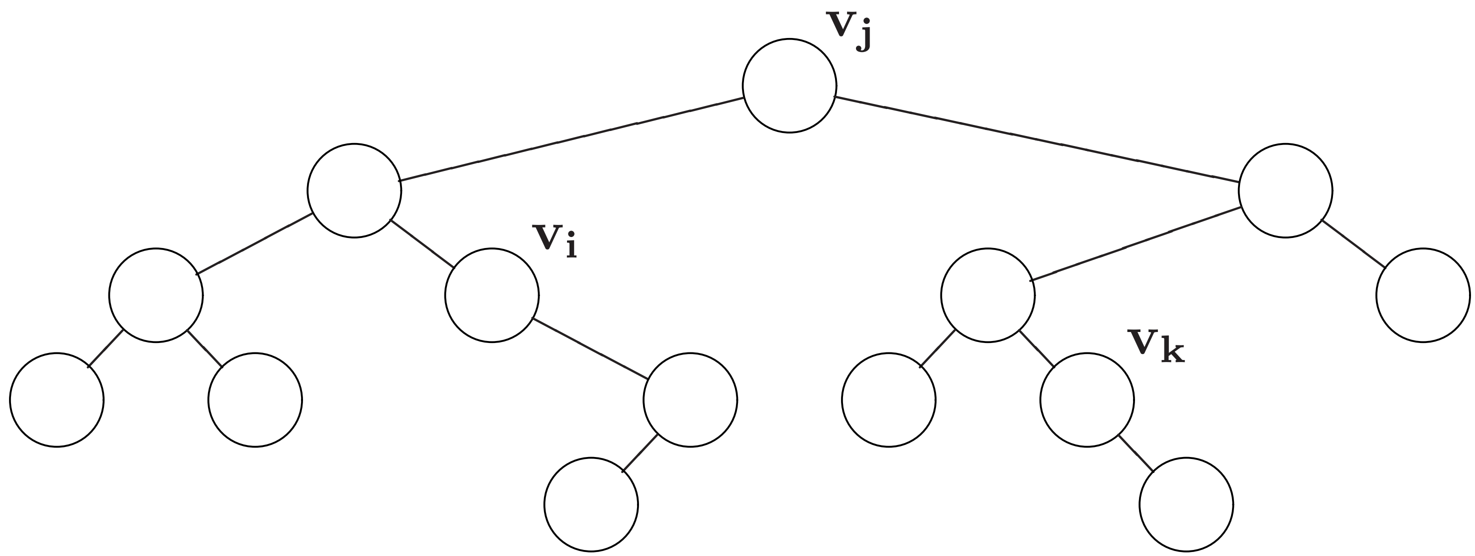
while (x.right $\neq$ **NIL**) **do**

  $x \leftarrow x$.right;

end

return (x);

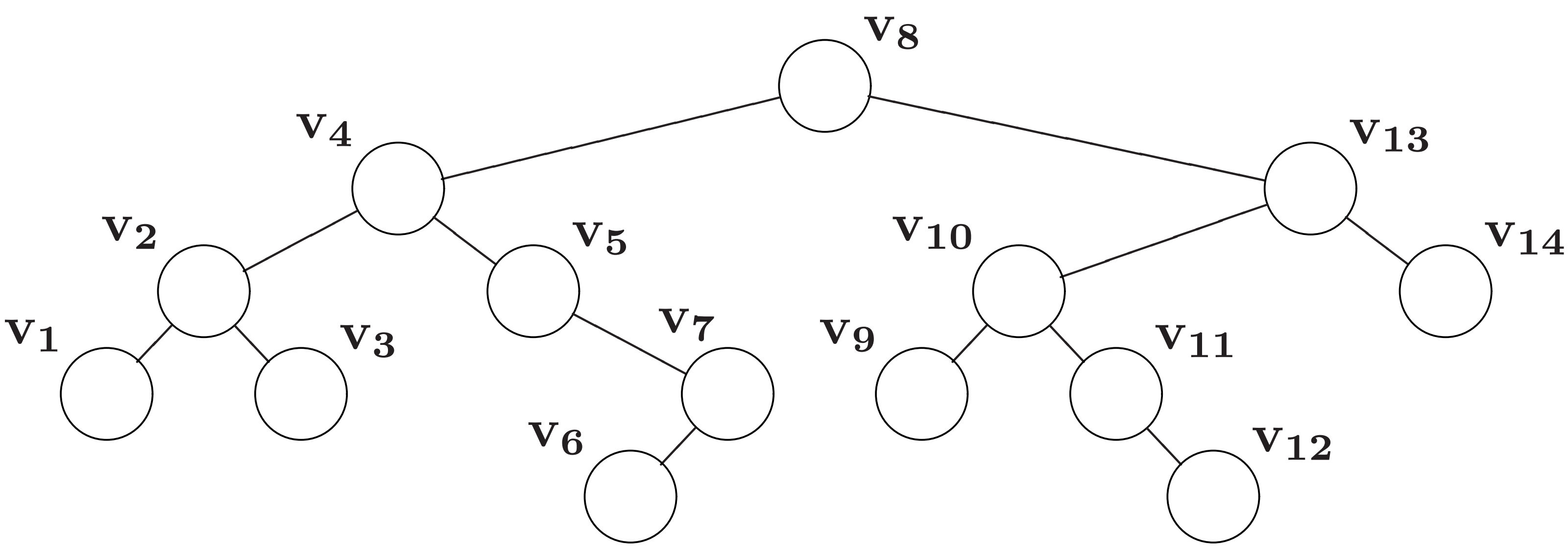
Inorder



- If node v_i is in the subtree rooted at v_j.`Left`, then $v_i \prec v_j$.
- If node v_k is in the subtree rooted at v_j.`Right`, then $v_j \prec v_k$.

Note: If node v_i is in the subtree rooted at v_j.`Left` and node v_k is in the subtree rooted at v_j.`Right`, then $v_i \prec v_j \prec v_k$.

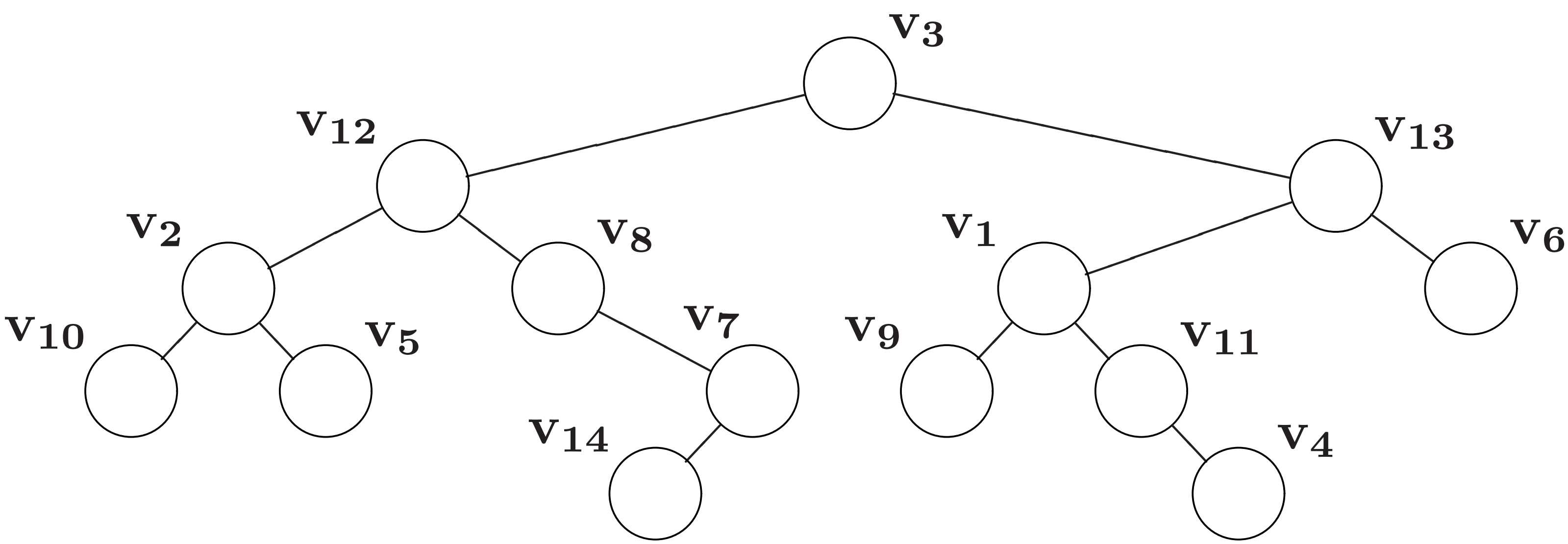
Inorder



- If node v_i is in the subtree rooted at v_j.`Left`, then $v_i \prec v_j$.
- If node v_k is in the subtree rooted at v_j.`Right`, then $v_j \prec v_k$.

Inorder sequence of vertices: $v_1, v_2, v_3, \ldots, v_{14}$.

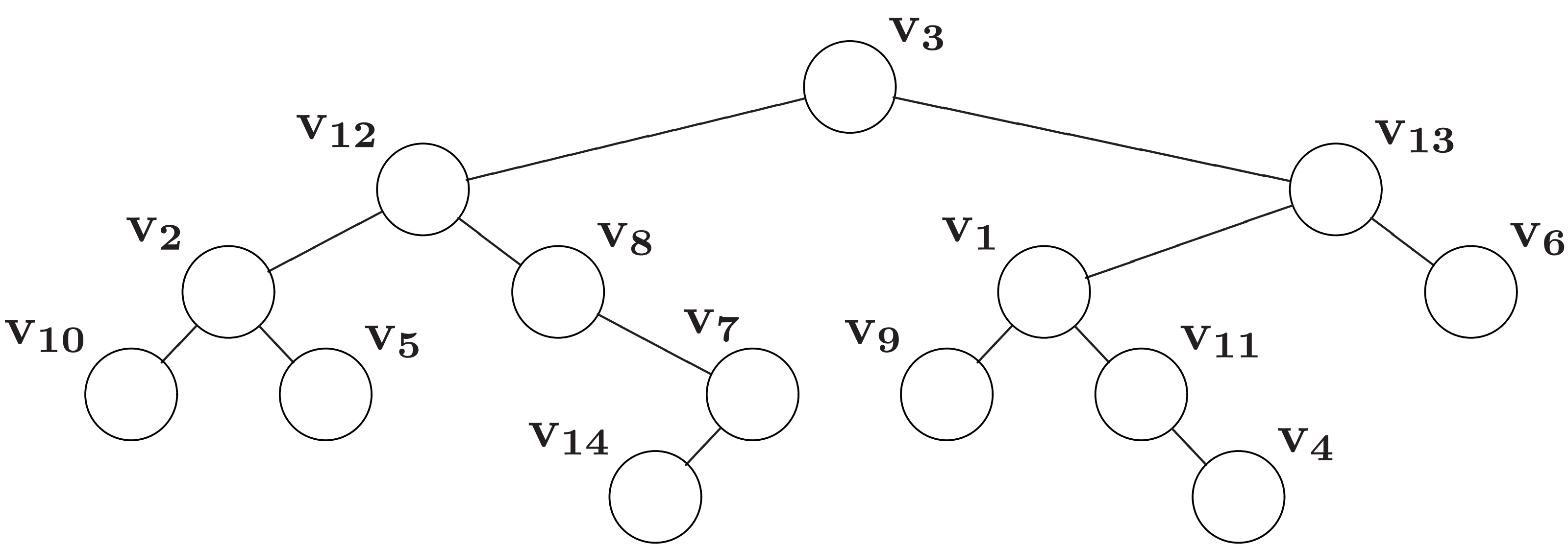
Inorder



- If node v_i is in the subtree rooted at v_j.`Left`, then $v_i \prec v_j$.
- If node v_k is in the subtree rooted at v_j.`Right`, then $v_j \prec v_k$.

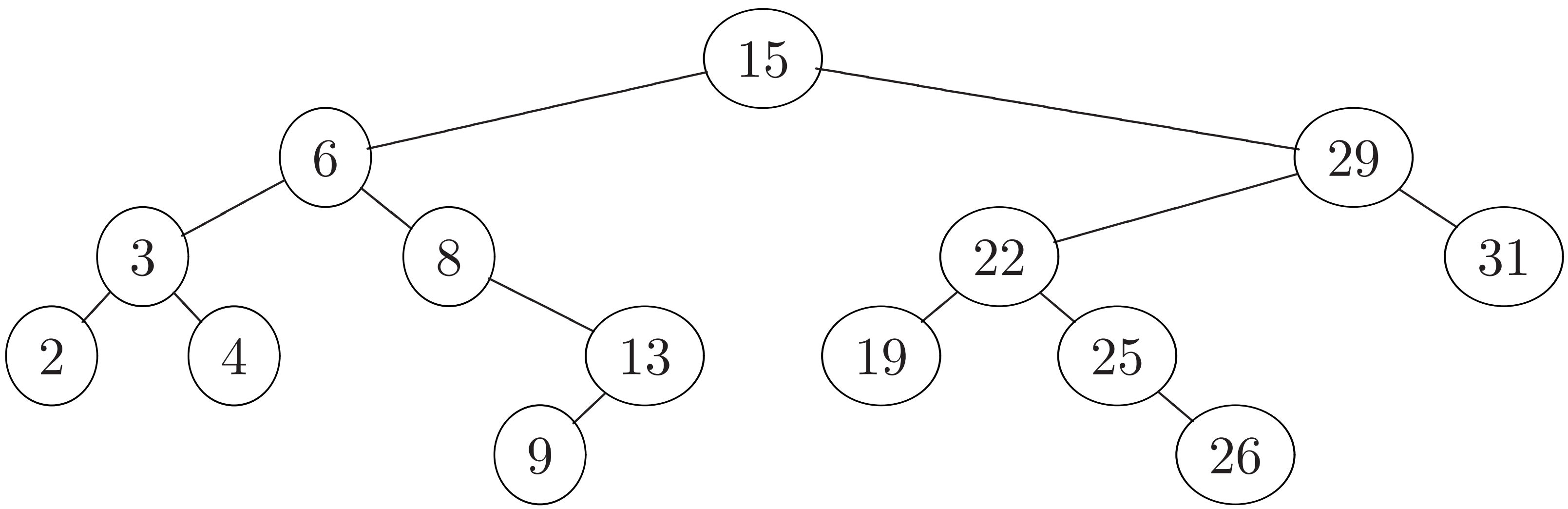
What is the inorder sequence of vertices?

Tree Successor



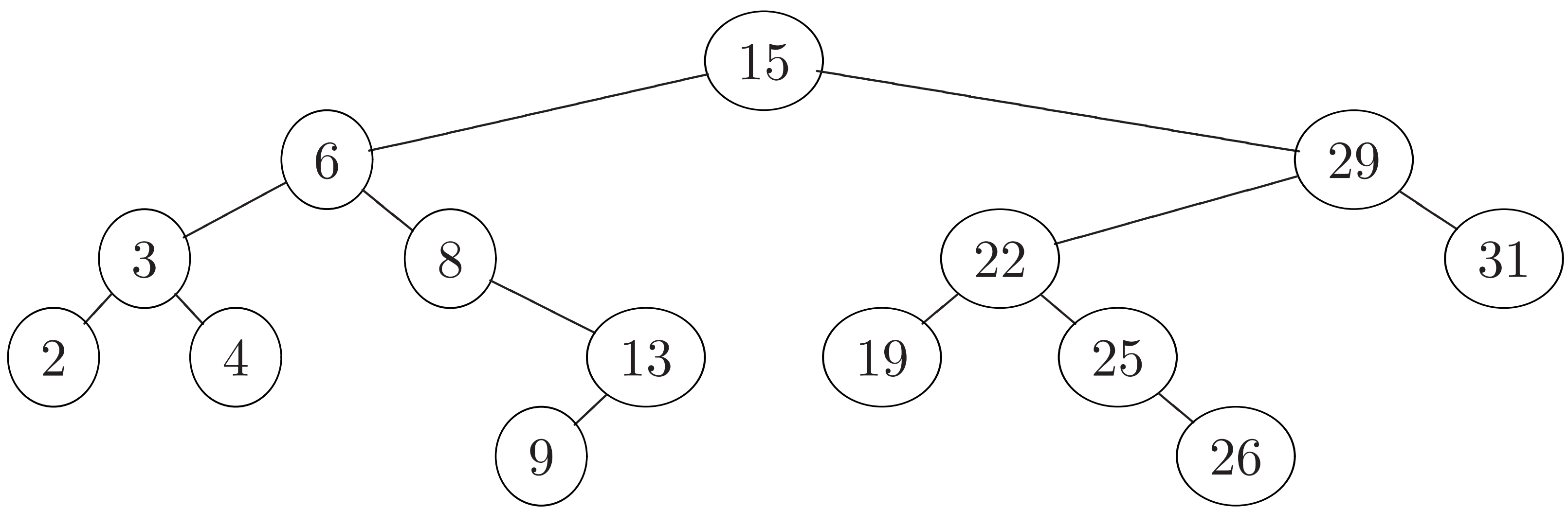
The **successor** of node v_i is the next node in the inorder sequence.

Inorder



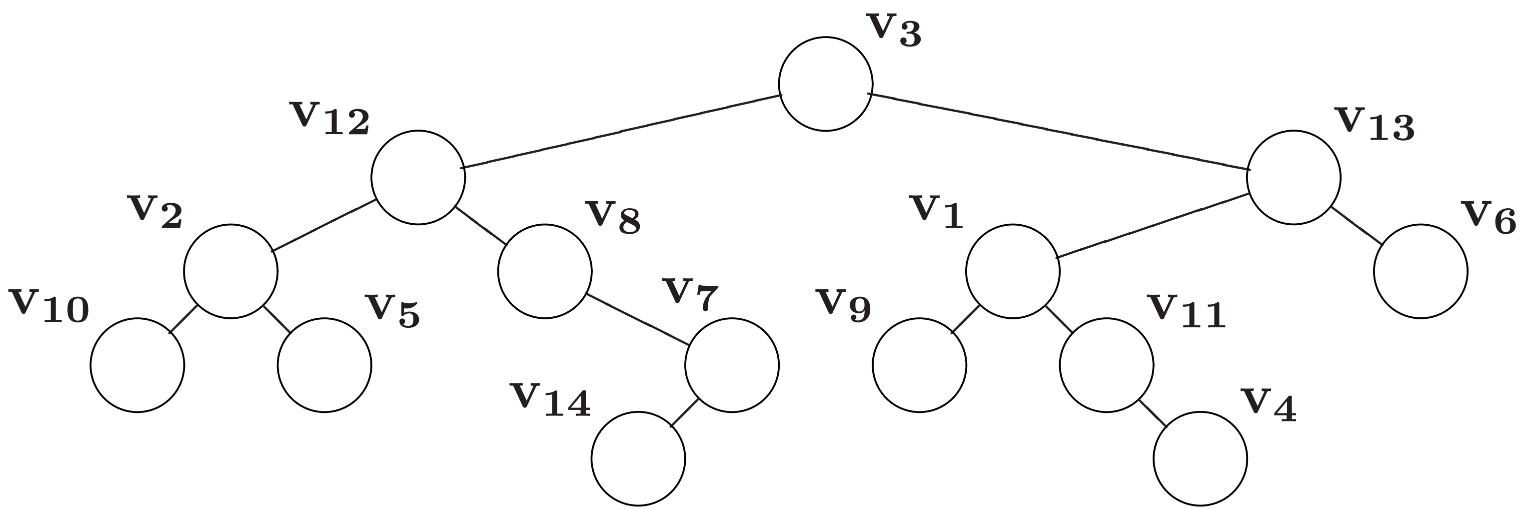
If T is a binary search tree and all the values at nodes are different, then nodes in the inorder sequence are ordered by increasing value.

Tree Successor



If T is a binary search tree and all the values at nodes are different, then the successor of node v_i is the node with smallest value greater than the value of v_i.

Tree Successor



Case I. x.right $\neq$ **NIL**:

Return `TreeMin`(x.right);

Case II. x.right $=$ **NIL**:

Return closest ancestor y of x where x is in left subtree of y.

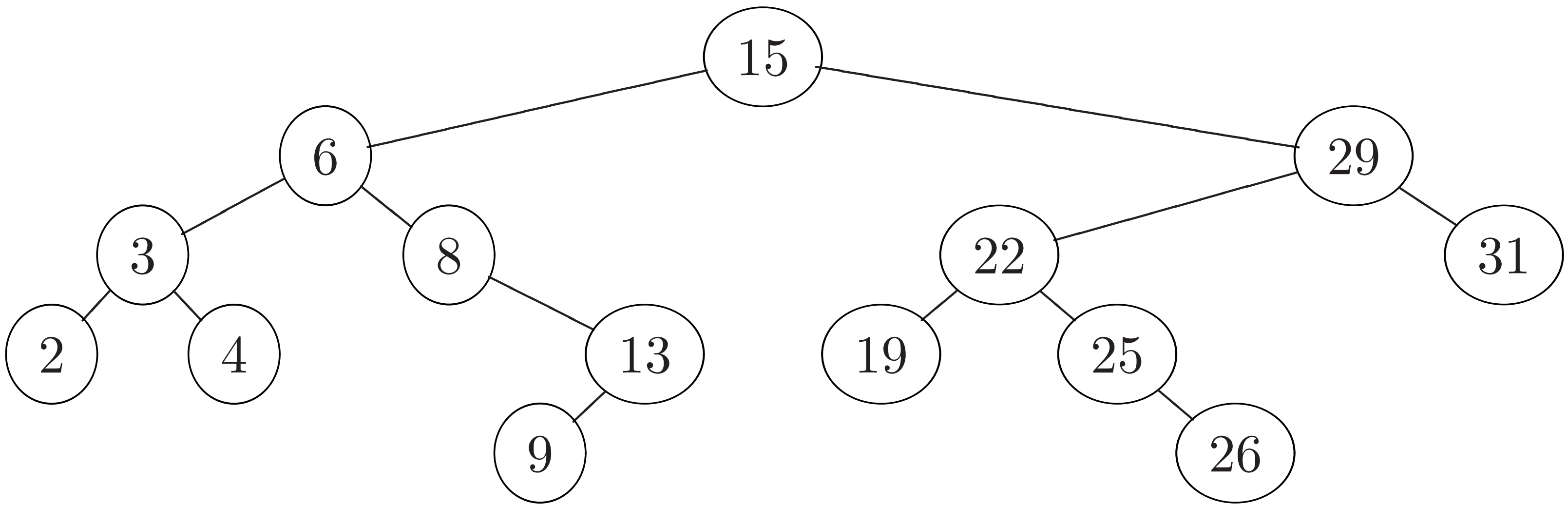
Tree Successor

```
procedure TreeSuccessor(x)
if (x.right ≠ NIL) then return (TreeMin(x.right));
y ← x.parent;
while (y ≠ NIL) and (x = y.right) do
    x ← y;
    y ← y.parent;
end
return (y);
```

Binary Search Trees:
Reporting, Searching and Counting

Report All Nodes

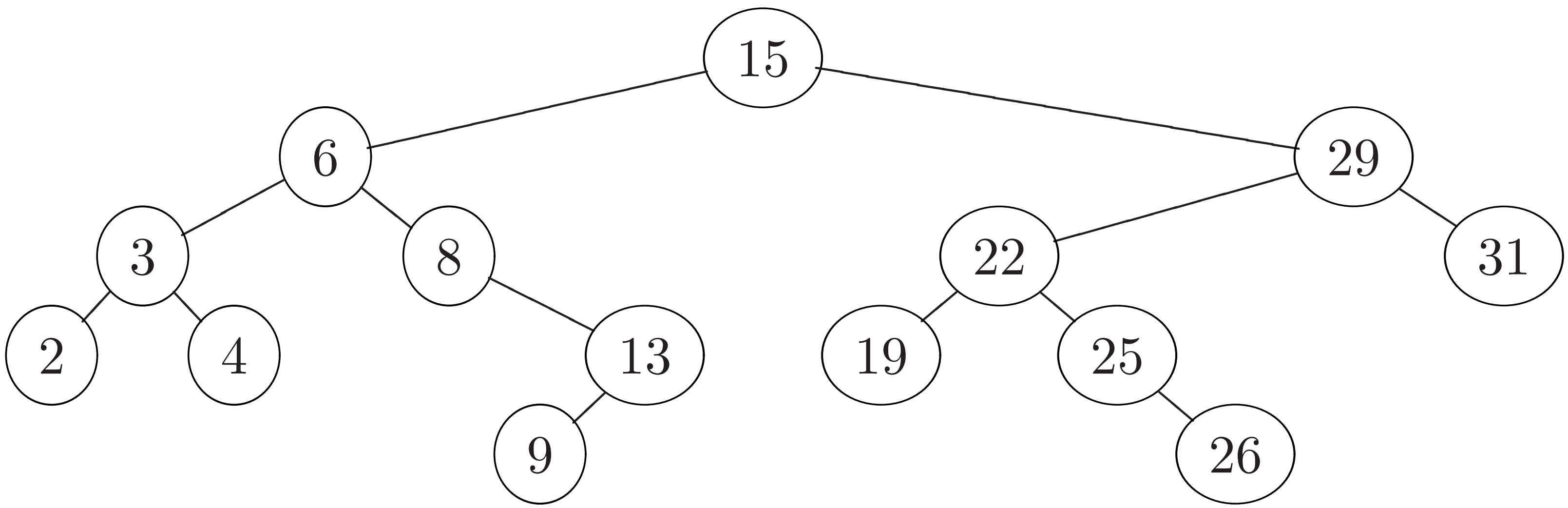
Report all nodes of the following tree:




```
procedure InorderTreeWalk(x)
if (x ≠ NIL) then
    InorderTreeWalk(x.left);
    print x.key;
    InorderTreeWalk(x.right);
end
```

Report in Range

Report all nodes of the following tree with keys in range $[8, 25]$:
(Range $[8, 25]$ includes 8 and 25.)



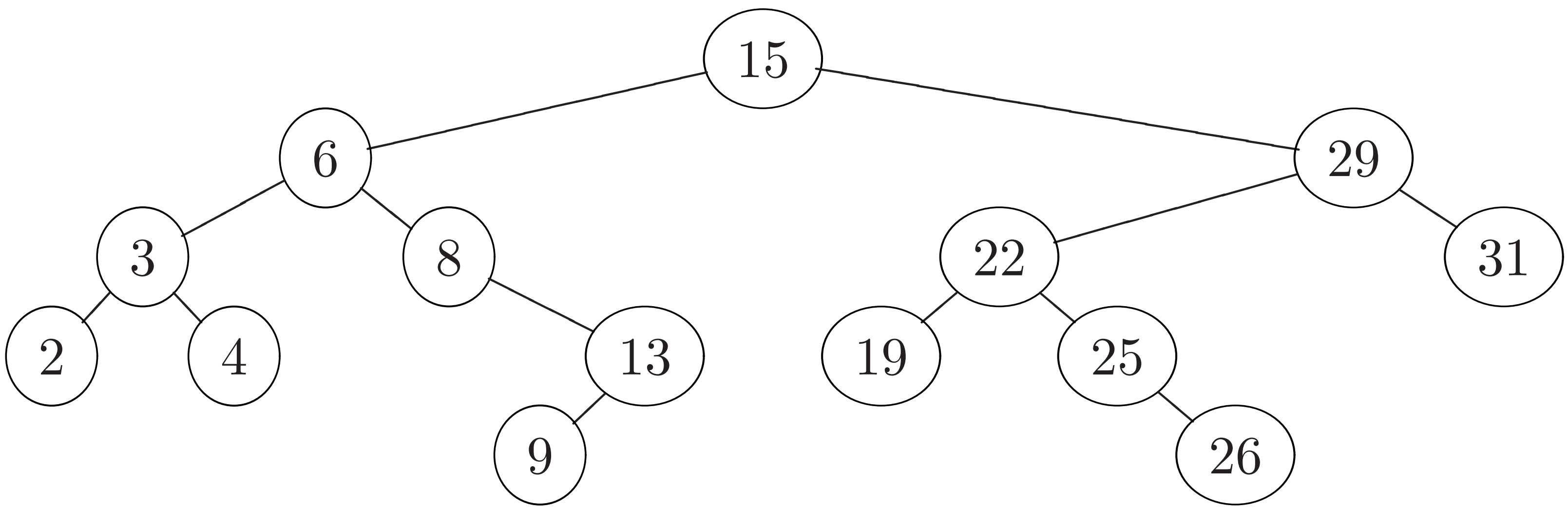
Report in Range

procedure `TreeRangeReport`(x, k_{min}, k_{max})

```
if (x ≠ NIL) then
    if (k_min ≤ x.key) then
        TreeRangeReport(x.left, k_min, k_max);
    end
    if (k_min ≤ x.key ≤ k_max) then  print x.key;
    if (x.key ≤ k_max) then
        TreeRangeReport(x.right, k_min, k_max);
    end
end
```

Report in Range

Report all nodes of the following tree with keys in range $[k_{min}, k_{max}]$:



h = tree height
I = number of nodes reported.

Running time:

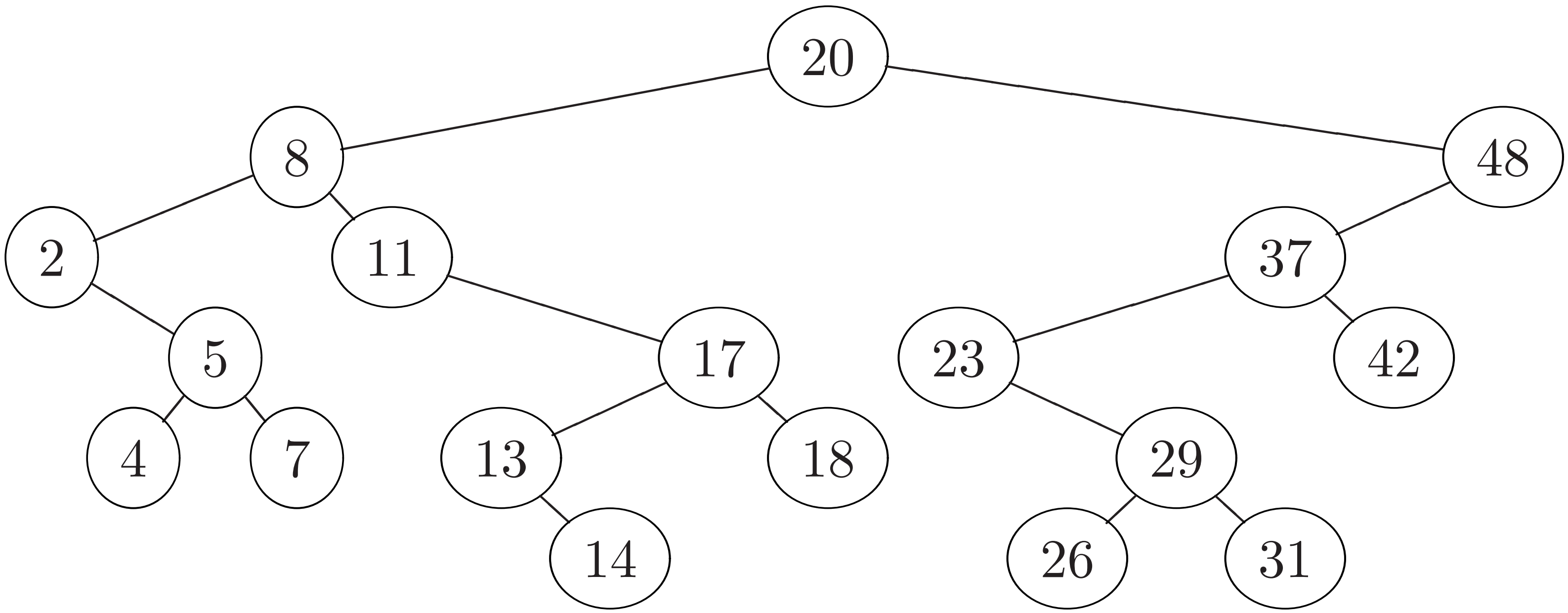
Report in Range: Running Time

```
procedure TreeRangeReport(x, k_min, k_max)
if (x ≠ NIL) then
    if (k_min ≤ x.key) then
        TreeRangeReport(x.left, k_min, k_max);
    end
    if (k_min ≤ x.key ≤ k_max) then print x.key;
    if (x.key ≤ k_max) then
        TreeRangeReport(x.right, k_min, k_max);
    end
end
```

h = tree height
I = number of nodes reported.

Running time:

Tree Minimum



procedure `TreeMin`(x)

while (x.left $\neq$ **NIL**) **do**

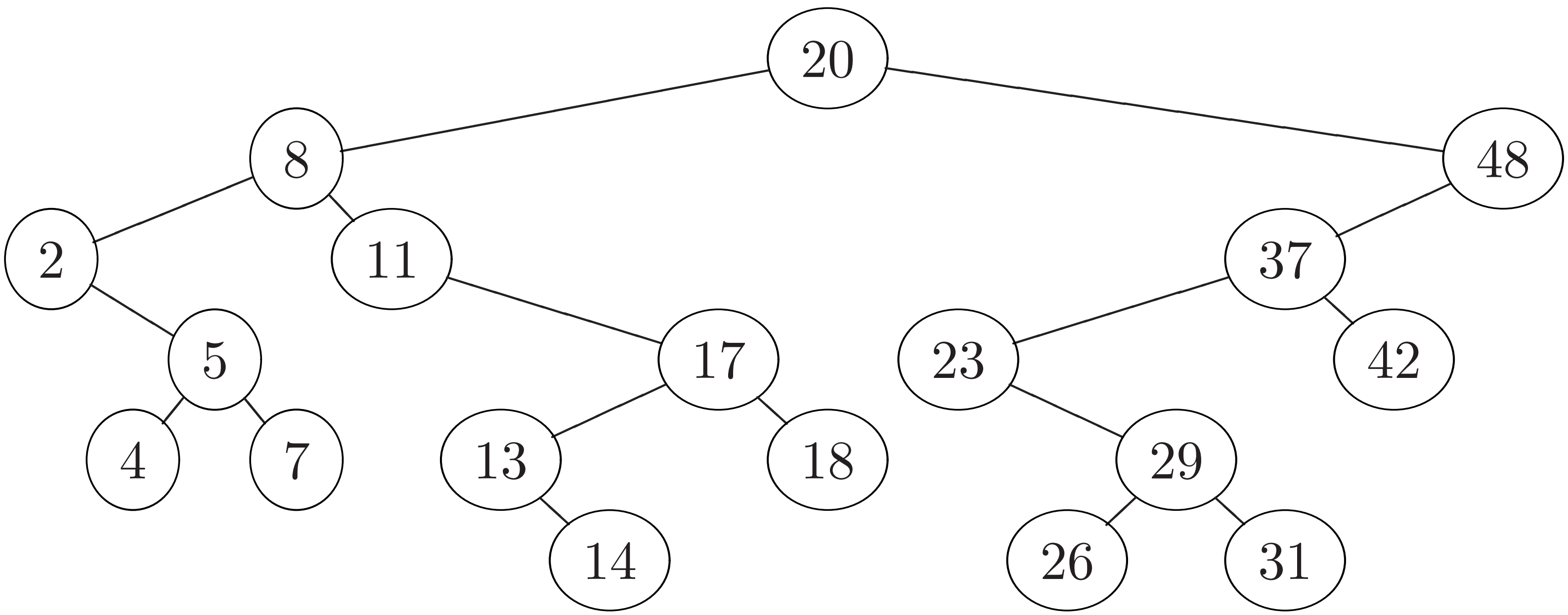
  $x \leftarrow x$.left;

end

return (x);

Tree Minimum Greater Than or Equal to

Find minimum key greater than or equal to 12.

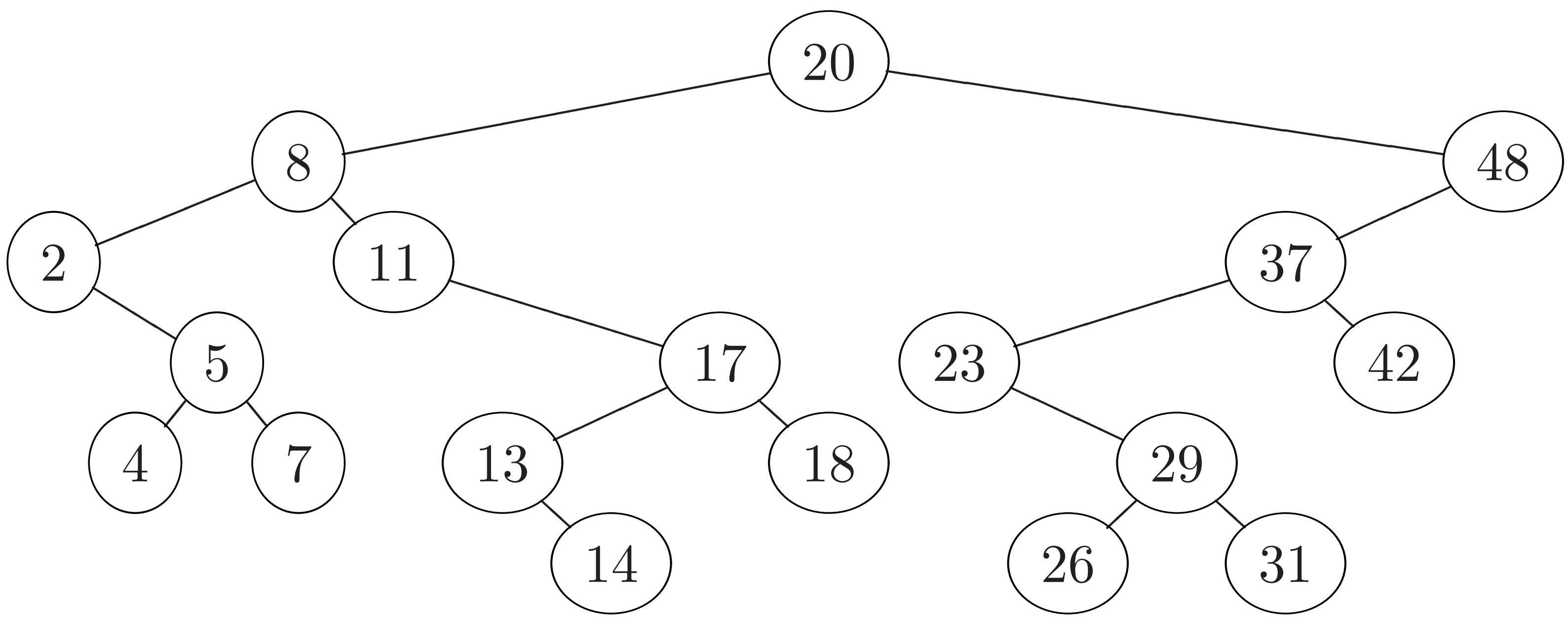


Tree Minimum Greater Than or Equal to

```
function TreeMinGE(T, K)
/* Return node with min key greater than or equal to K */
u ← NIL;
v ← T.root;
while (v ≠ NIL) do
    if (K ≤ v.key) then
        u ← v;
        v ← v.left;
    else
        v ← v.right;
    end
end
return (u);
```

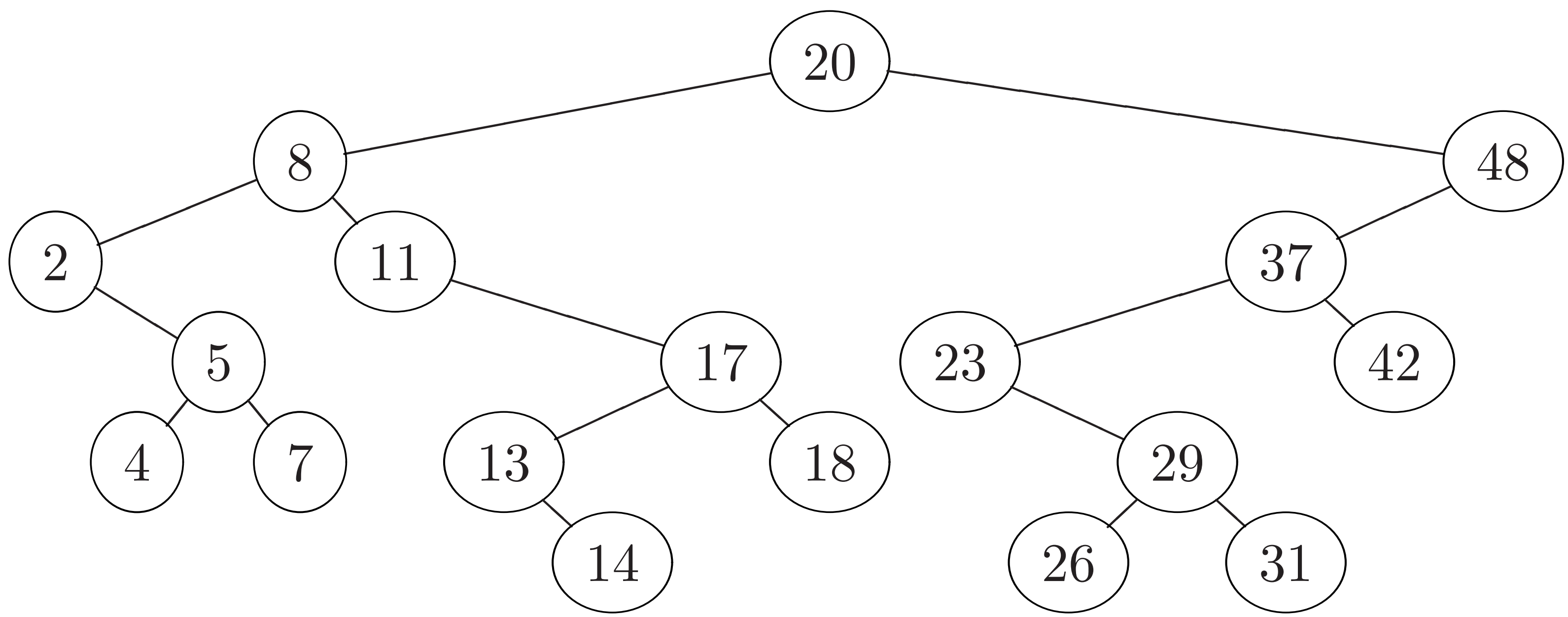
Tree Minimum Greater Than or Equal to

Find minimum key greater than or equal to 12.



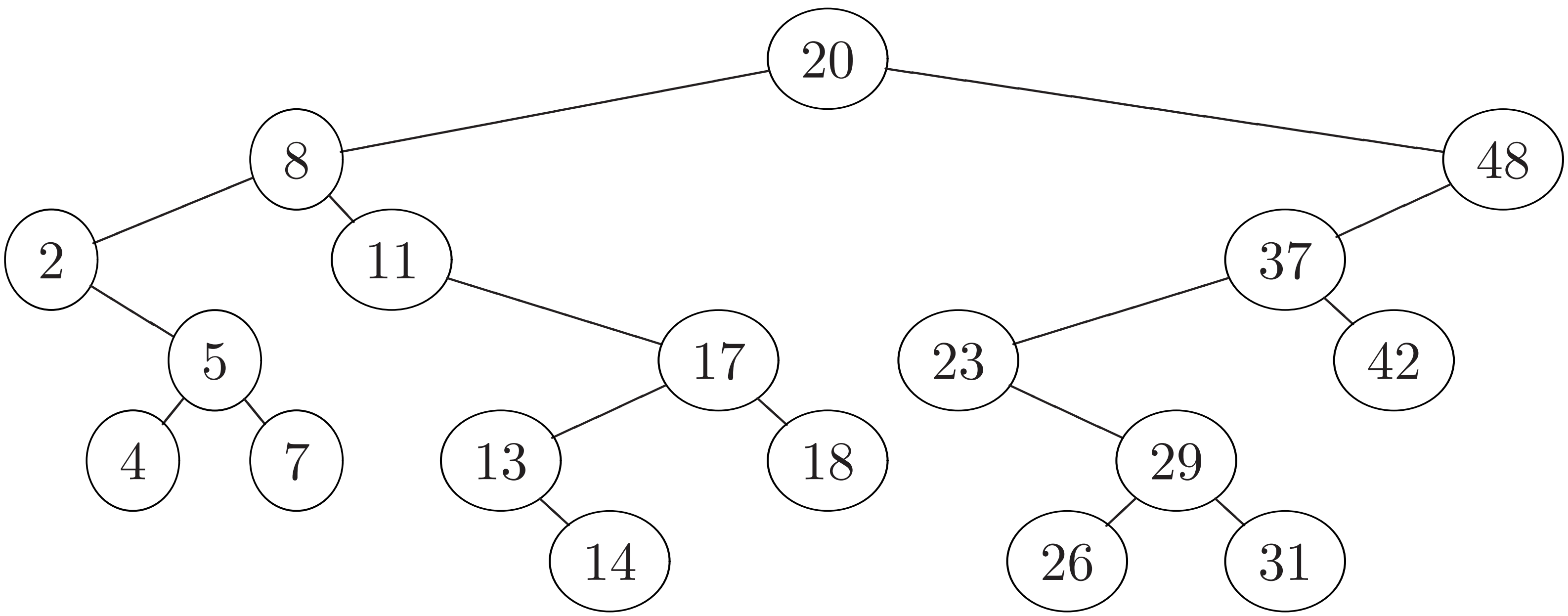
Tree Minimum Greater Than or Equal to

Find minimum key greater than or equal to 28.



Count Nodes in Range

Count number of nodes with keys in range [6, 42].
(Range [6, 42] includes 6 and 42.)

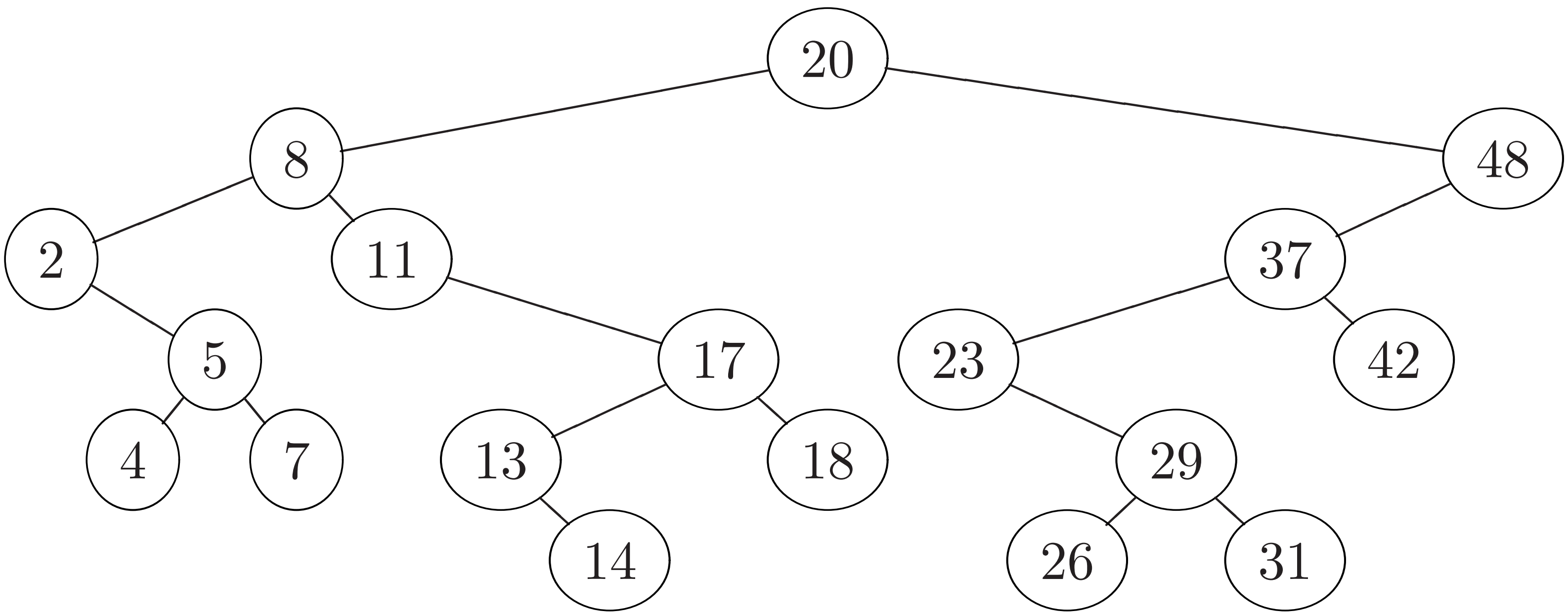


Report in Range

procedure `TreeRangeReport`(x, k_{min}, k_{max})
if ($x \neq$ **NIL**) **then**
    if ($k_{min} \leq x$.key) **then**
        `TreeRangeReport`(x.left, k_{min}, k_{max});
    end
    if ($k_{min} \leq x$.key $\leq k_{max}$) **then** **print** x.key;
    if (x.key $\leq k_{max}$) **then**
        `TreeRangeReport`(x.right, k_{min}, k_{max});
    end
end

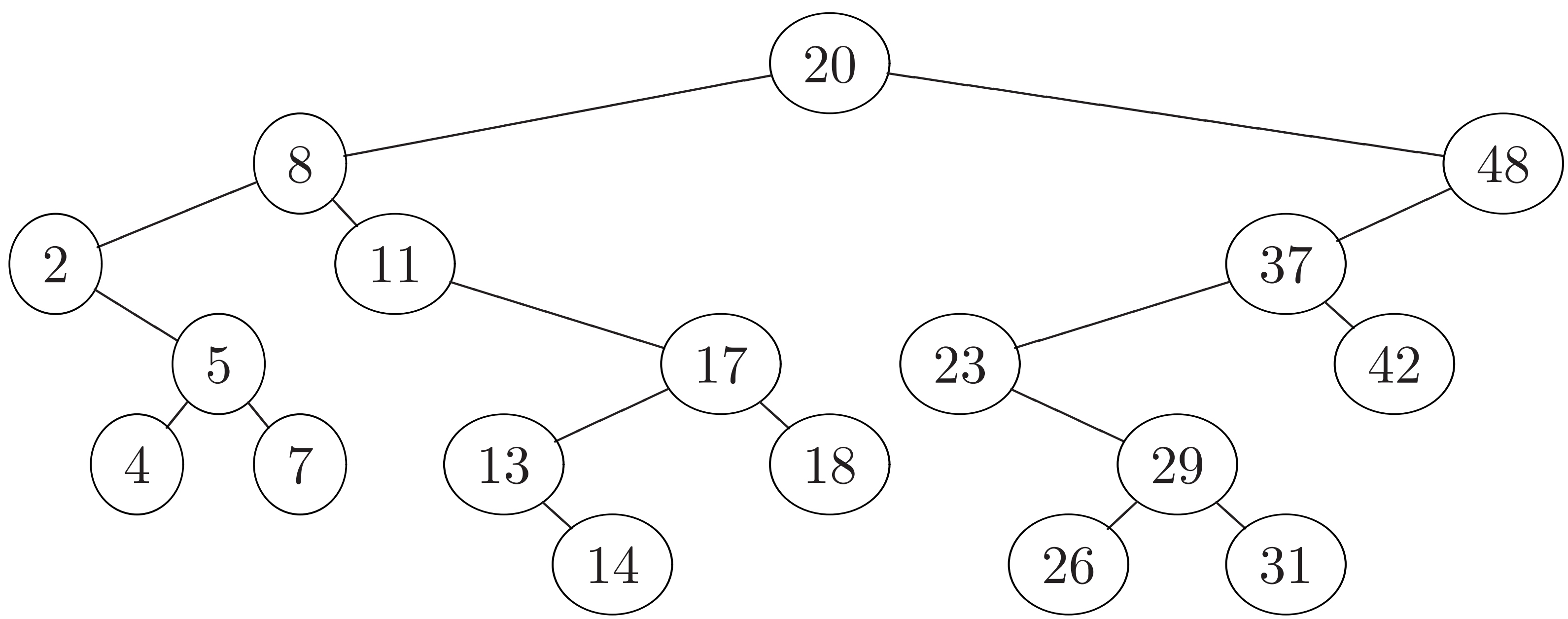
Count Nodes Greater Than or Equal to

Count number of nodes with keys greater than or equal to 6.

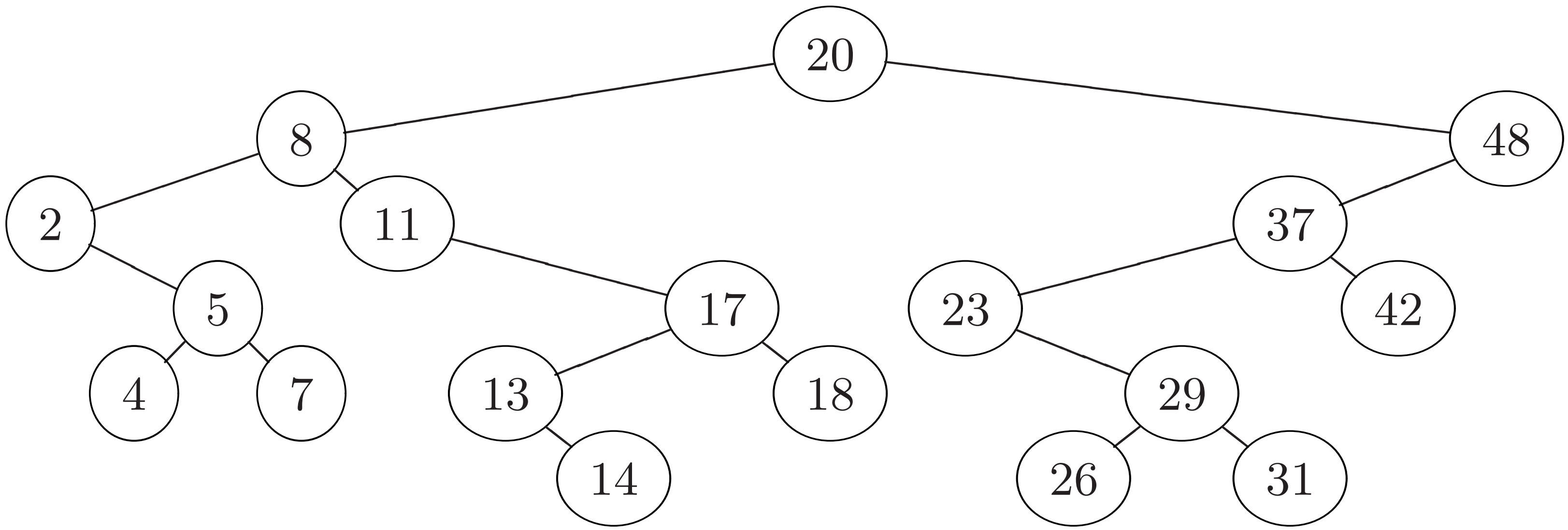


Binary Search Tree: Counting

For each node u in the binary search tree, add a field $u.\mathsf{size}$ which stores the number of nodes in the subtree rooted at u.



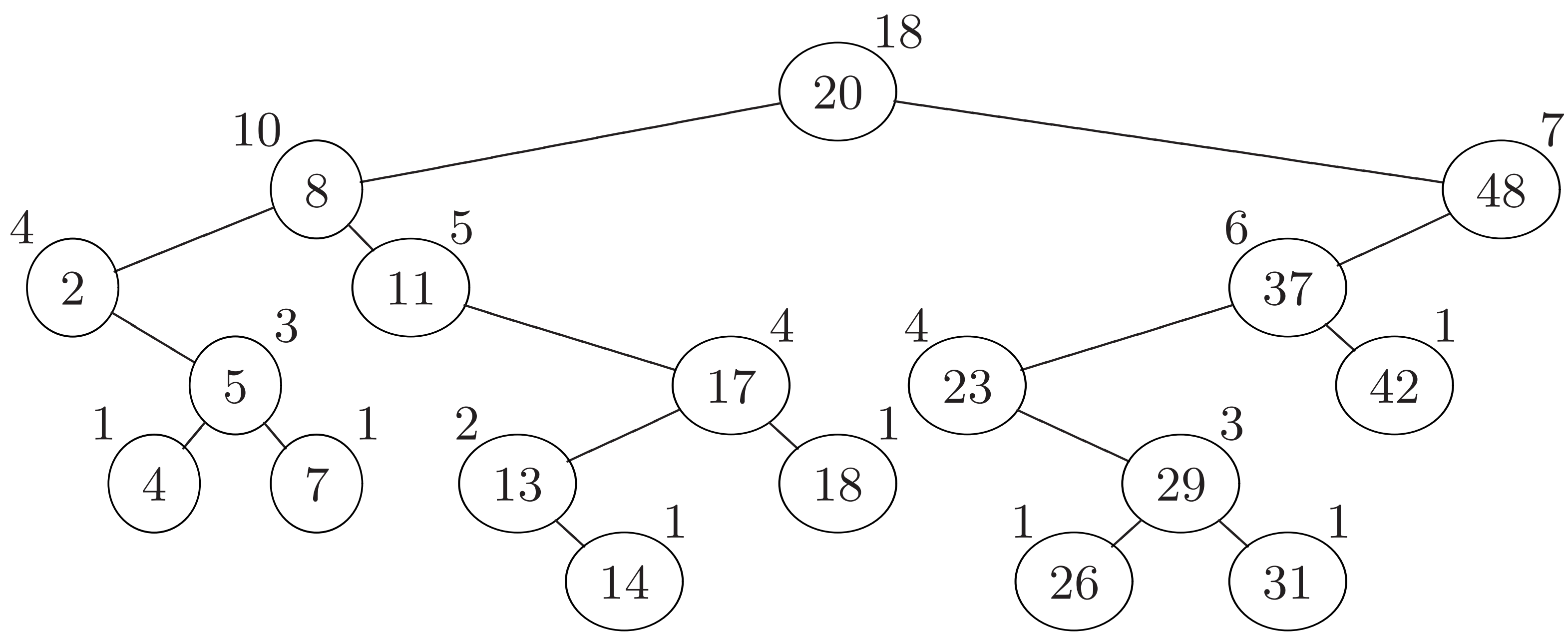
Compute Size




```
procedure TreeComputeSize(x)
if (x ≠ NIL) then
    TreeComputeSize (x.left);
    TreeComputeSize (x.right);
    x.size ← 1;
    if (x.left ≠ NIL) then  x.size ← x.size + x.left.size;
    if (x.right ≠ NIL) then  x.size ← x.size + x.right.size;
end
```

Count Nodes Greater Than or Equal to

Count number of nodes greater than or equal to 6.

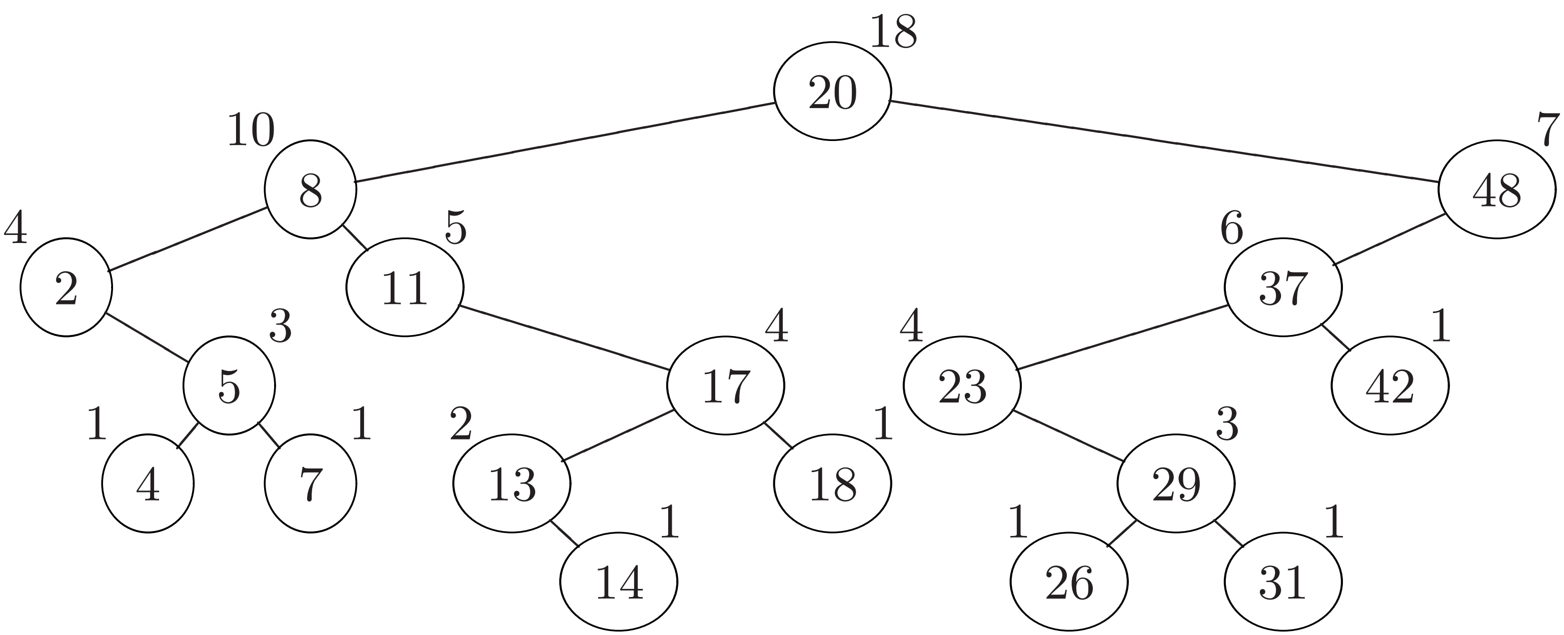


Count Nodes Greater Than or Equal to

```
function TreeCountGE(x, K)
/* Return number of nodes with keys greater than or equal
   to K in subtree rooted at x                                  */
count ← 0;
v ← x;
while (v ≠ NIL) do
    if (v.key ≥ K) then
        count ← count + 1;
        if (v.right ≠ NIL) then  count ← count + v.right.size;
        v ← v.left;
    else
        v ← v.right;
    end
end
return (count);
```

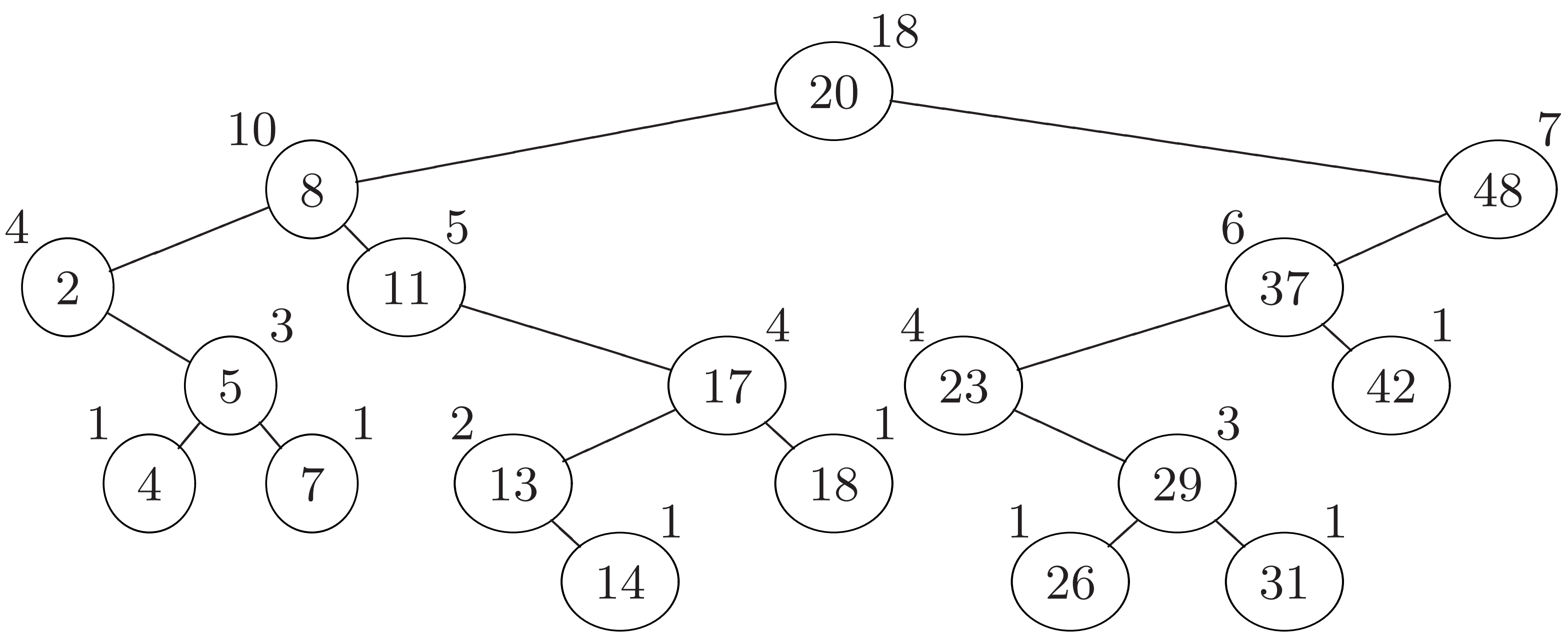
Count Nodes Greater Than or Equal to

Count number of nodes greater than or equal to 6.
Count number of nodes greater than or equal to 17.



Count Nodes Less Than or Equal to

Count number of nodes less than or equal to 42.

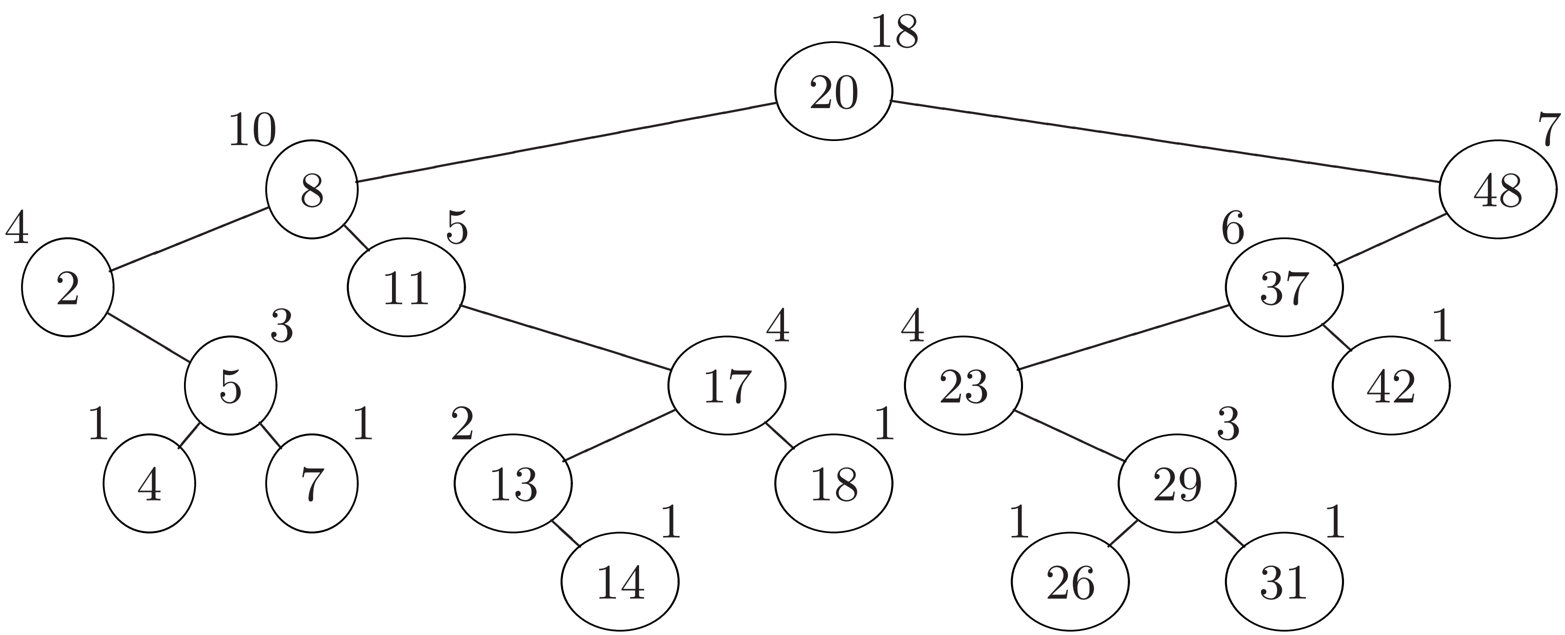


Count Number of Nodes Less Than or Equal to

```
function TreeCountLE(x, K)
/* Return number of nodes with keys less than or equal to K
   in subtree rooted at x                                    */
count ← 0;
v ← x;
while (v ≠ NIL) do
    if (v.key ≤ K) then
        count ← count + 1;
        if (v.left ≠ NIL) then  count ← count + v.left.size;
        v ← v.right;
    else
        v ← v.left;
    end
end
return (count);
```

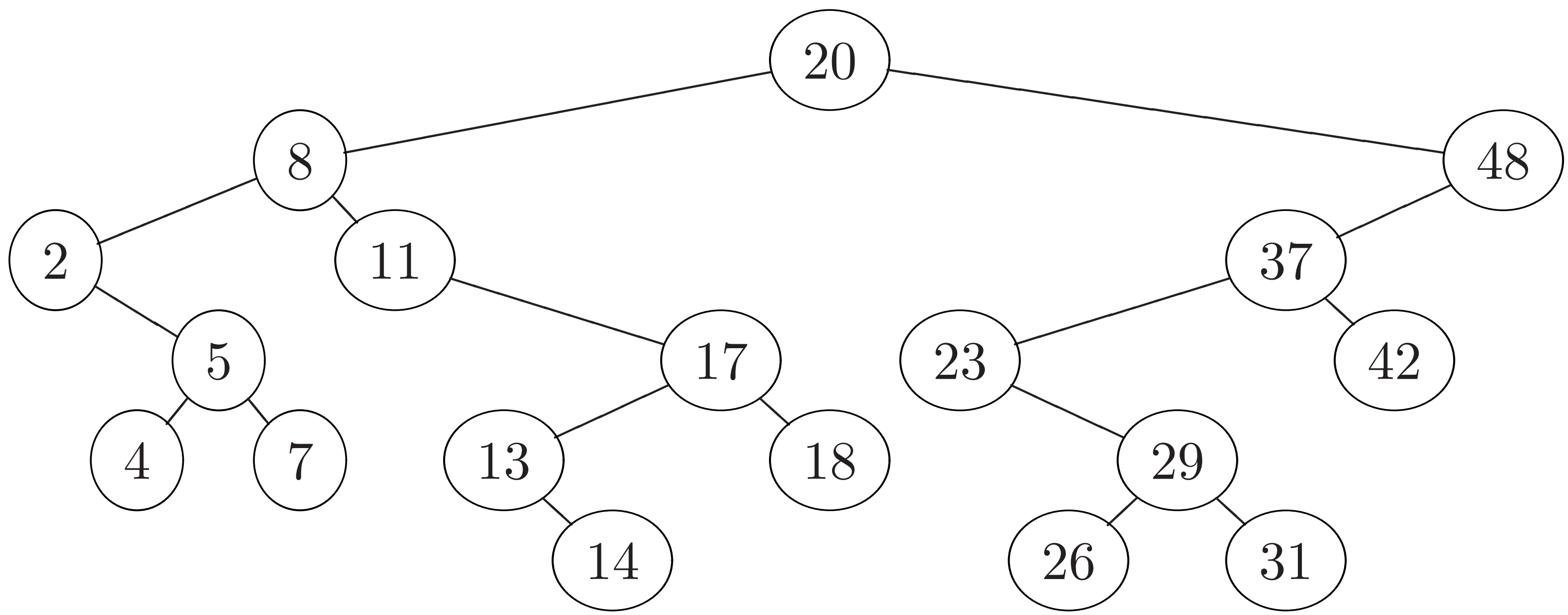
Count Nodes Less Than or Equal to

Count number of nodes less than or equal to 42.
Count number of nodes less than or equal to 16.



Count Nodes in Range

Count number of nodes with keys in range $[6, 42]$.
(Range $[6, 42]$ includes 6 and 42.)



Count Number of Nodes in Range

function `TreeRangeCount(`x, k_{min}, k_{max}`)`

/* *Return number of nodes with keys in range* $[k_{\min}, k_{\max}]$ *in subtree rooted at* x */

$v \leftarrow x$;

while $(v \neq$ **NIL**$)$ **and** $(v.\mathsf{key} \notin [k_{min}, k_{max}])$ **do**

  if $(v.\mathsf{key} \leq k_{min})$ **then** $v \leftarrow v.\mathsf{right}$;

  else if $(v.\mathsf{key} \geq k_{max})$ **then** $v \leftarrow v.\mathsf{left}$;

end

if $(v \neq$ **NIL**$)$ **then**

  $\mathsf{count}_L \leftarrow$ `TreeCountGE`$(v.\mathsf{left}, k_{min})$;

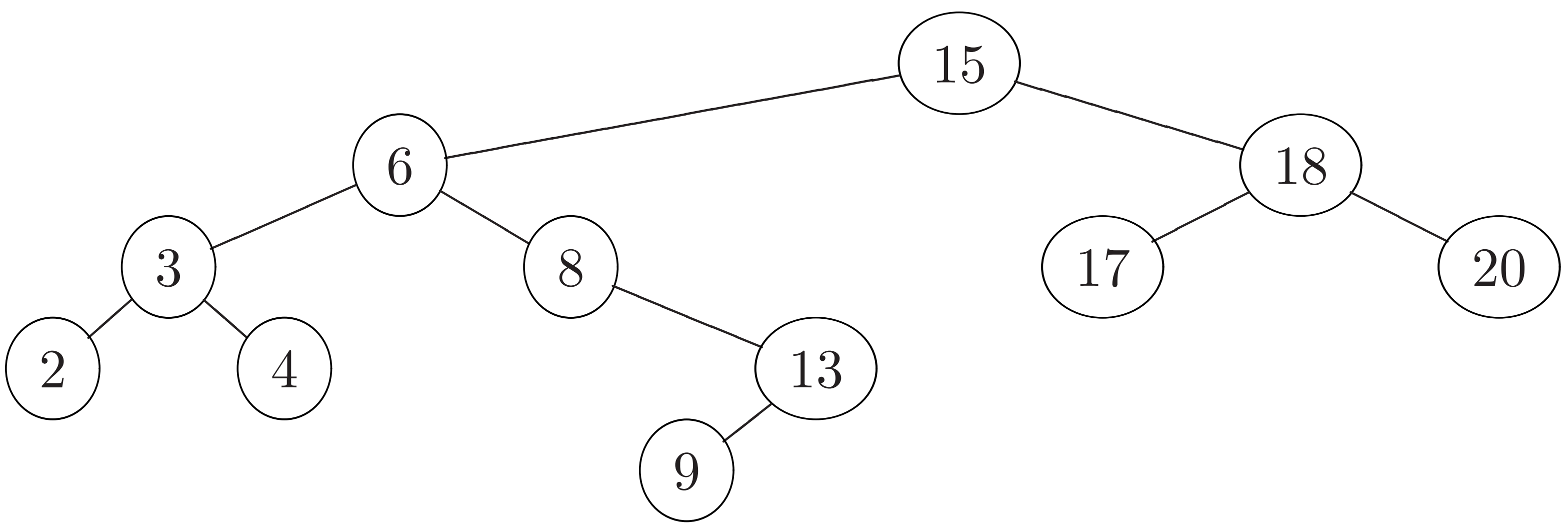
  $\mathsf{count}_R \leftarrow$ `TreeCountLE`$(v.\mathsf{right}, k_{max})$;

  return $(1 + \mathsf{count}_L + \mathsf{count}_R)$;

else **return** (0);

Binary Search Trees: Insertion

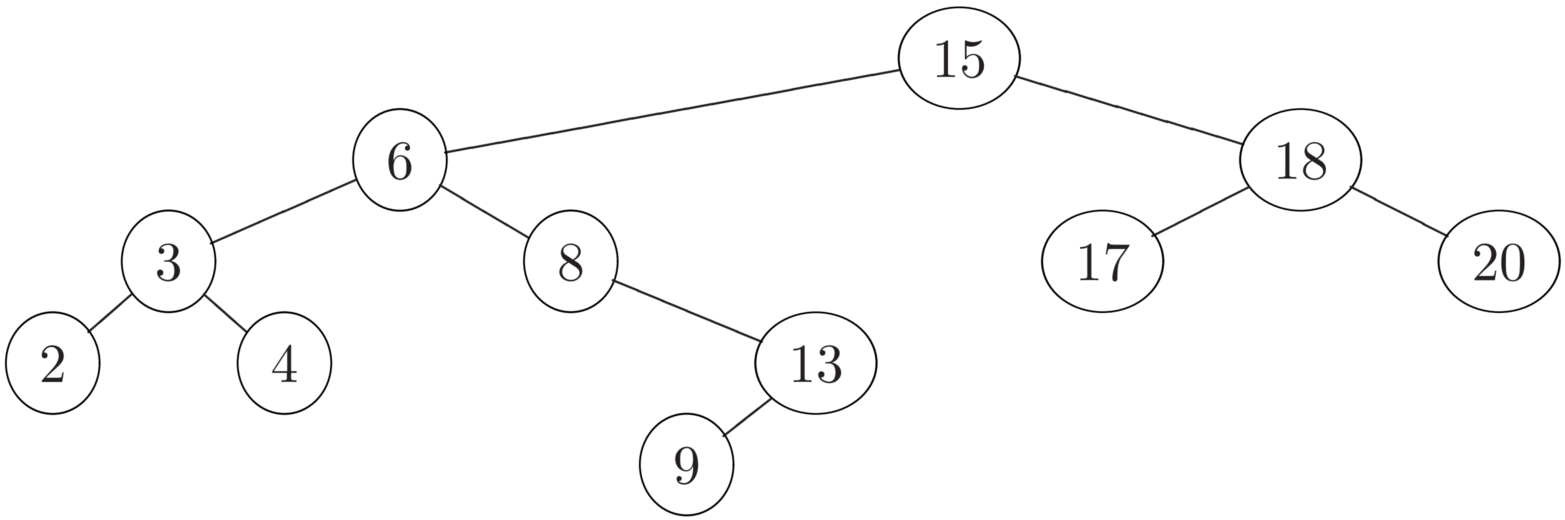
Tree Insert



Tree Insert

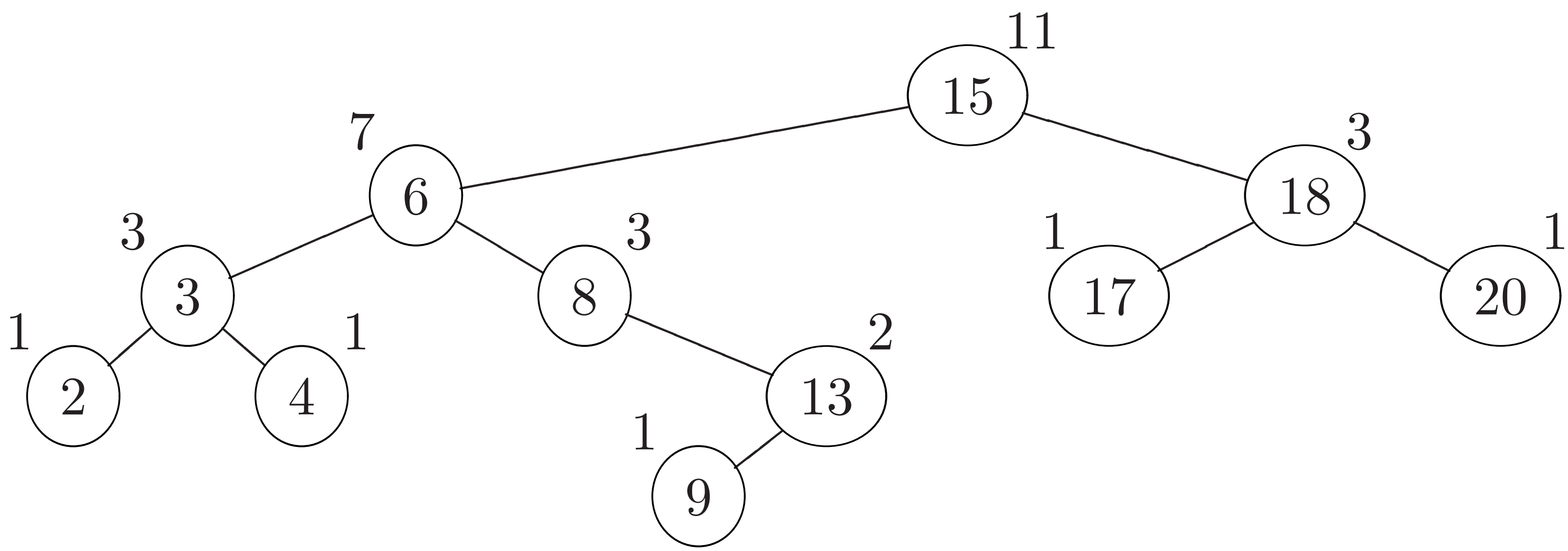
```
function LocateParent(T, z)
/* Return future parent of z in tree                              */
y ← NIL;
x ← T.root;
while (x ≠ NIL) do
    y ← x;
    if (z.key < x.key) then  x ← x.left;
    else  x ← x.right;
end
return (y);
```

Tree Insert

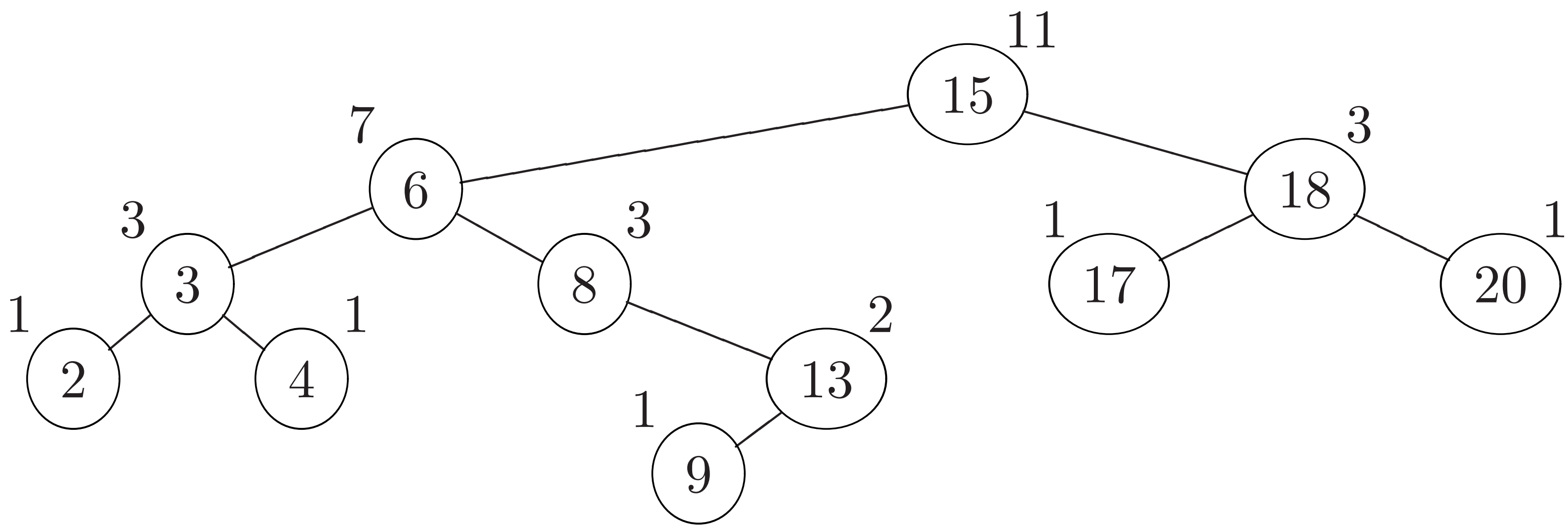



```
function TreeInsert(T, z)
y ← LocateParent(T, z);
z.parent ← y;
if (y = NIL) then T.root ← z;            /* tree T was empty*/
else if (z.key < y.key) then y.left ← z;
else y.right ← z;
```

Tree Insert: Size



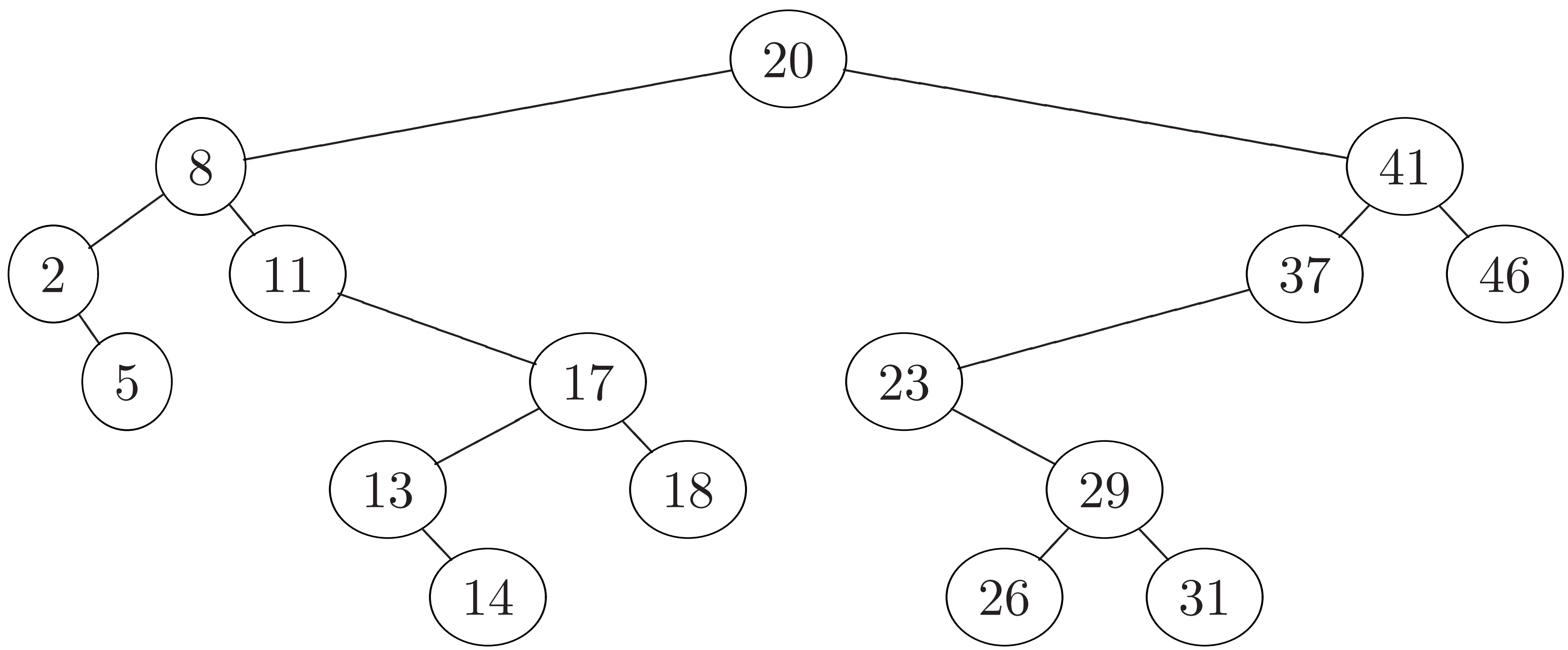
Update Size



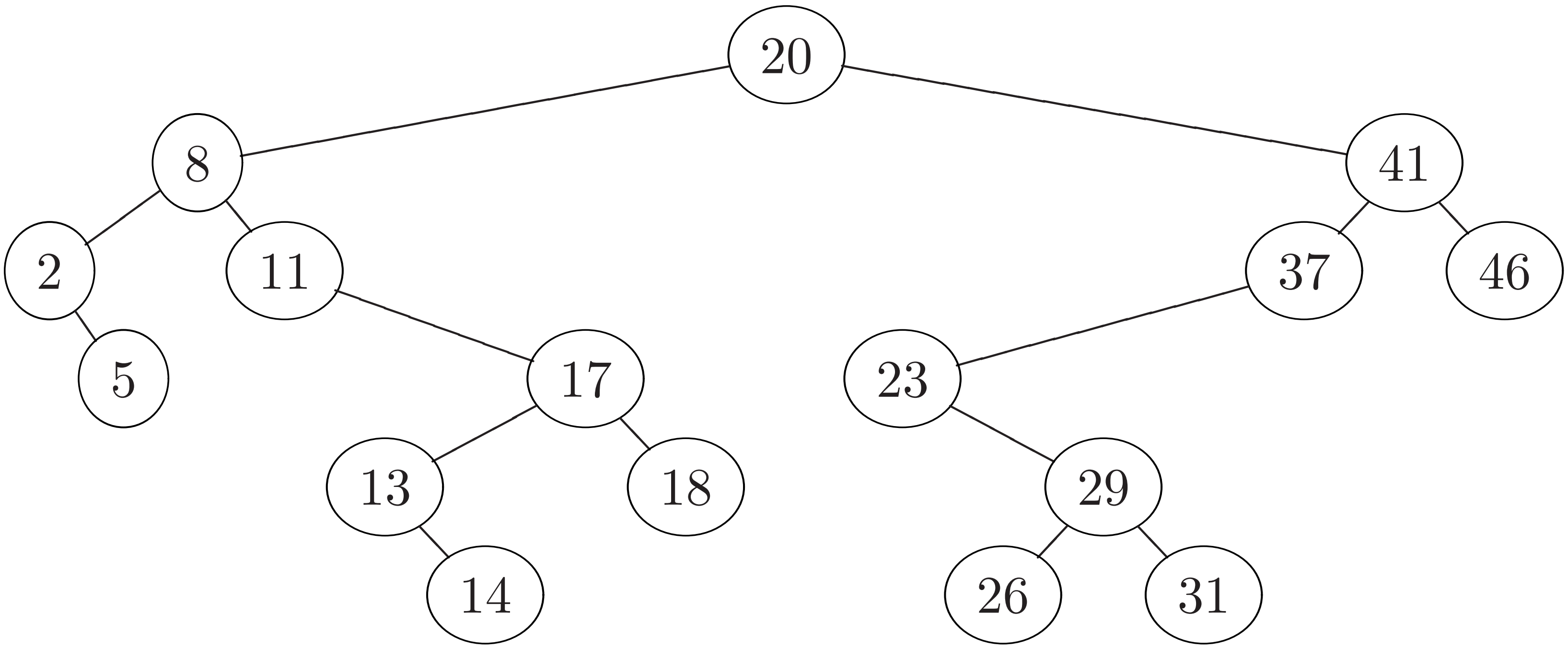

```
function InsertAndUpdateSize(T, z)
TreeInsert(T, z);
z.size ← 1;
y ← z.parent;
while (y ≠ NIL) do
    y.size ← y.size + 1;
    y ← y.parent;
end
```

Binary Search Trees:
Deletion

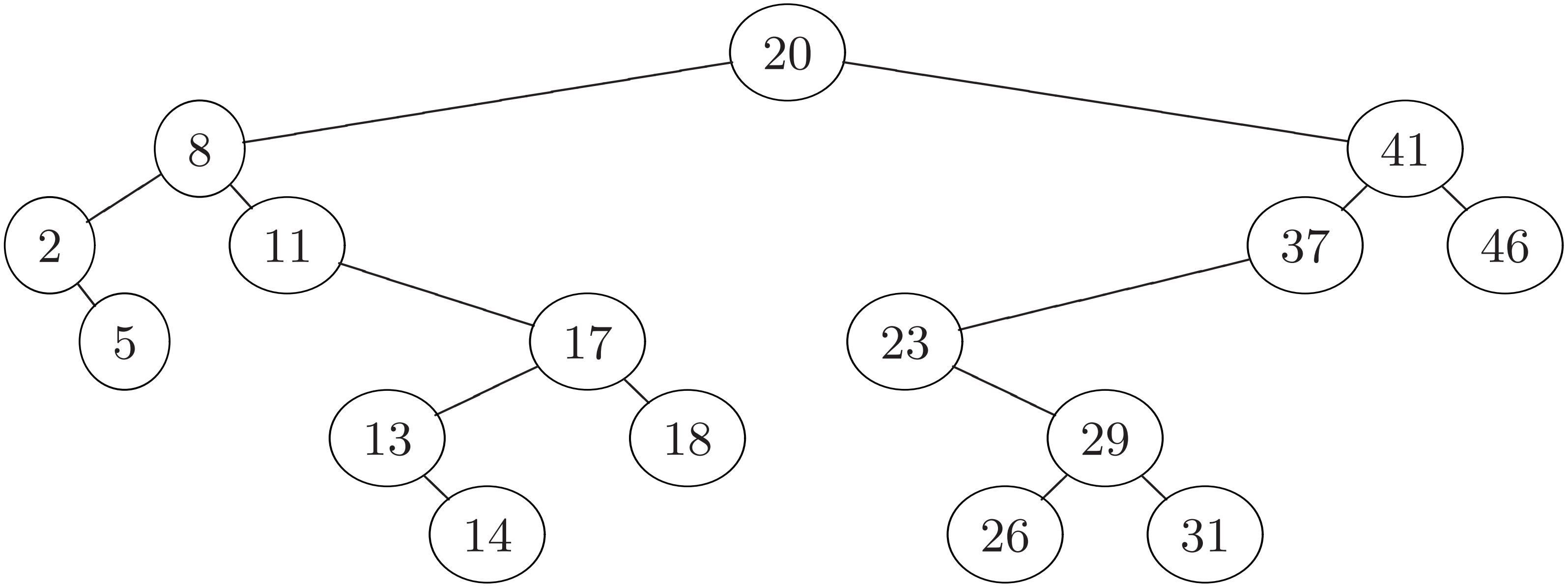
Tree Delete



Tree Delete



Tree Delete



Case I. Node z has zero children: Delete z.

Case II. Node z has one child: Replace z with its child.

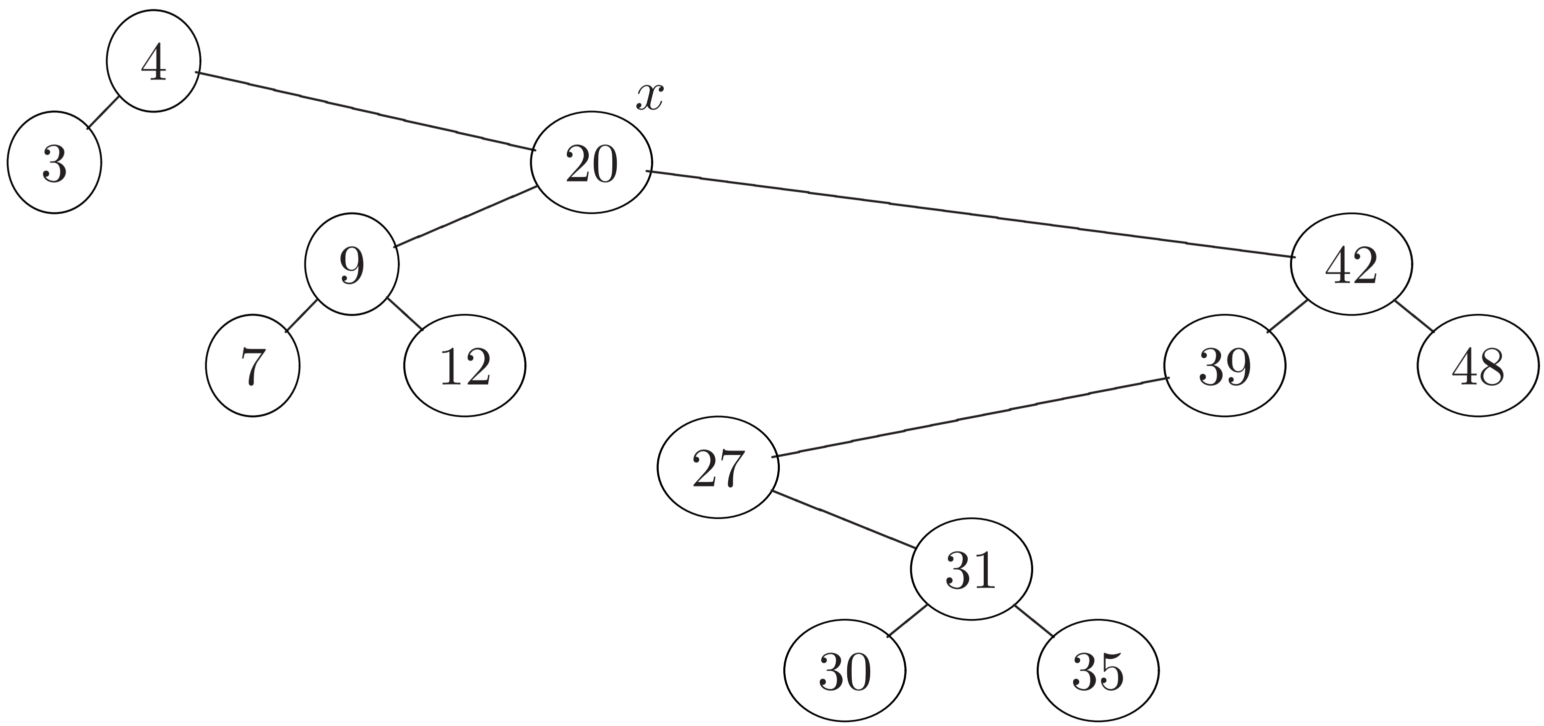
Case III. Node z has two children:

Replace z with its successor y.

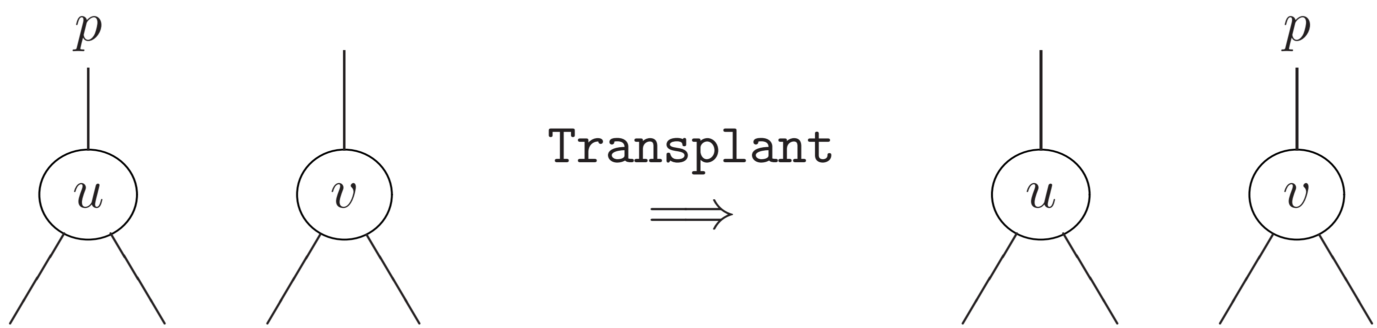
Replace y with its right child.

Tree Delete: Exercise

Delete node x from the following tree:



Transplant

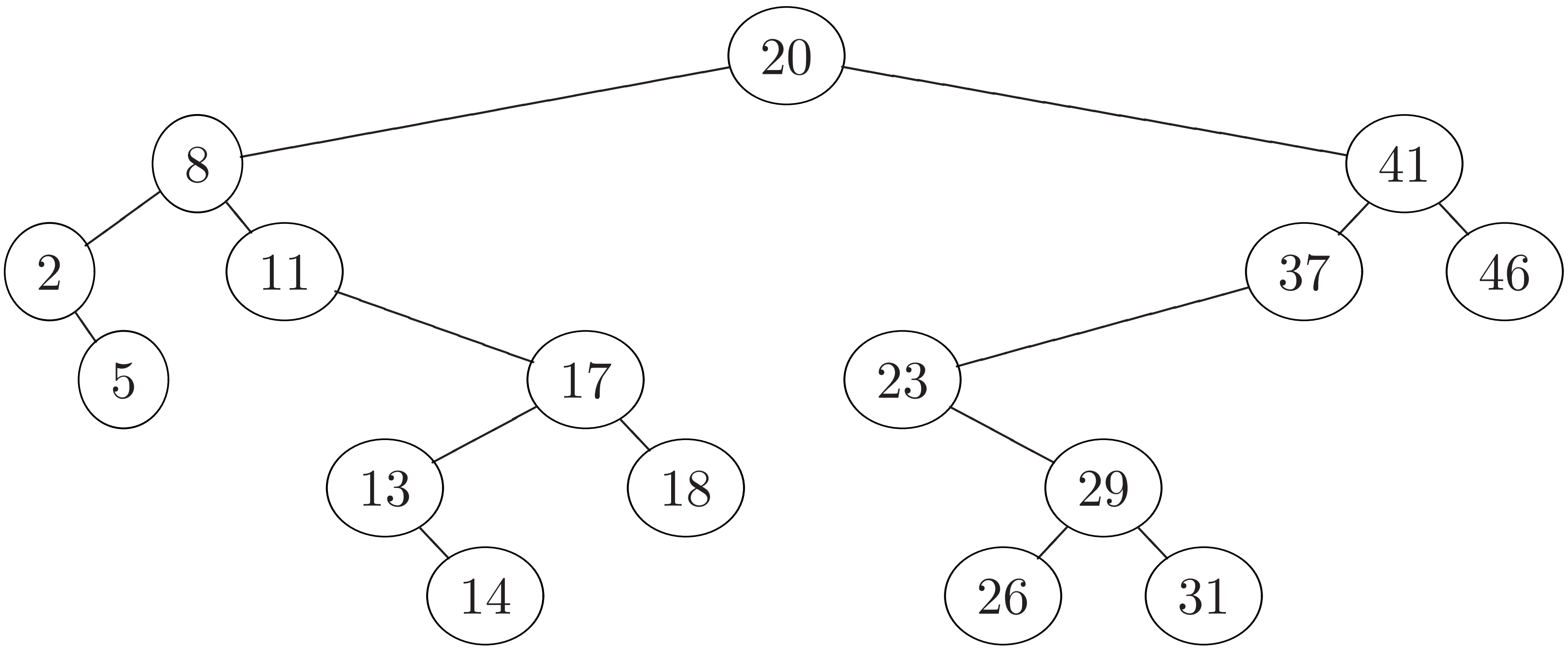



```
function Transplant(T, u, v)
/* Replace subtree rooted at u with subtree rooted at v.        */
p ← u.parent;
if (p = NIL) then T.root ← v;
else if (u = p.left) then  p.left ← v;
else p.right ← v;
if (v ≠ NIL) then v.parent ← p;
```

Tree Delete

```
function TreeDelete(T, z)
if (z.left = NIL) then Transplant(T,z,z.right);
else if (z.right = NIL) then Transplant(T,z,z.left);
else
    y ← TreeMin(z.right) ;          /* y is the successor of z */
    if (y ≠ z.right) then
        Transplant(T, y, y.right);
        y.right ← z.right;
        y.right.parent ← y;
    end
    Transplant(T, z, y);
    y.left ← z.left;
    y.left.parent ← y;
end
```

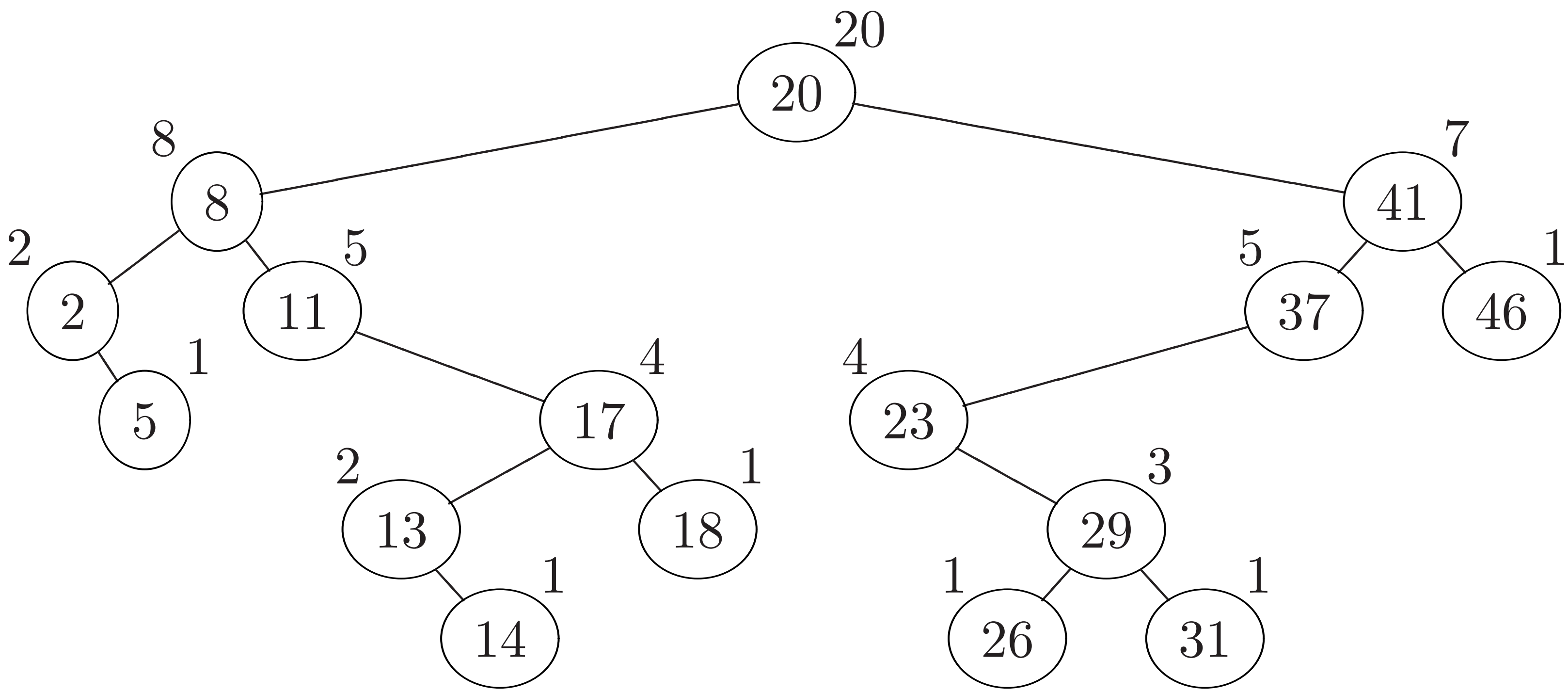
Tree Delete



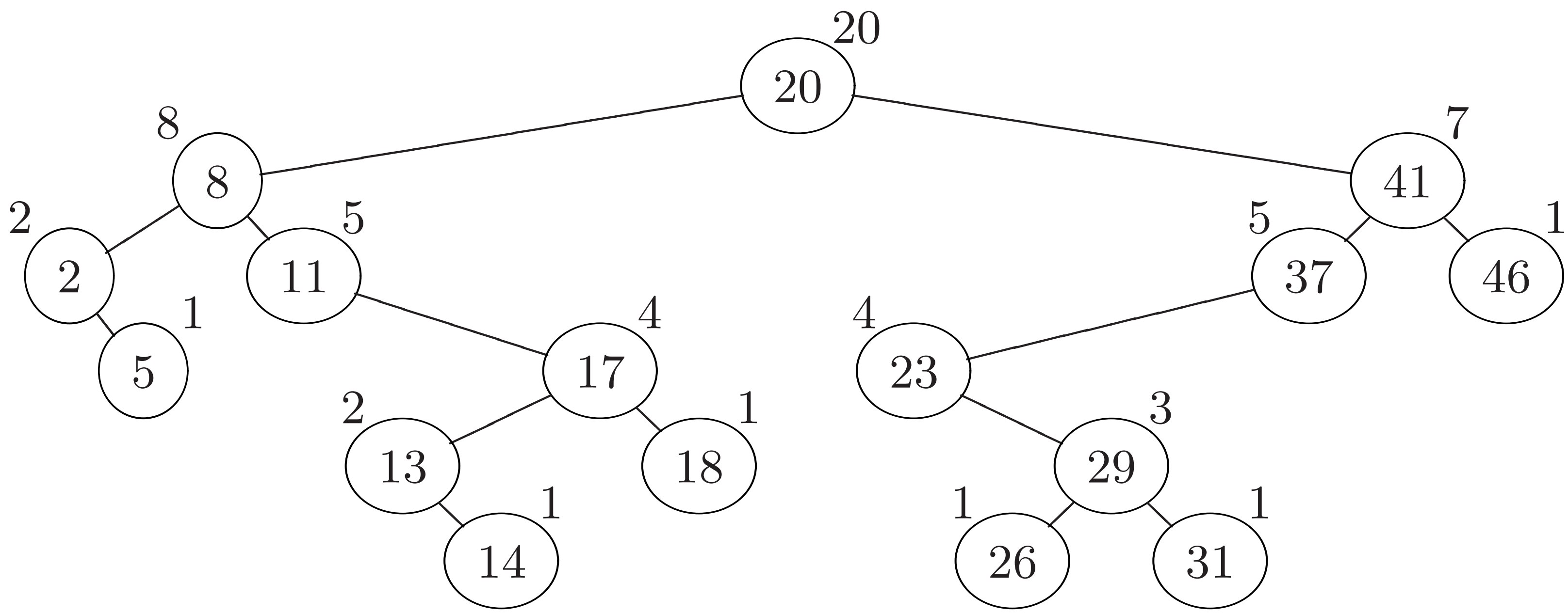
Running Time Analysis: TreeDelete

```
function TreeDelete(T, z)
if (z.left = NIL) then Transplant(T,z,z.right);
else if (z.right = NIL) then Transplant(T,z,z.left);
else
    y ← TreeMin(z.right) ;          /* y is the successor of z */
    if (y ≠ z.right) then
        Transplant(T, y, y.right);
        y.right ← z.right;
        y.right.parent ← y;
    end
    Transplant(T, z, y);
    y.left ← z.left;
    y.left.parent ← y;
end
```

Tree Delete: Size



Update Size




```
function UpdateSize(T, z)
while (z ≠ NIL) do
    z.size ← 1;
    if (z.left ≠ NIL) then  z.size ← z.size + z.left.size;
    if (z.right ≠ NIL) then  z.size ← z.size + z.right.size;
    z ← z.parent;
end
```

Tree Delete

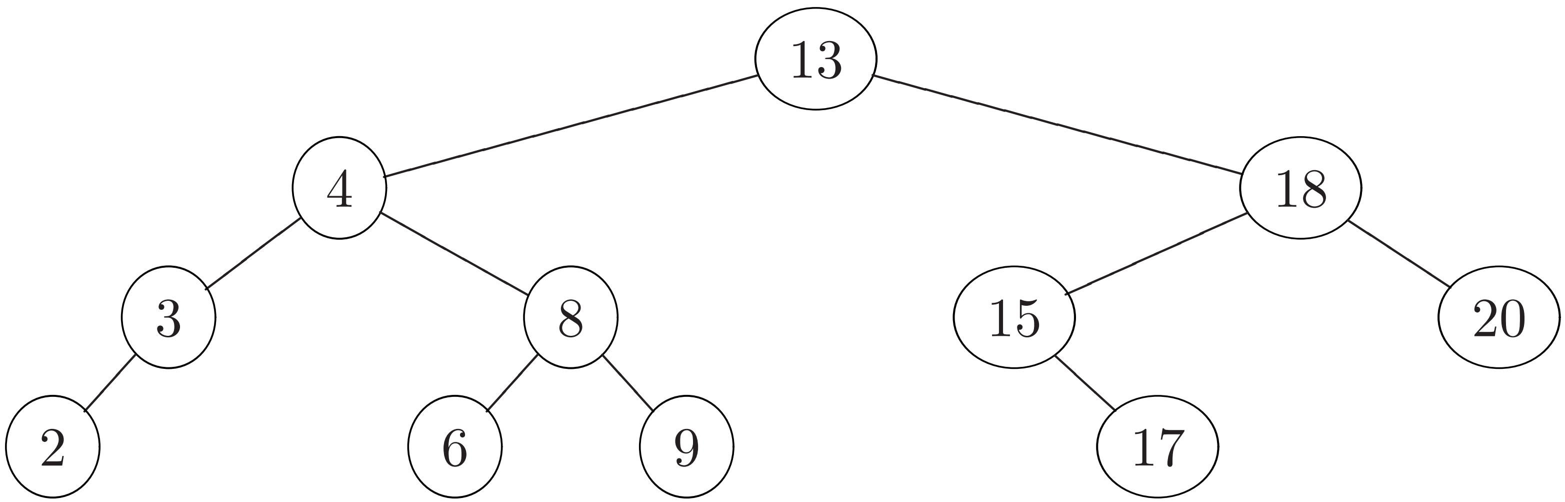
```
function TreeDeleteB(T, z)
if (z.left = NIL) then
    Transplant(T,z,z.right);
    UpdateSize (T, z.parent);
else if (z.right = NIL) then
    Transplant(T,z,z.left);
    UpdateSize (T, z.parent);
else
    ...
```

Tree Delete (continued)

```
function TreeDeleteB(T, z)
...
else
    y ← TreeMin(z.right) ;          /* y is the successor of z */
    x ← y.parent;
    if (y ≠ z.right) then
        Transplant(T, y, y.right);
        y.right ← z.right;
        y.right.parent ← y;
    end
    Transplant(T, z, y);
    y.left ← z.left;
    y.left.parent ← y;
    UpdateSize (T, x);
end
```

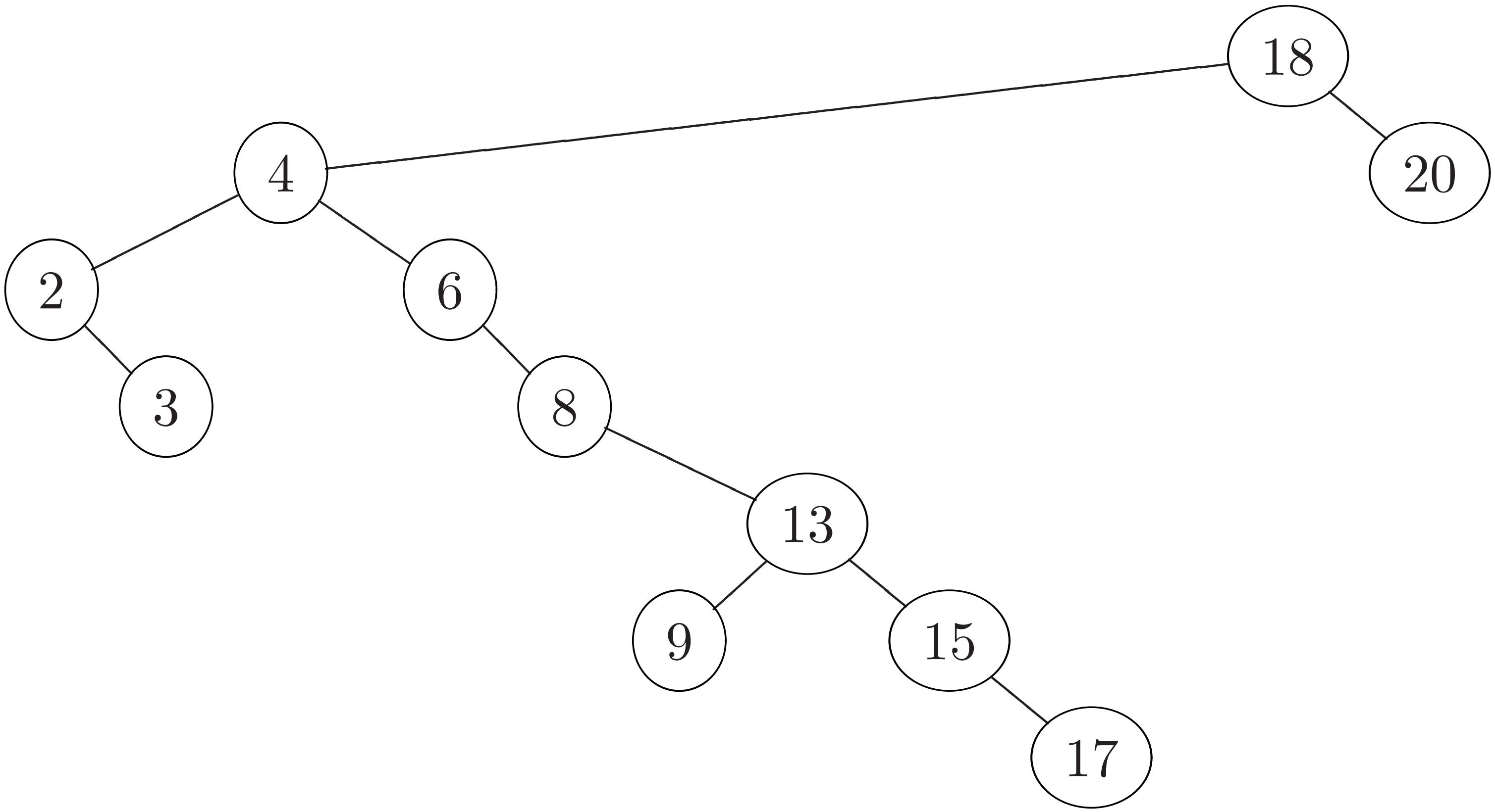
Balanced Search Trees

"Balanced" Binary Search Tree



Binary search tree on: (2, 3, 4, 6, 8, 9, 13, 15, 17, 18, 20)

"Unbalanced" Binary Search Tree



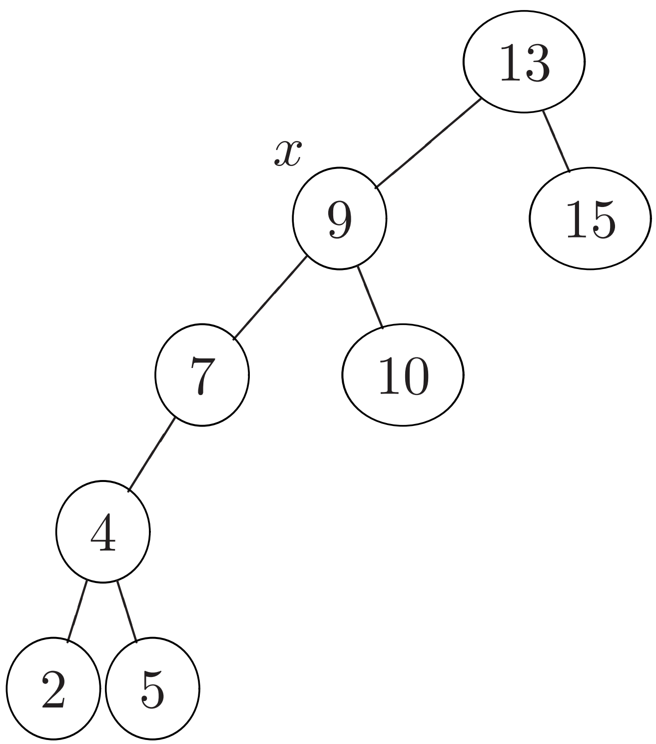
Binary search tree on: (2, 3, 4, 6, 8, 9, 13, 15, 17, 18, 20)

Balanced Binary Search Trees

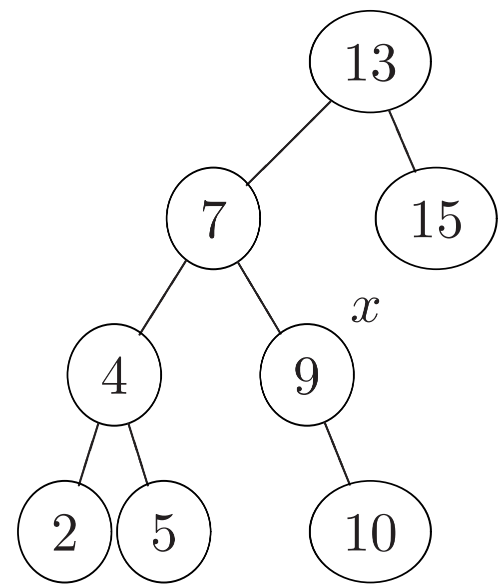
Balanced Binary Search Trees:

- AVL trees (height balanced trees);
- 2-3 Trees;
- Red-black trees;
- Splay trees (self-adjusting).

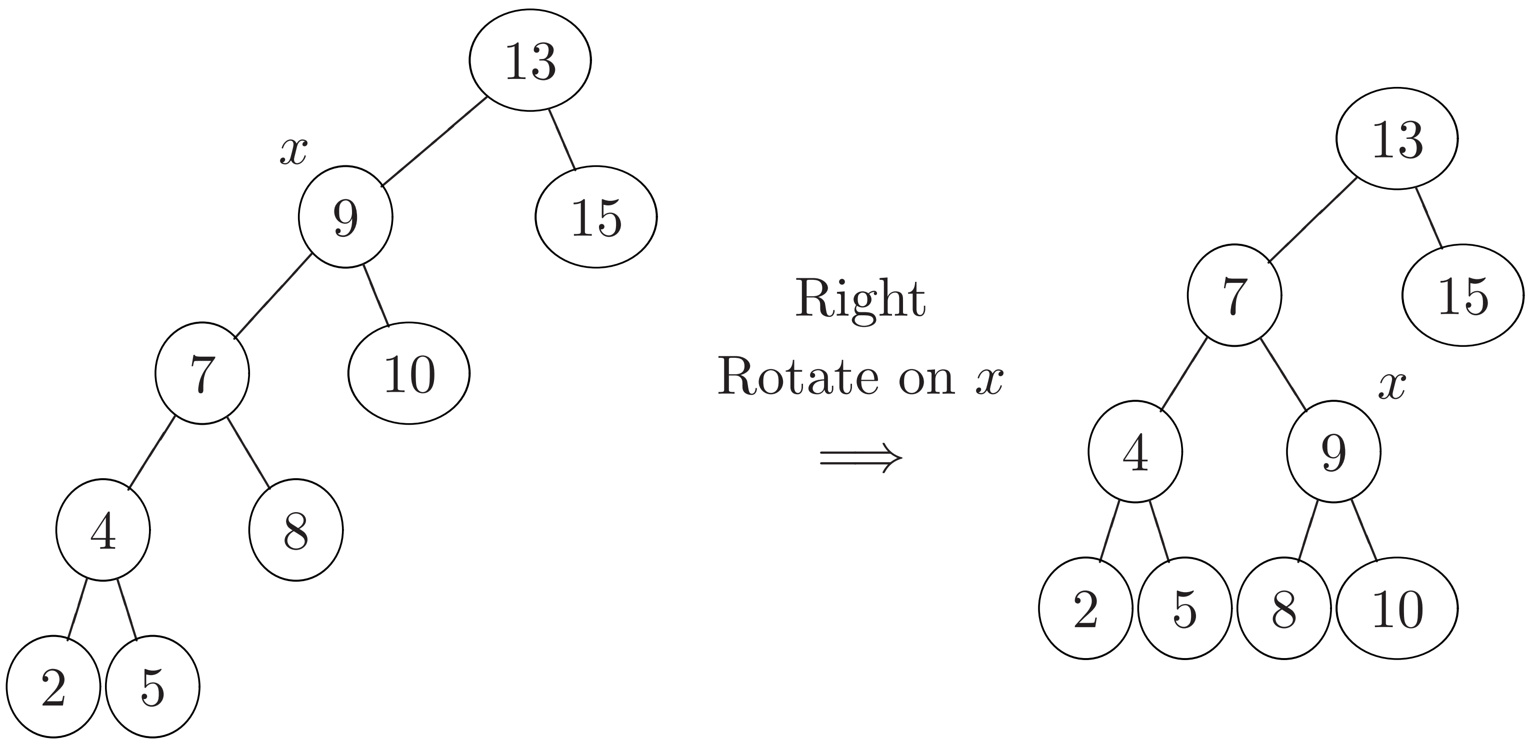
Right Rotate



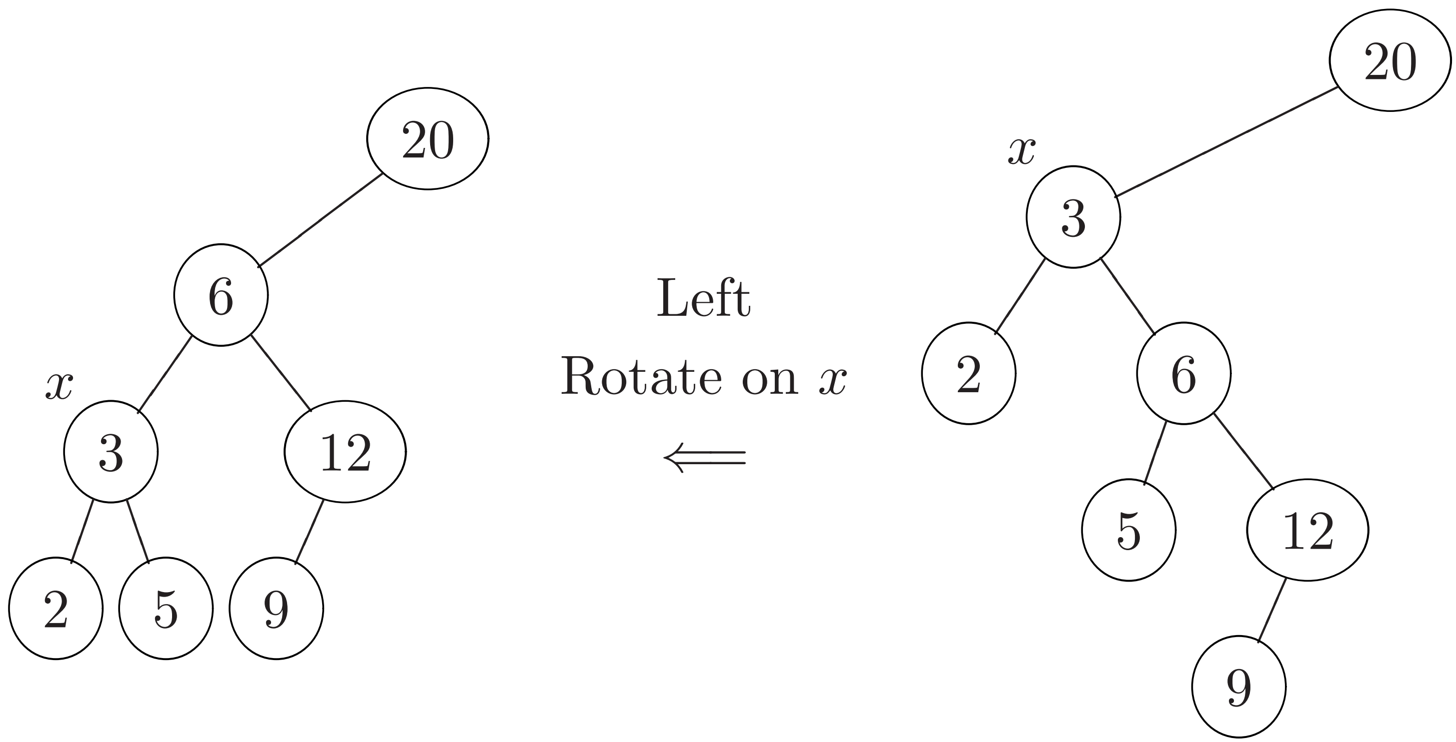
Right
Rotate on x
$\Longrightarrow$



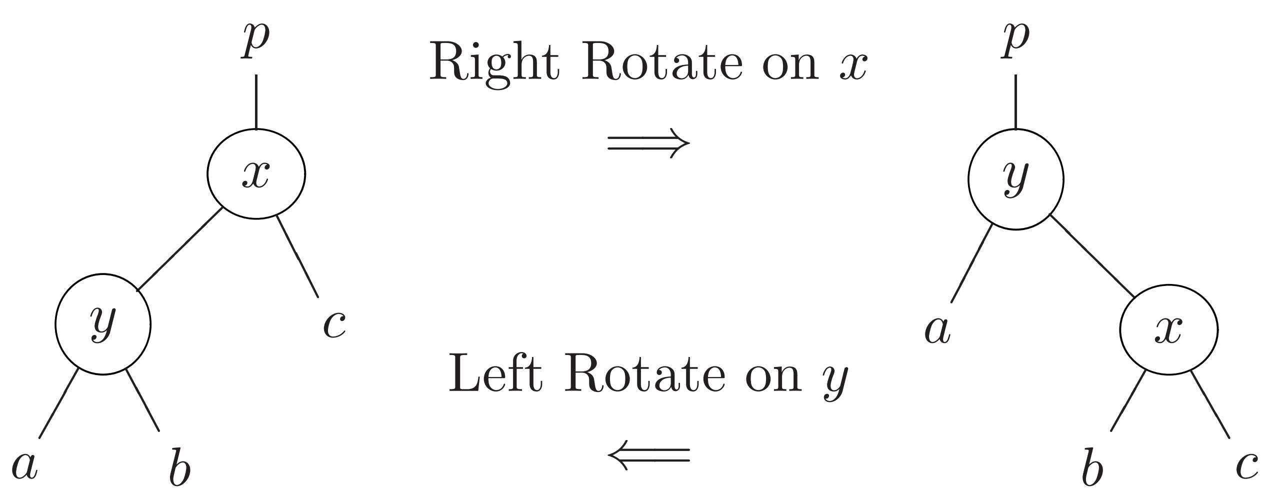
Right Rotate: Example II



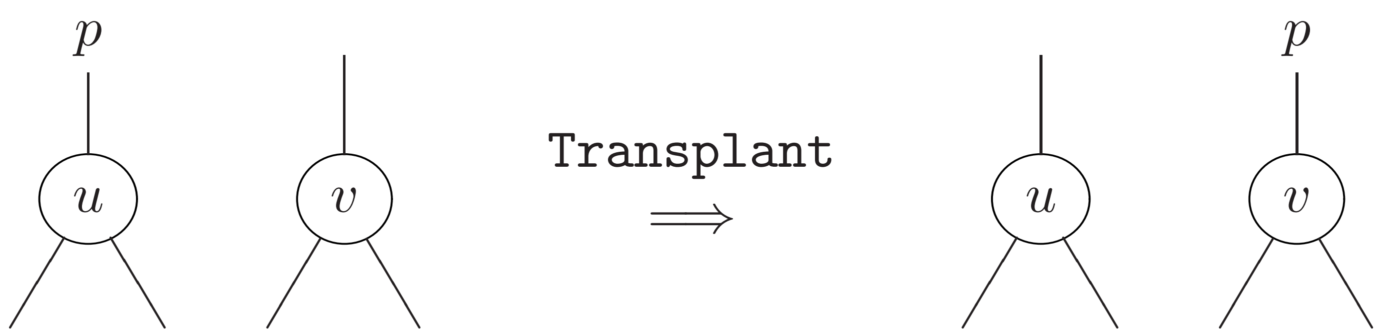
Left Rotate



Rotations



Transplant



function `Transplant`(T, u, v)

/ Replace subtree rooted at u with subtree rooted at v. */*

$p \leftarrow u$.parent;

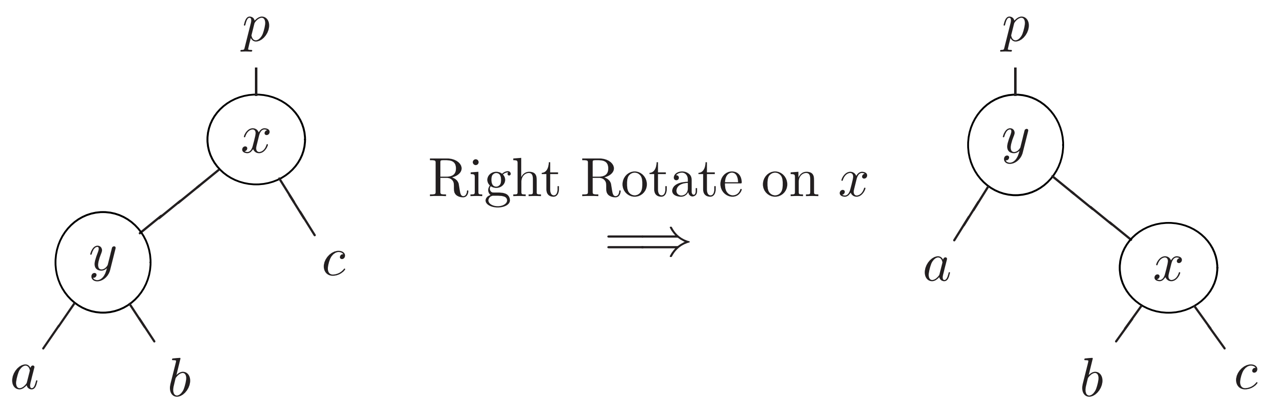
if $(p = \textbf{NIL})$ **then** T.root $\leftarrow v$;

else if $(u = p$.left$)$ **then** p.left $\leftarrow v$;

else p.right $\leftarrow v$;

if $(v \neq \textbf{NIL})$ **then** v.parent $\leftarrow p$;

Right Rotate



function `RightRotate`(T, x)

$y \leftarrow x$.left;

$b \leftarrow y$.right;

`Transplant`(T,x,y);

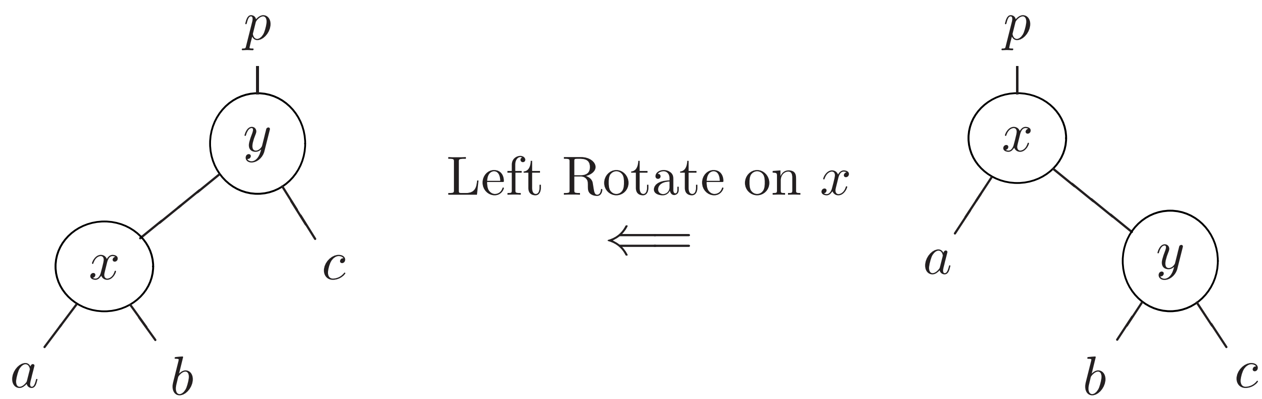
x.left $\leftarrow b$;

if ($b \neq$ **NIL**) **then** b.parent $\leftarrow x$;

y.right $\leftarrow x$;

x.parent $\leftarrow y$;

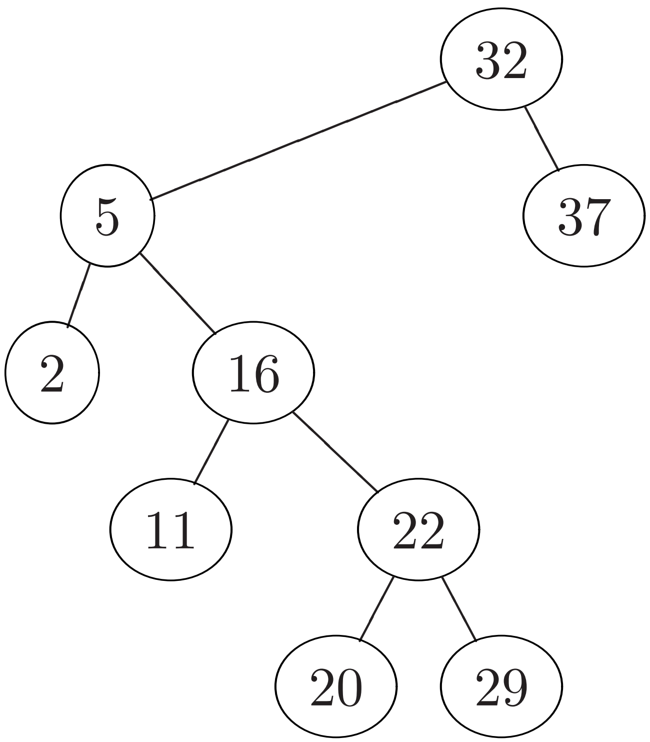
Left Rotate




```
function LeftRotate(T, x)
y ← x.right;
b ← y.left;
Transplant(T,x,y);
x.right ← b;
if (b ≠ NIL) then b.parent ← x;
y.left ← x;
x.parent ← y;
```

Rotate Example

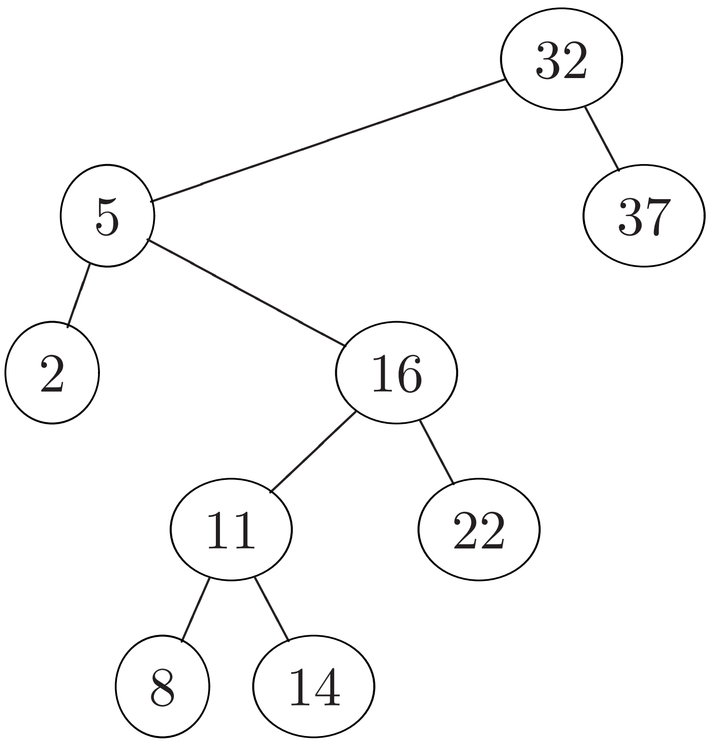
Apply rotation to reduce height of tree:



$\Longrightarrow$

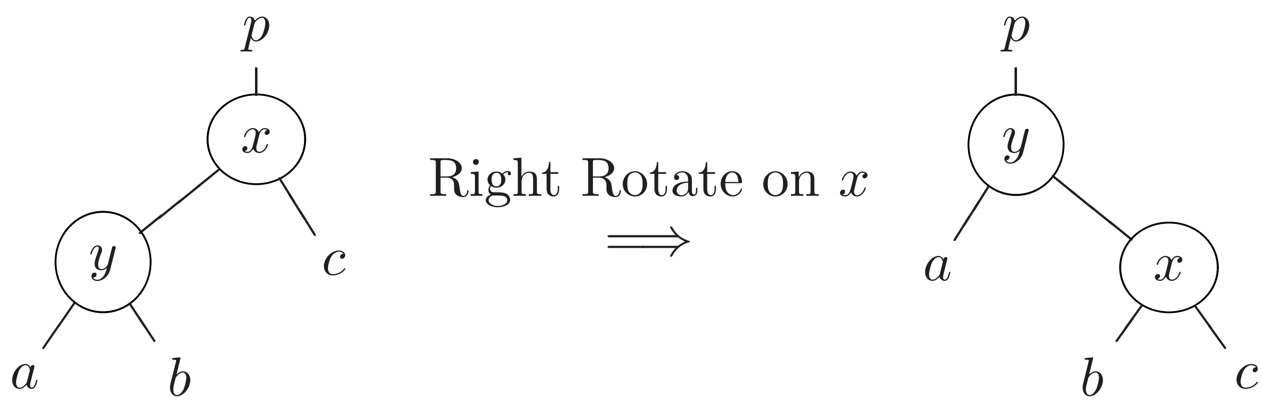
Rotate Example

Apply rotation to reduce height of tree:



$\Longrightarrow$

Right Rotate: Size




```
function RightRotate(T, x)
y ← x.left;
b ← y.right;
Transplant(T,x,y);
x.left ← b;
if (b ≠ NIL) then b.parent ← x;
y.right ← x;
x.parent ← y;
```