CSCI 3230 Data Structures

Search Trees

Weitian Tong, Ph.D.

Department of Computer Science
Georgia Southern University
Website: www.weitiantong.com
Email:wtong@georgiasouthern.edu

Table of contents

Binary Search Tree

Binary search tree

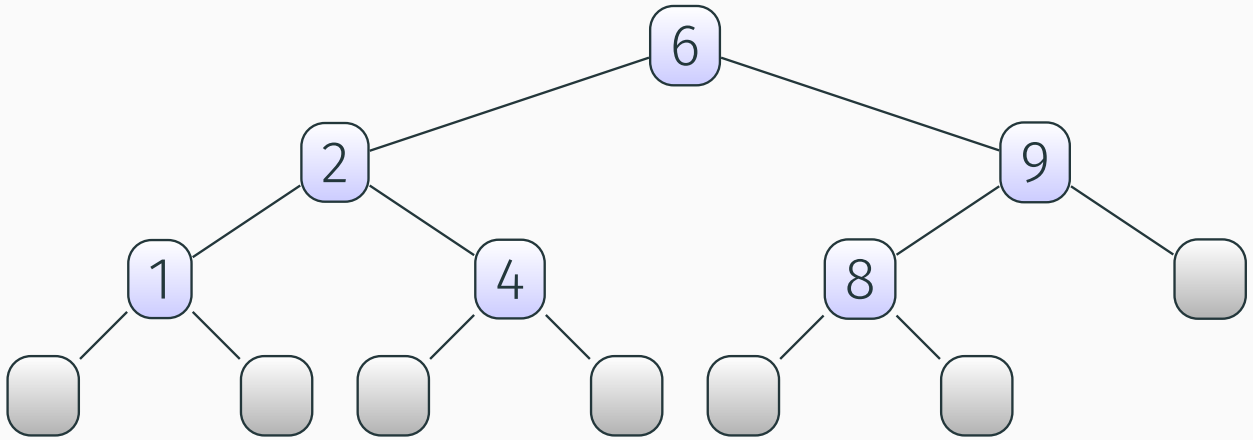
Binary search tree

A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

- Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have

$$key(u) \leq key(v) \leq key(w)$$

- External nodes do not store items



An inorder traversal visits the keys in increasing order

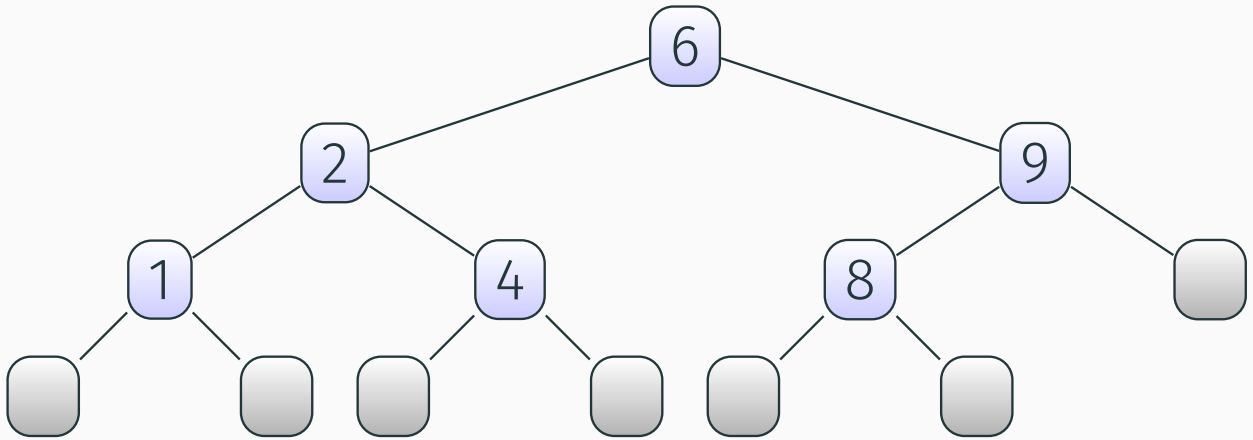
Search

```
1  // TreeSearch(k, v):
2  if T.isExternal (v)
3    return v // not find
4  if k < key(v)
5    return TreeSearch(k, left(v))
6  else if k = key(v)
7    return v
8  else // k > key(v)
9    return TreeSearch(k, right(v))
```

Search

```
1  // TreeSearch(k, v):
2  if T.isExternal (v)
3    return v // not find
4  if k < key(v)
5    return TreeSearch(k, left(v))
6  else if k = key(v)
7    return v
8  else // k > key(v)
9    return TreeSearch(k, right(v))
```

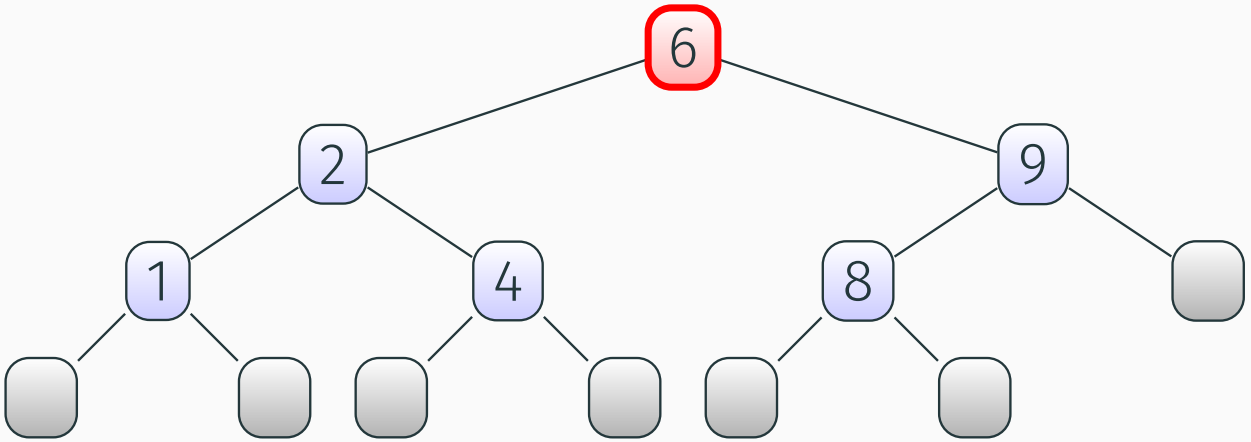
Search for 4



Search

```
1  // TreeSearch(k, v):
2  if T.isExternal (v)
3    return v // not find
4  if k < key(v)
5    return TreeSearch(k, left(v))
6  else if k = key(v)
7    return v
8  else // k > key(v)
9    return TreeSearch(k, right(v))
```

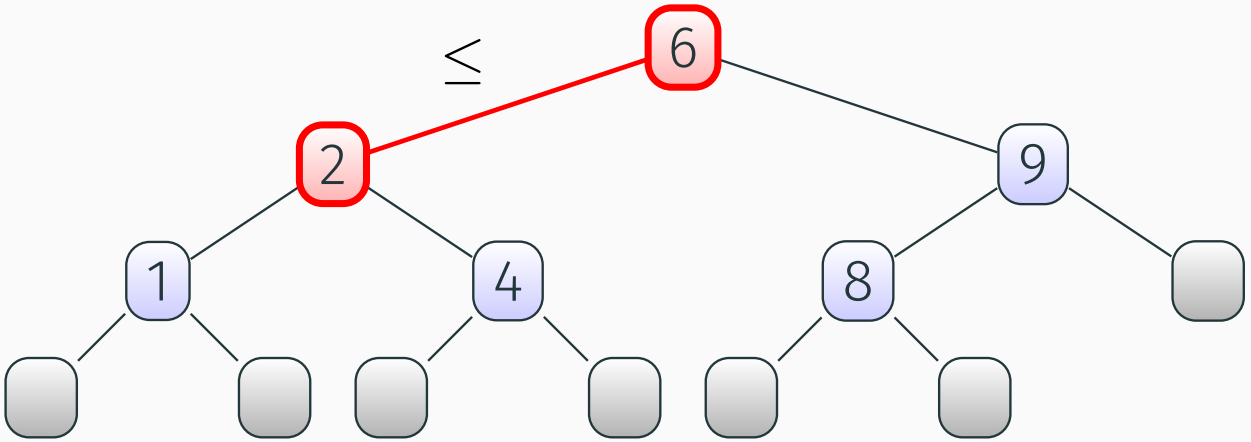
Search for 4



Search

```
1  // TreeSearch(k, v):
2  if T.isExternal (v)
3    return v // not find
4  if k < key(v)
5    return TreeSearch(k, left(v))
6  else if k = key(v)
7    return v
8  else // k > key(v)
9    return TreeSearch(k, right(v))
```

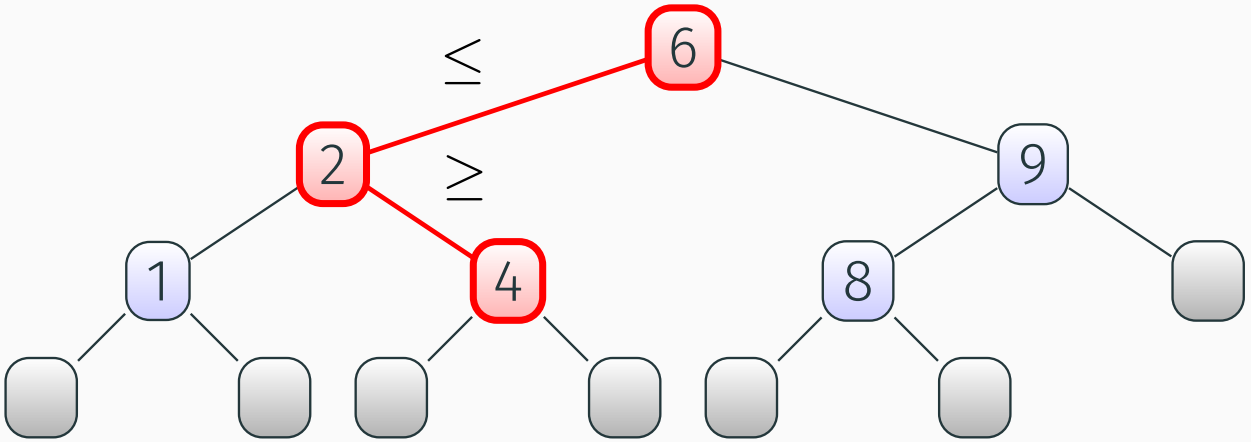
Search for 4



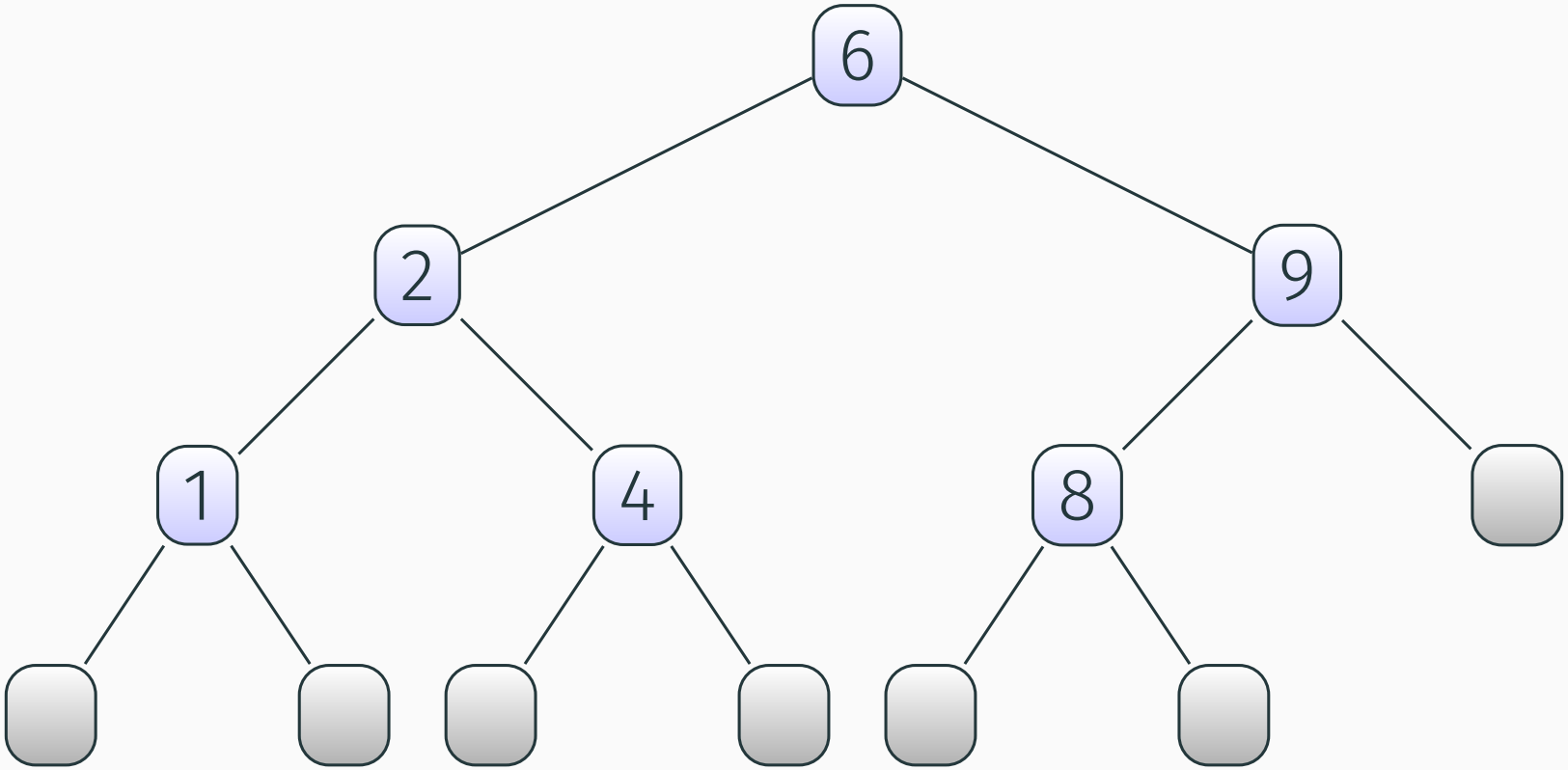
Search

```
1  // TreeSearch(k, v):
2  if T.isExternal (v)
3    return v // not find
4  if k < key(v)
5    return TreeSearch(k, left(v))
6  else if k = key(v)
7    return v
8  else // k > key(v)
9    return TreeSearch(k, right(v))
```

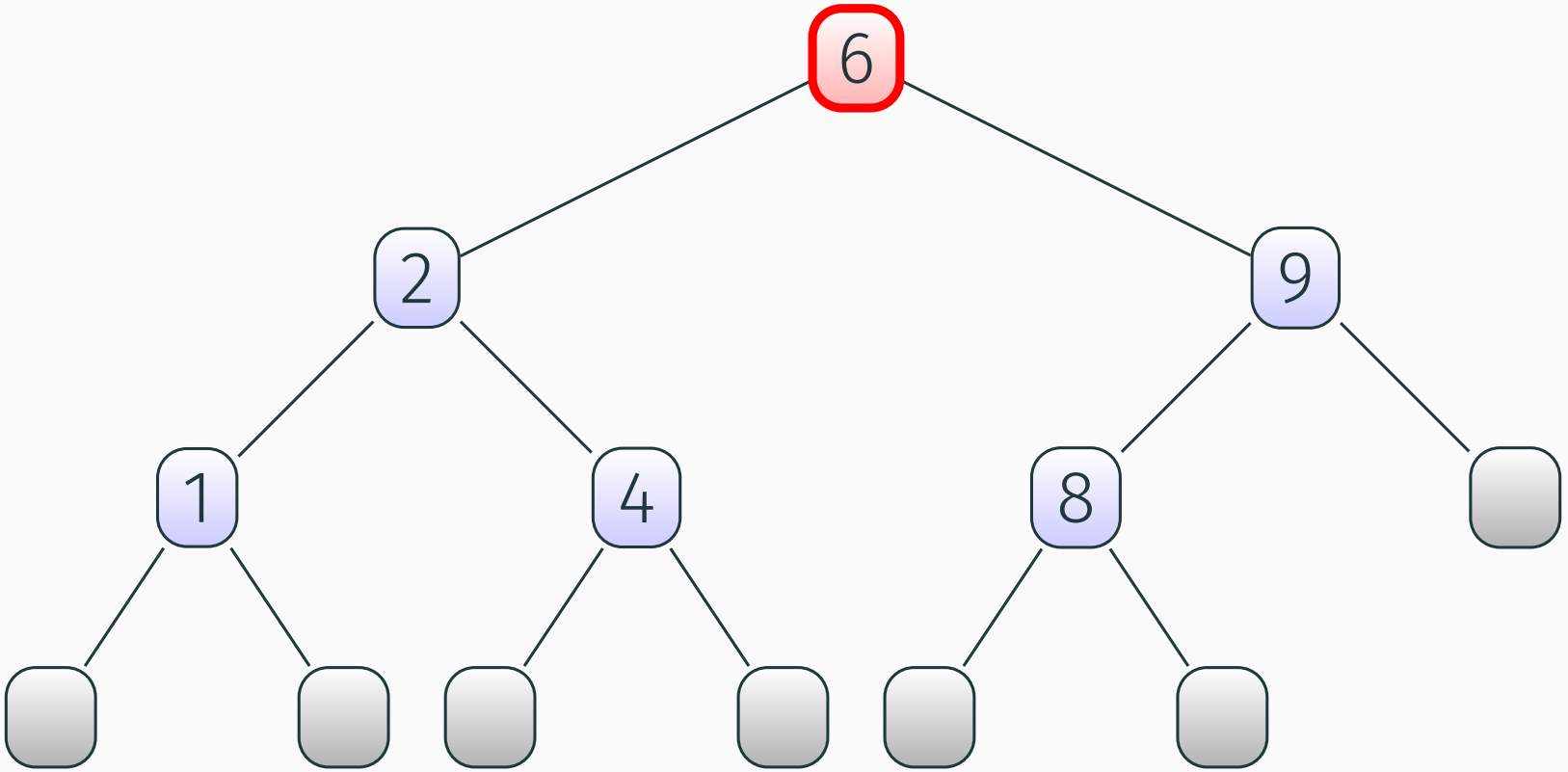
Search for 4



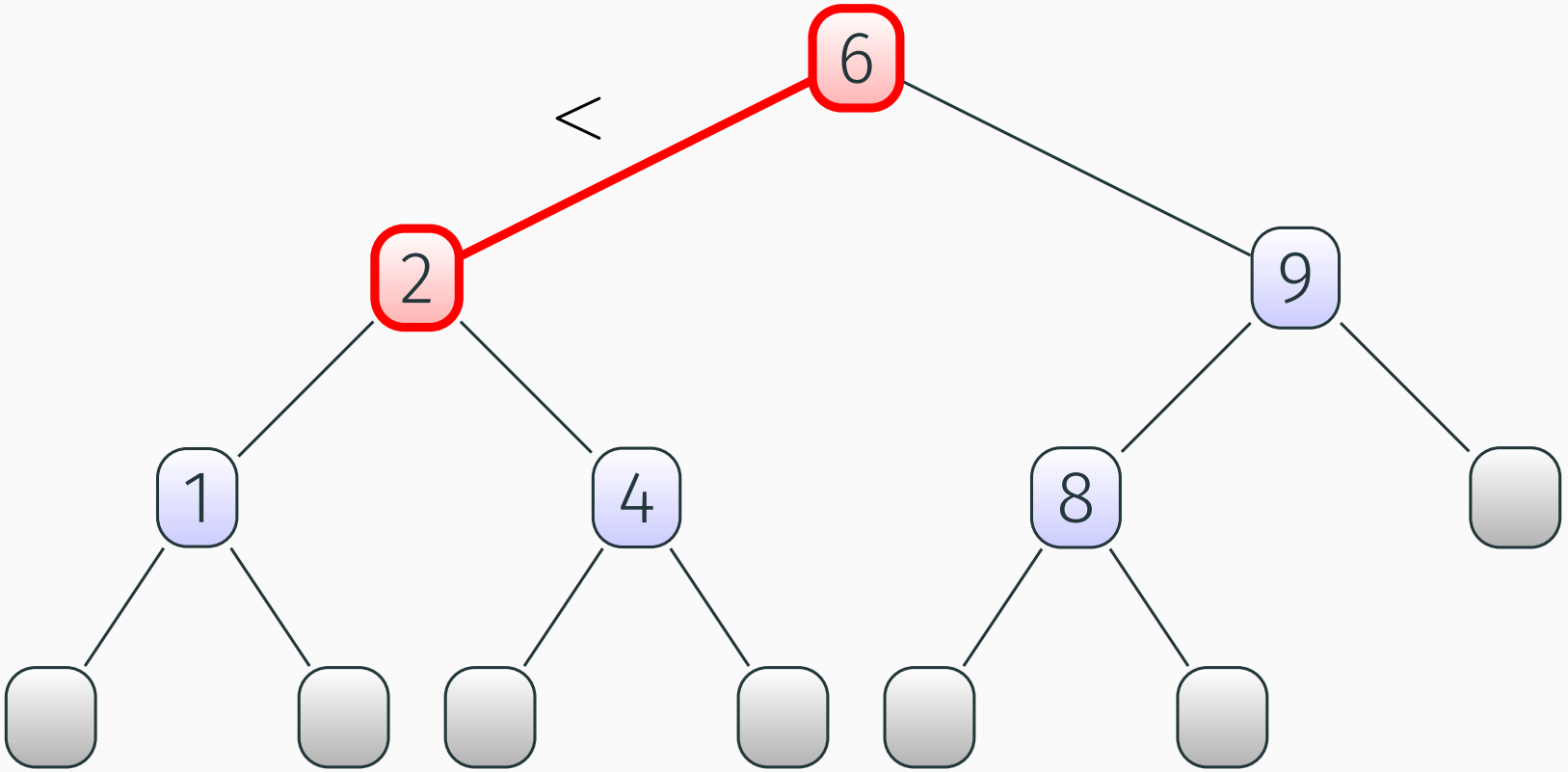
Insertion: insert 5



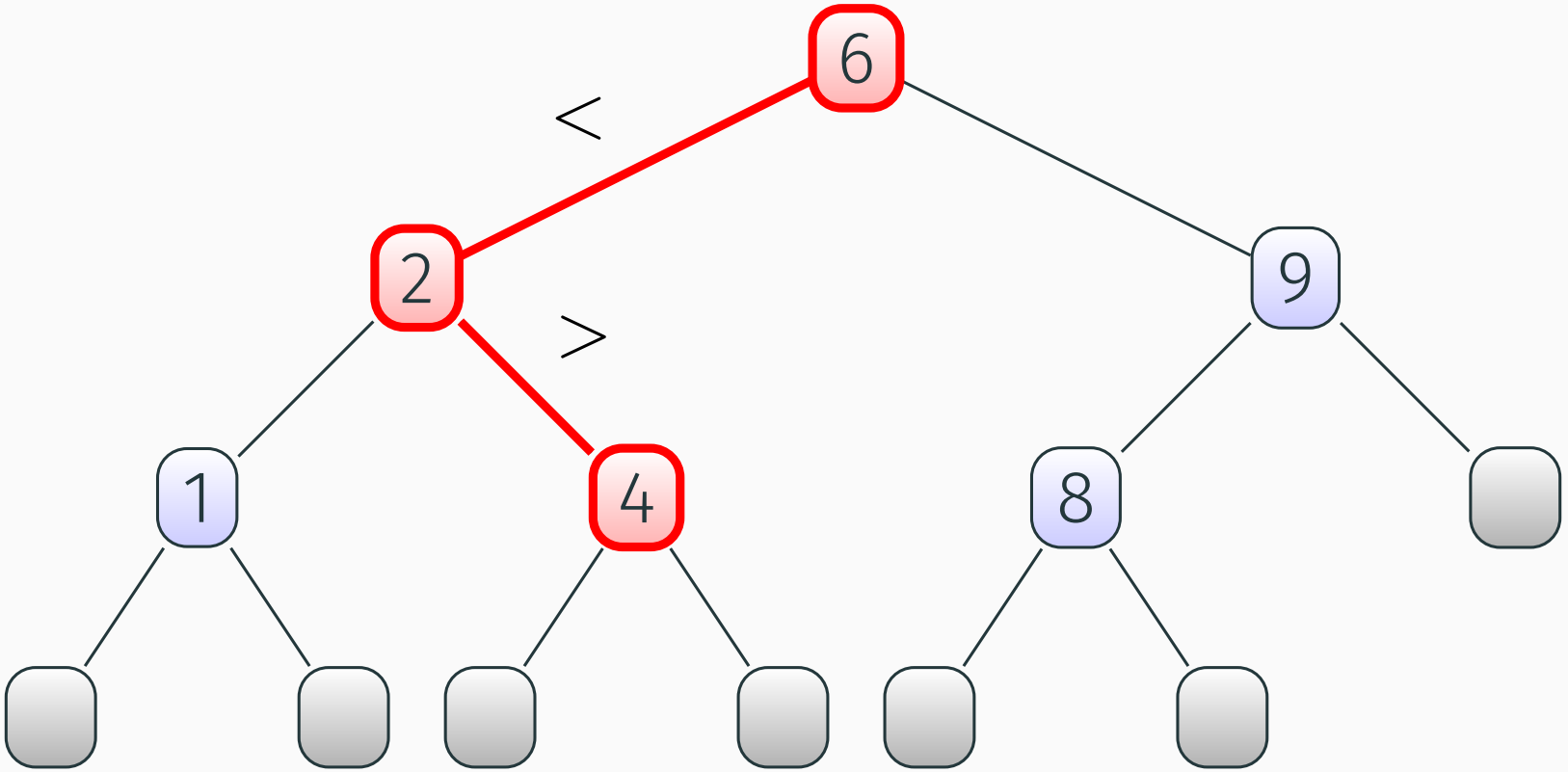
Insertion: insert 5



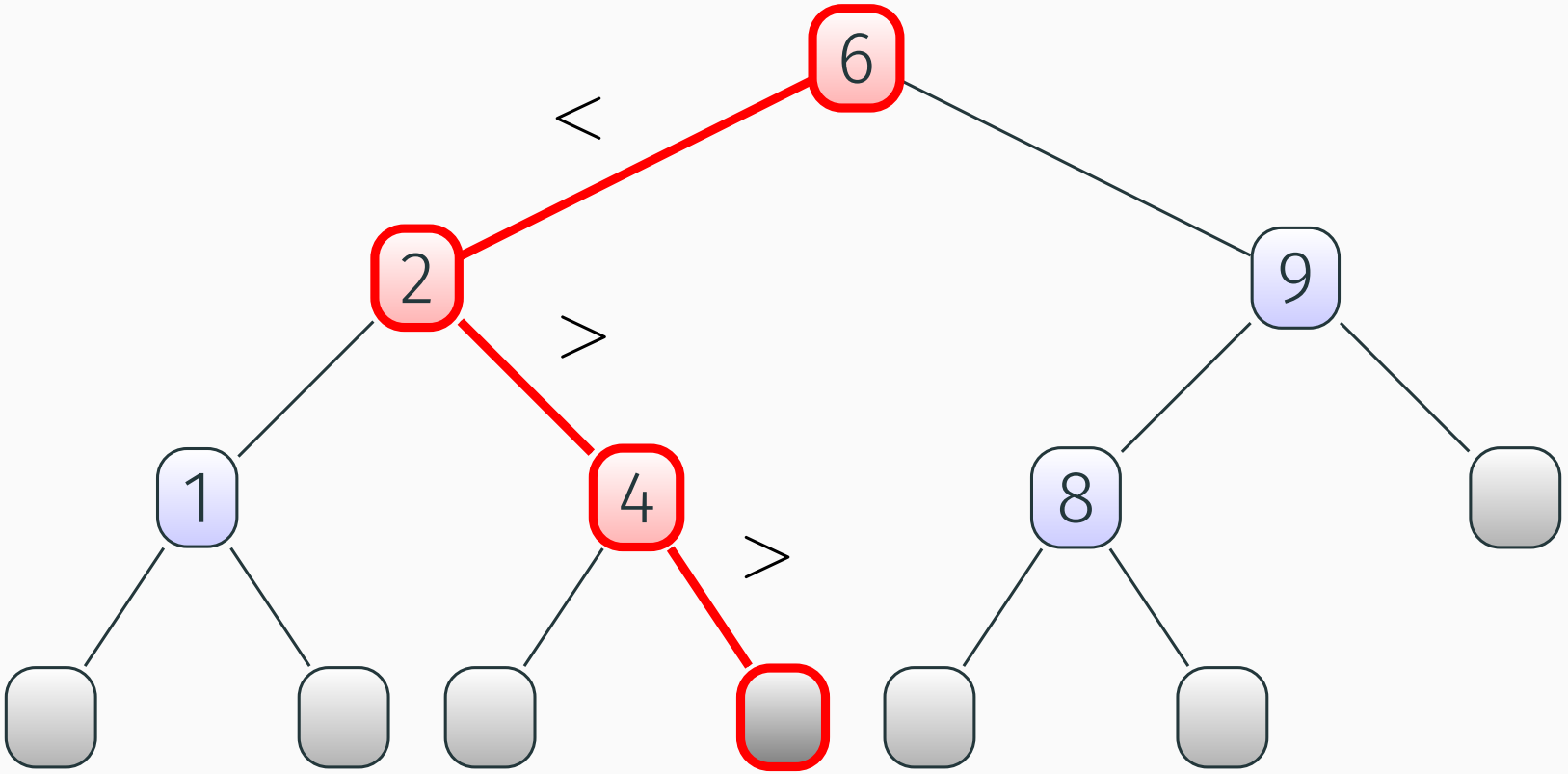
Insertion: insert 5



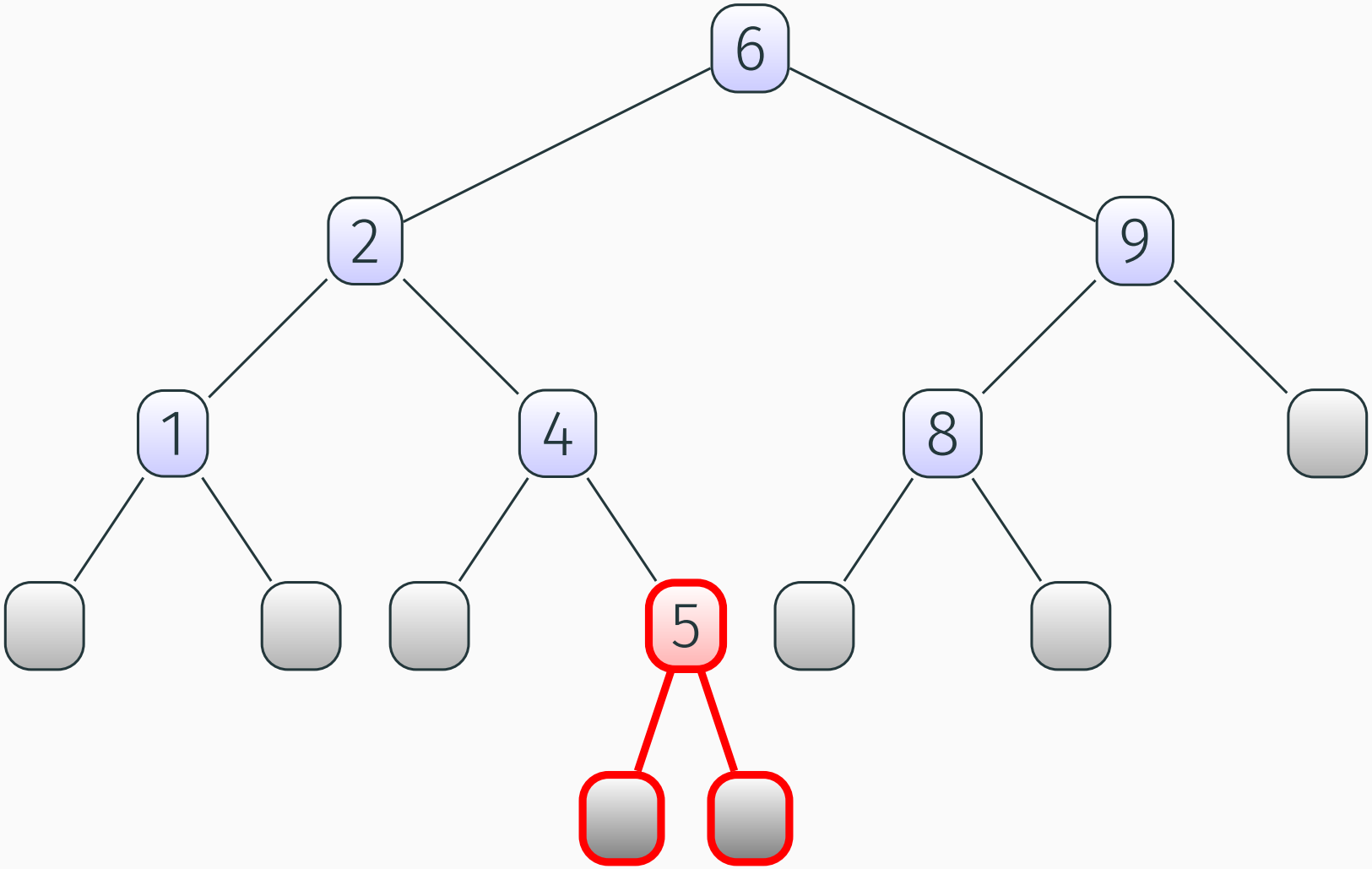
Insertion: insert 5



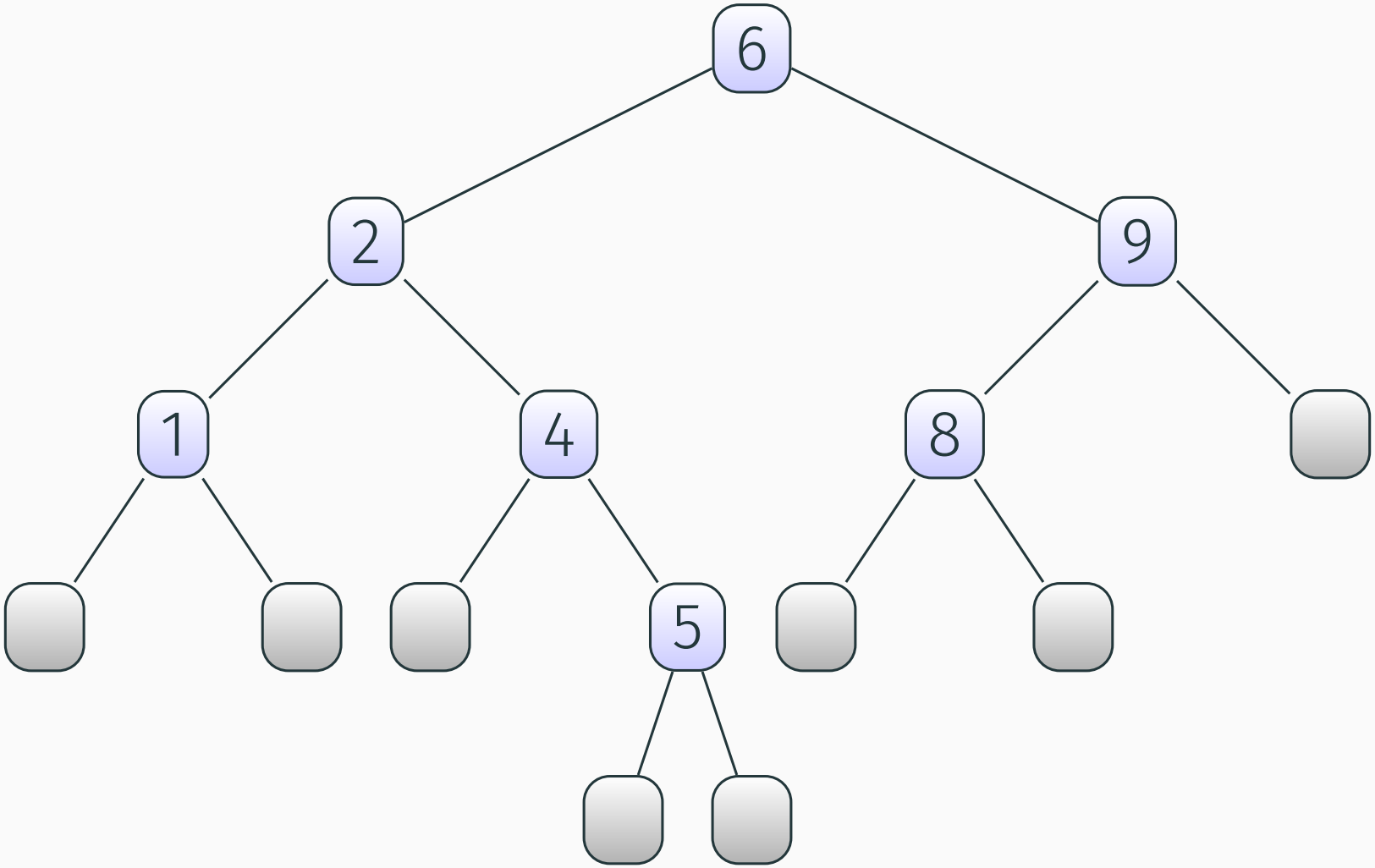
Insertion: insert 5



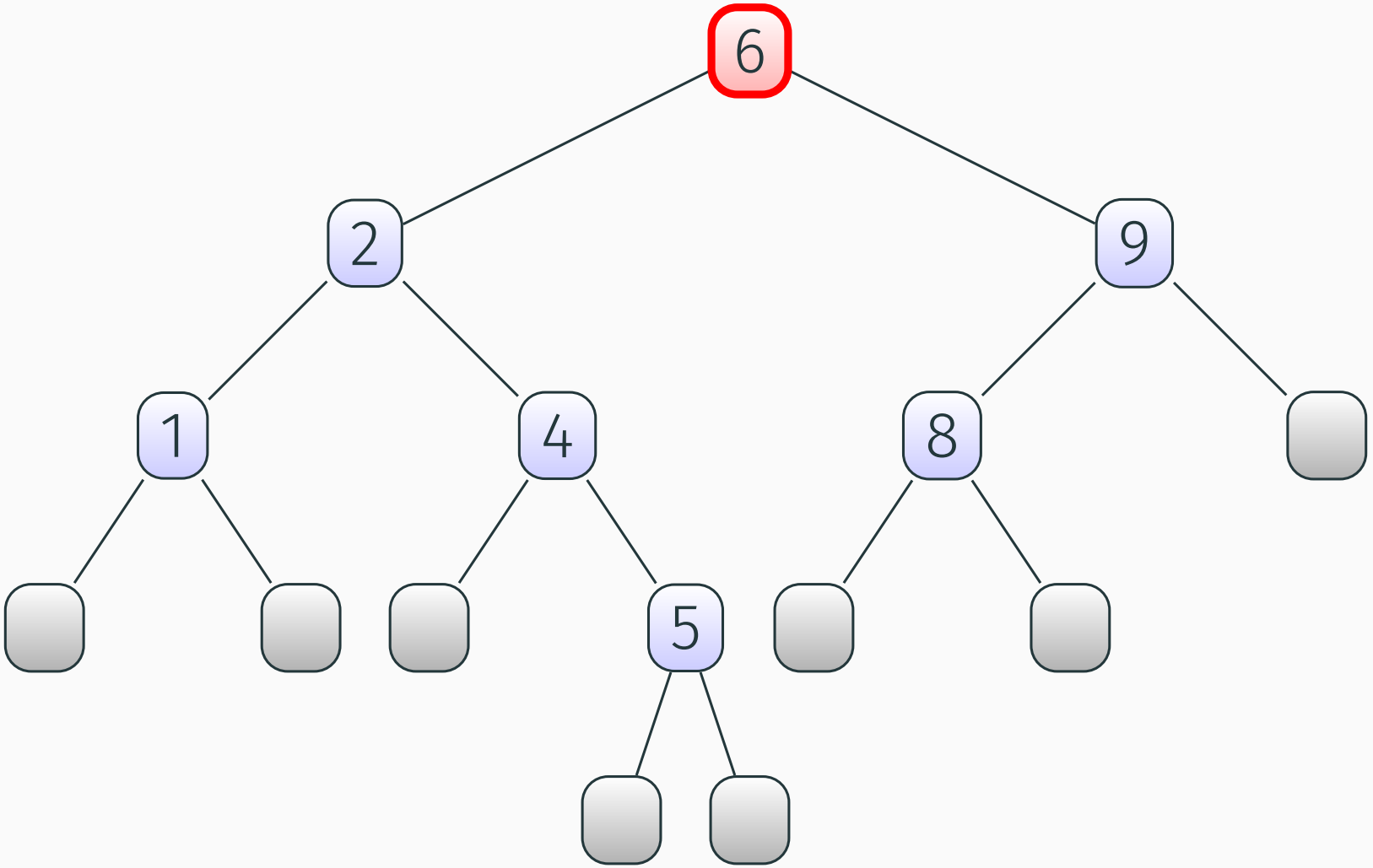
Insertion: insert 5



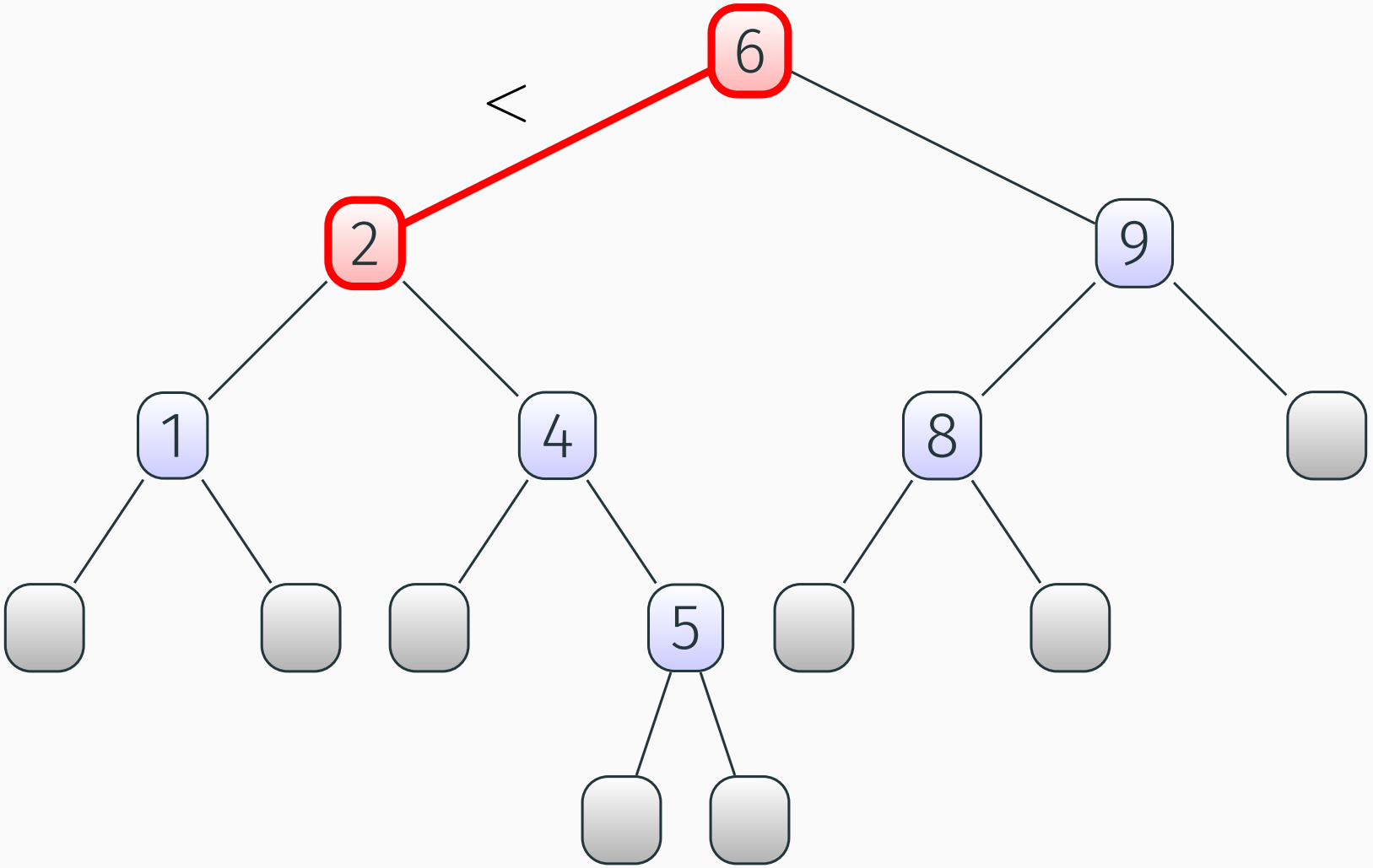
Deletion: delete 4



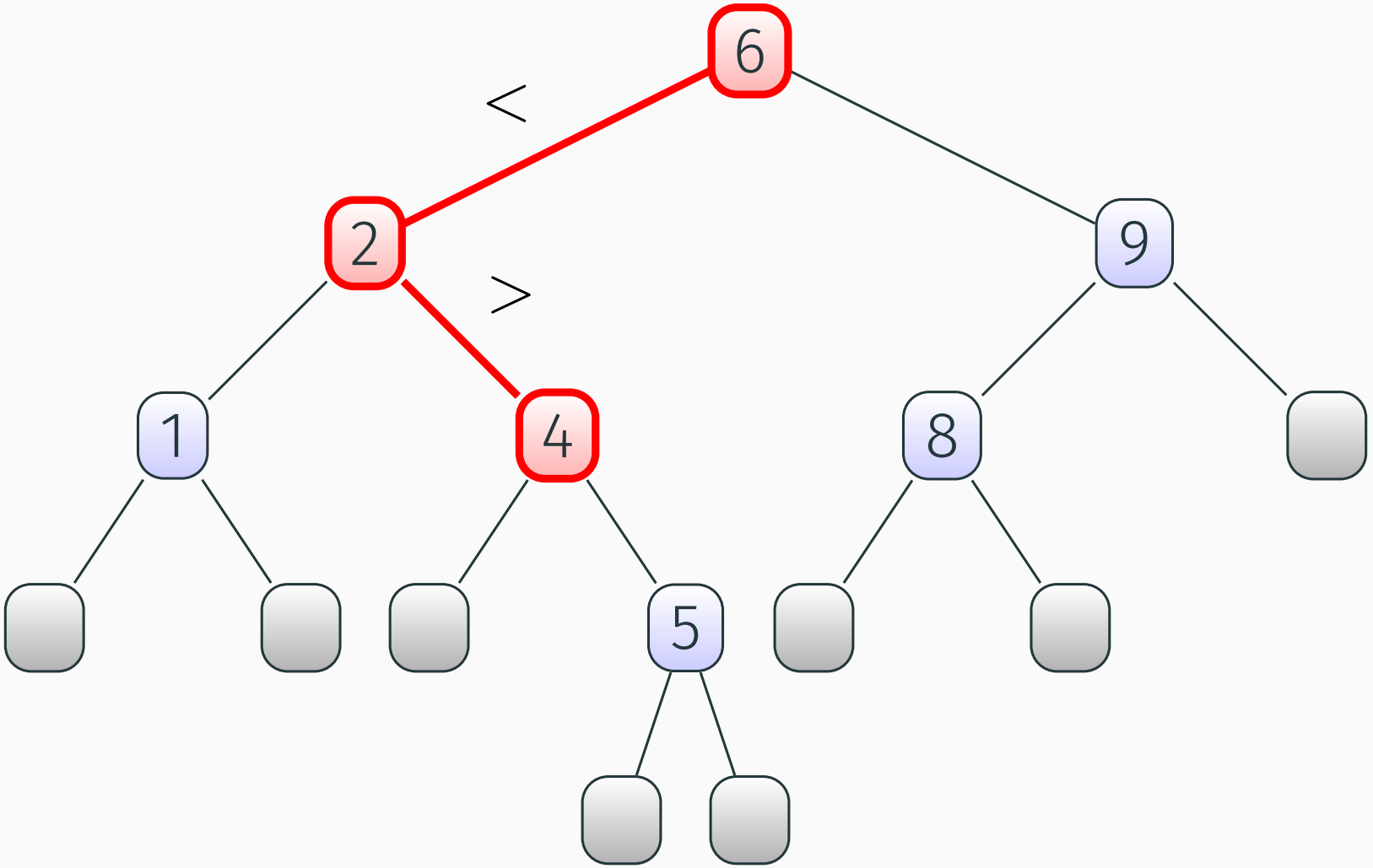
Deletion: delete 4



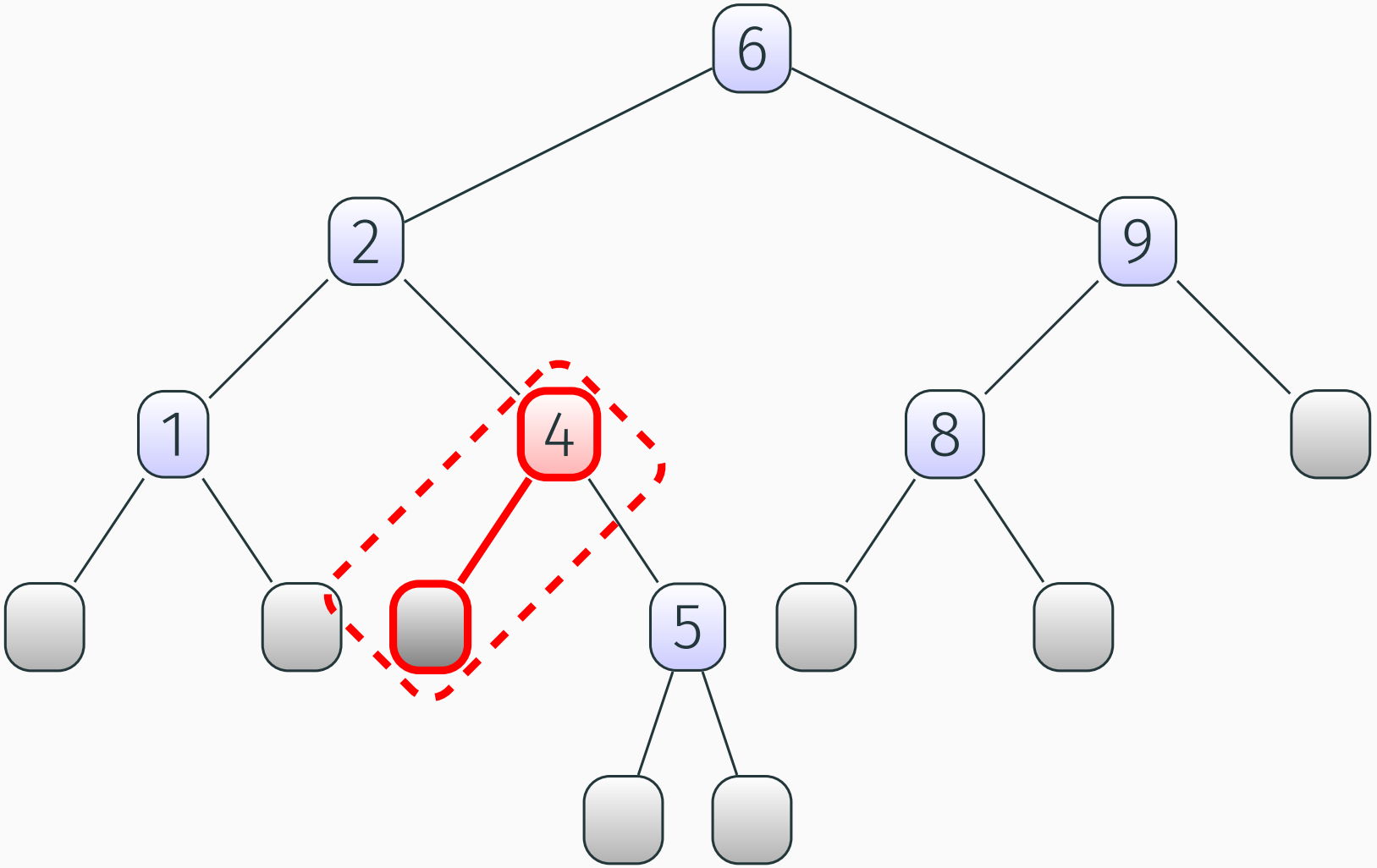
Deletion: delete 4



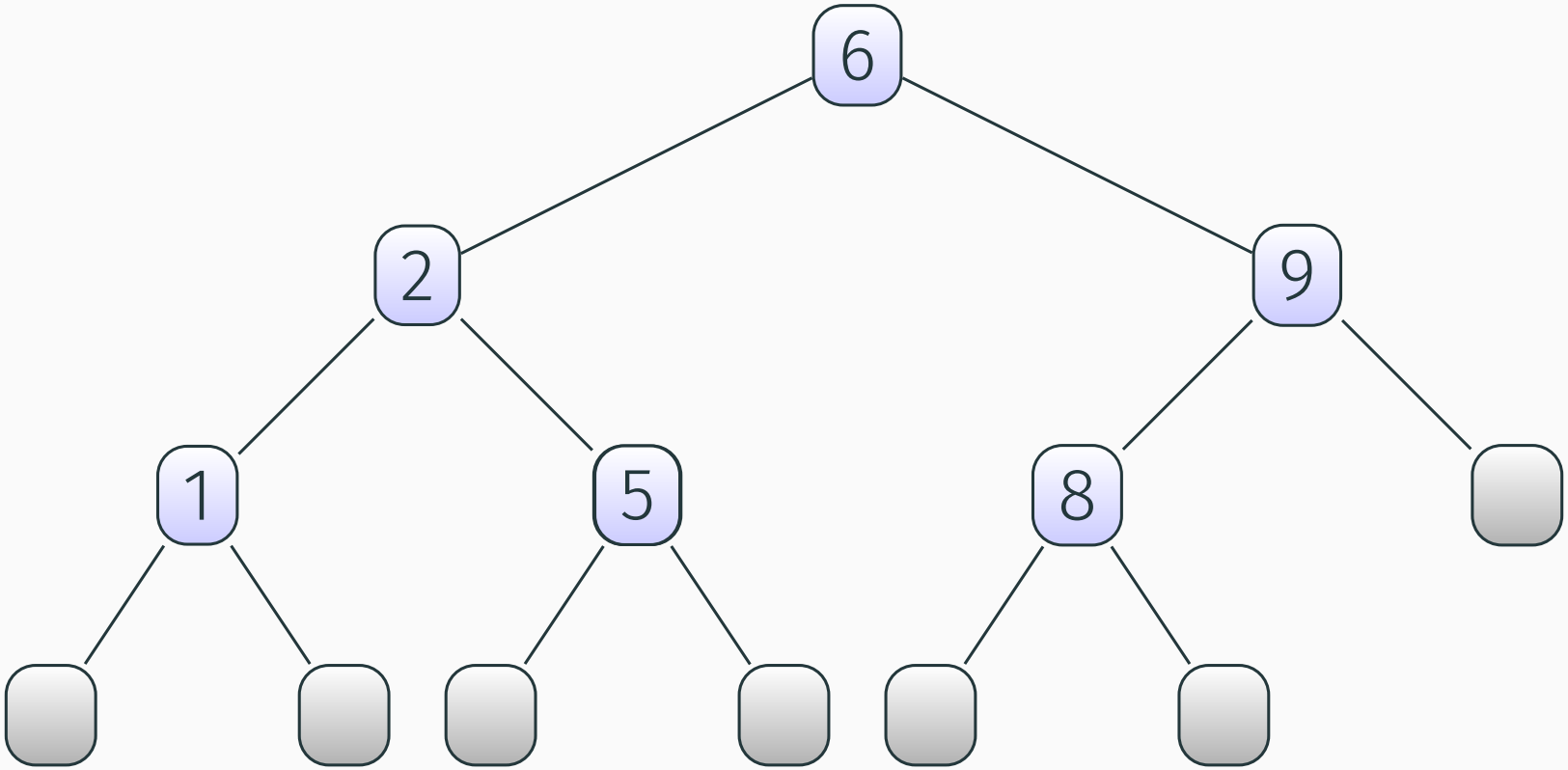
Deletion: delete 4



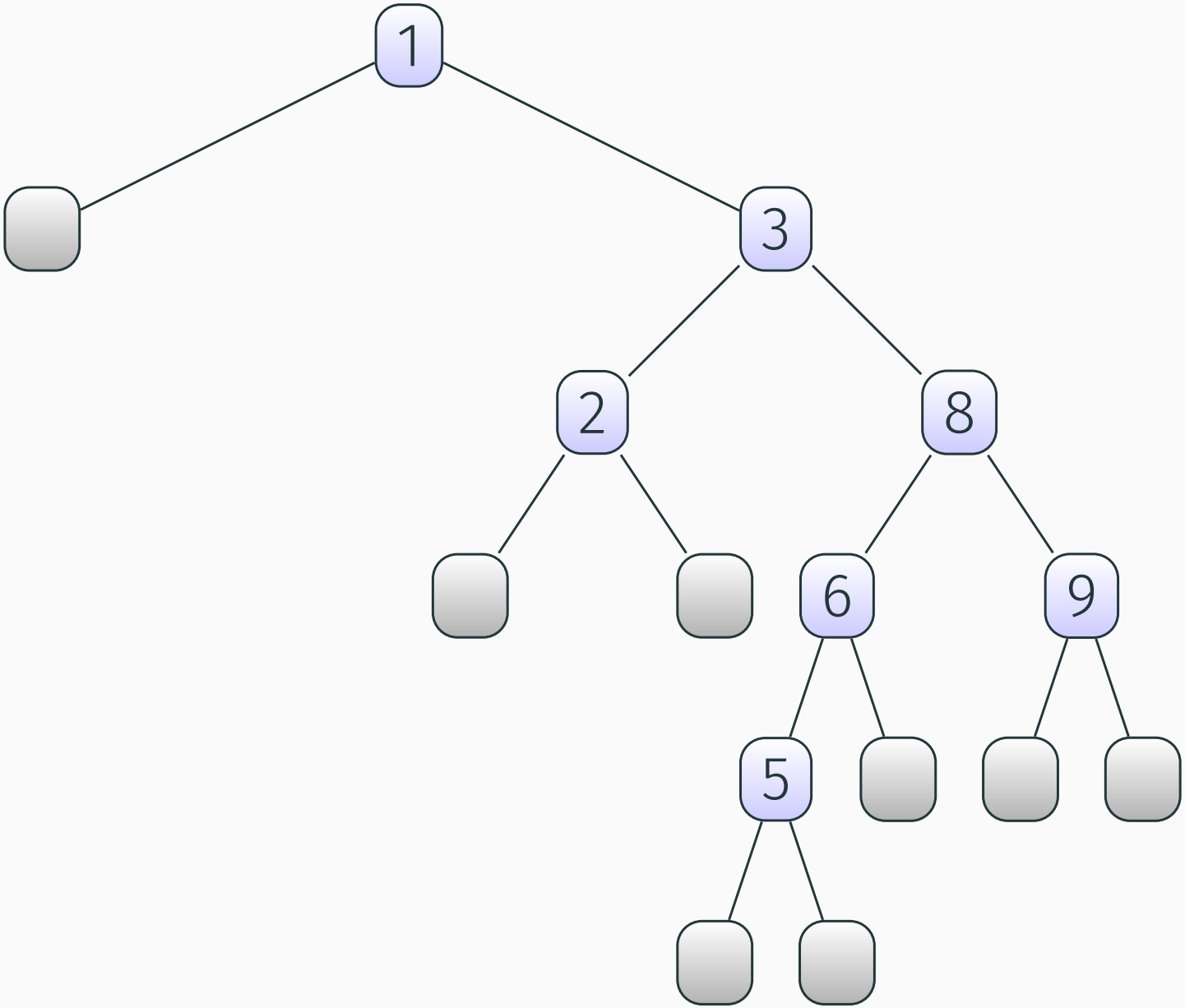
Deletion: delete 4



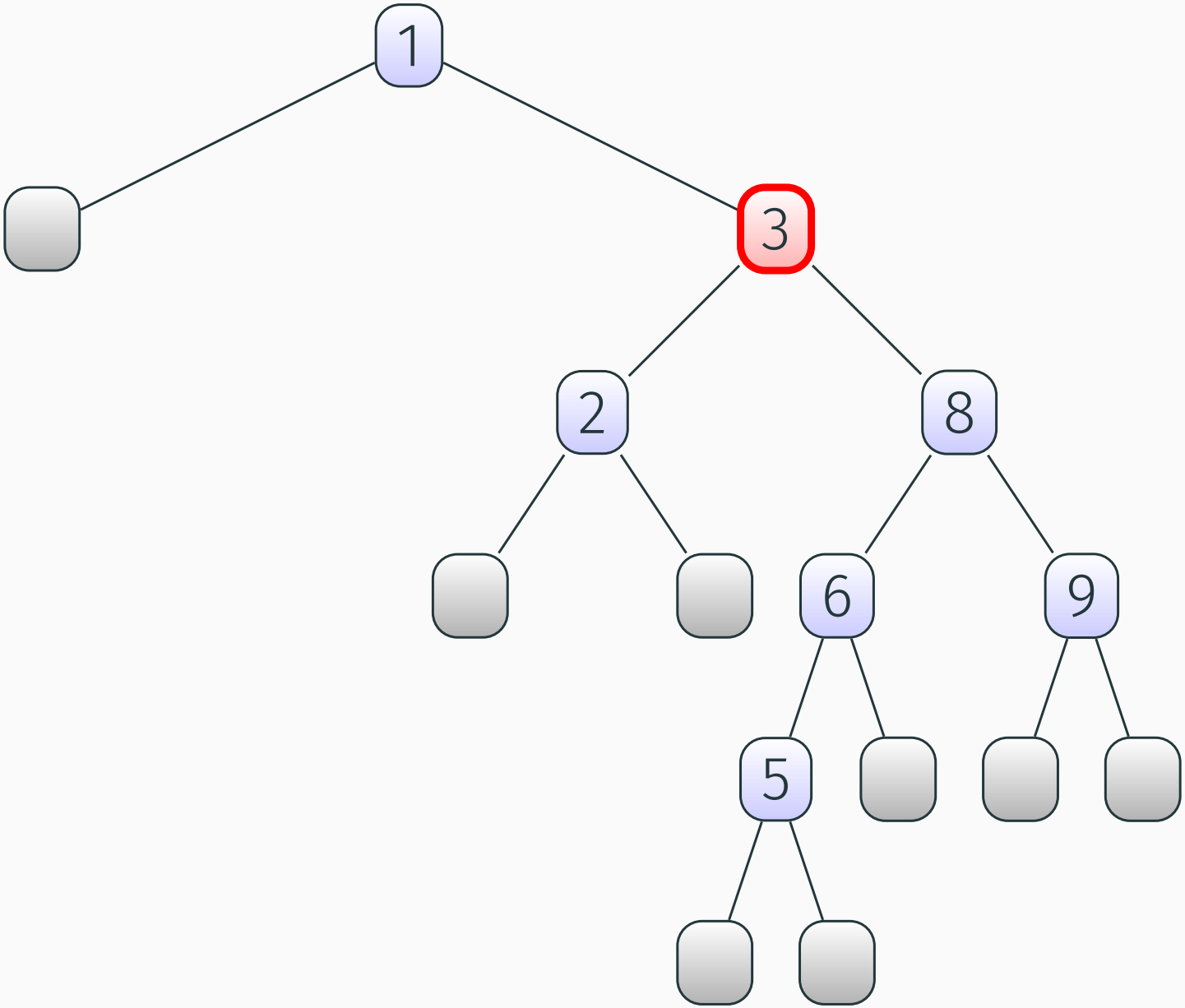
Deletion: delete 4



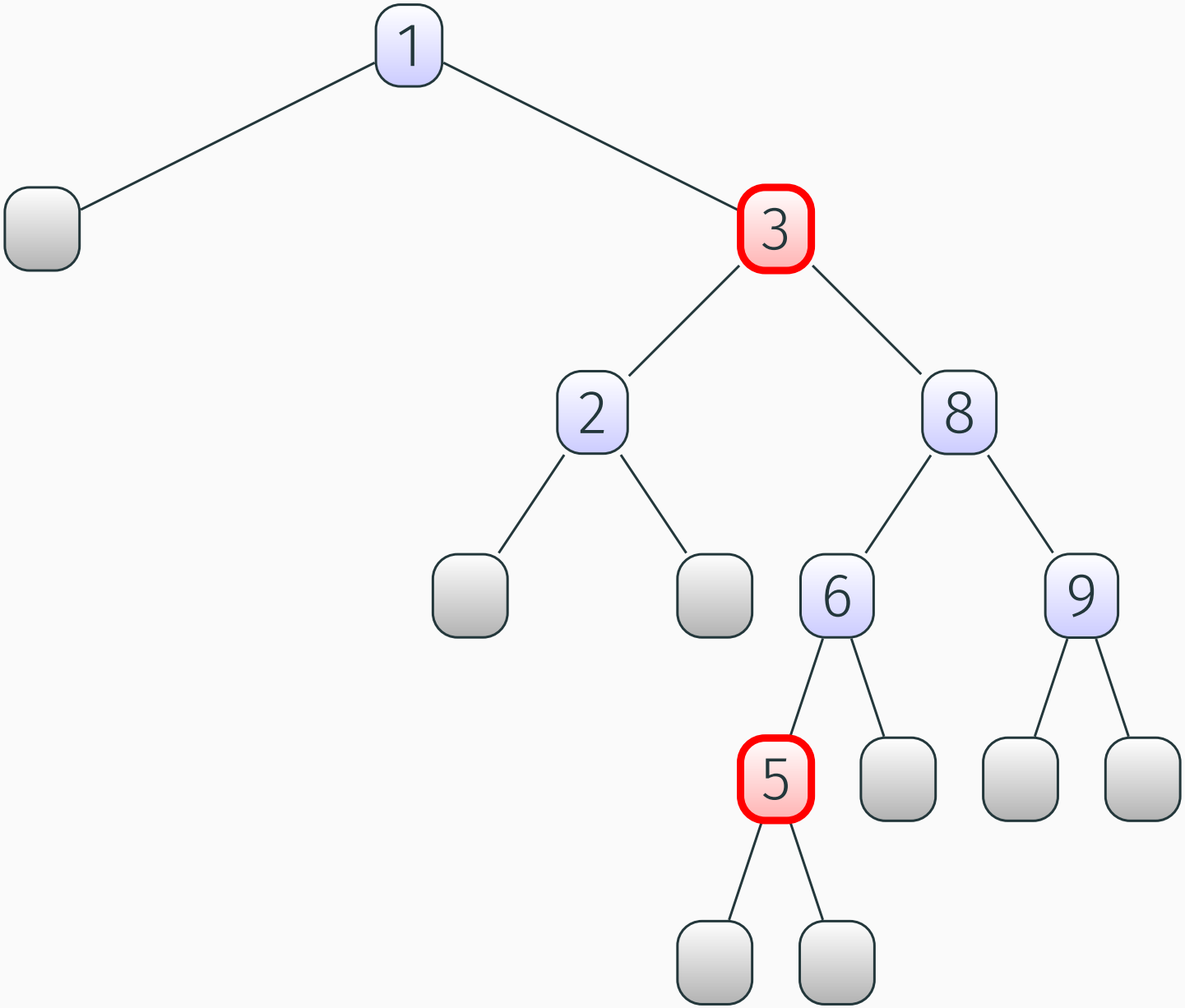
Deletion: (cont.) delete 3



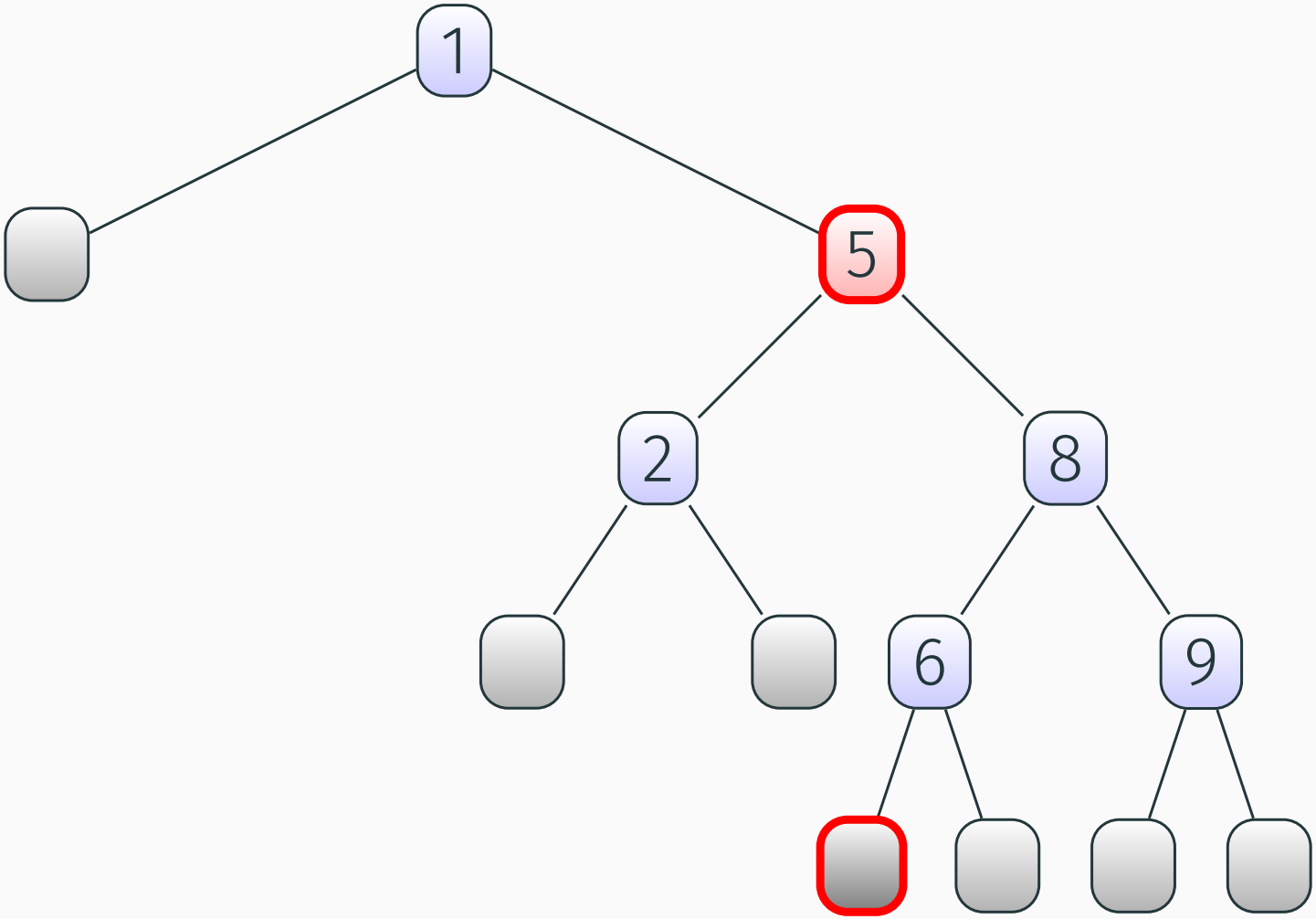
Deletion: (cont.) delete 3



Deletion: (cont.) delete 3



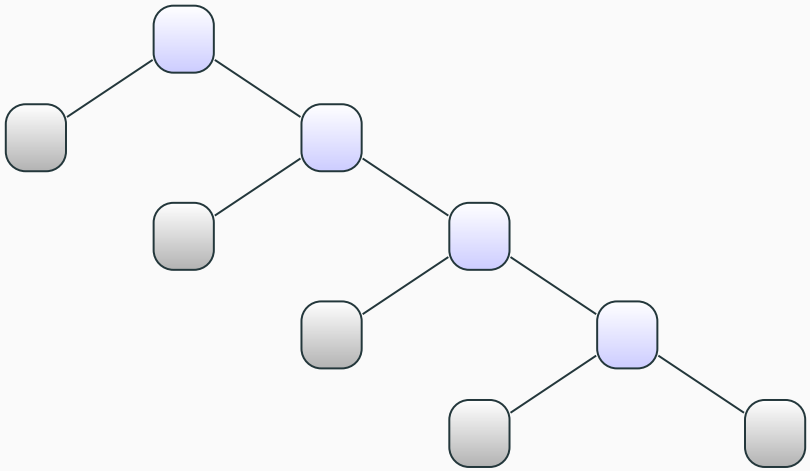
Deletion: (cont.) delete 3

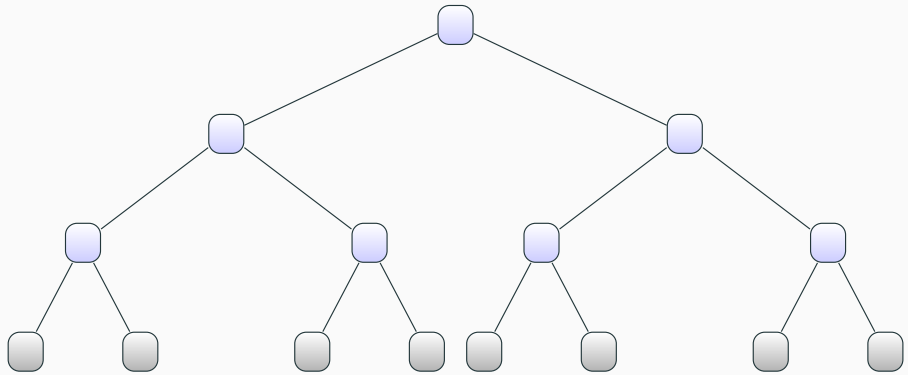


Performance

Consider a size-n binary search tree of height h

- the space used is $O(n)$
- *search*, *insertion*, *deletion* take $O(h)$ time
- The height h is
  - $O(n)$ in the worst case and
  - $O(logn)$ in the best case (**balanced**)



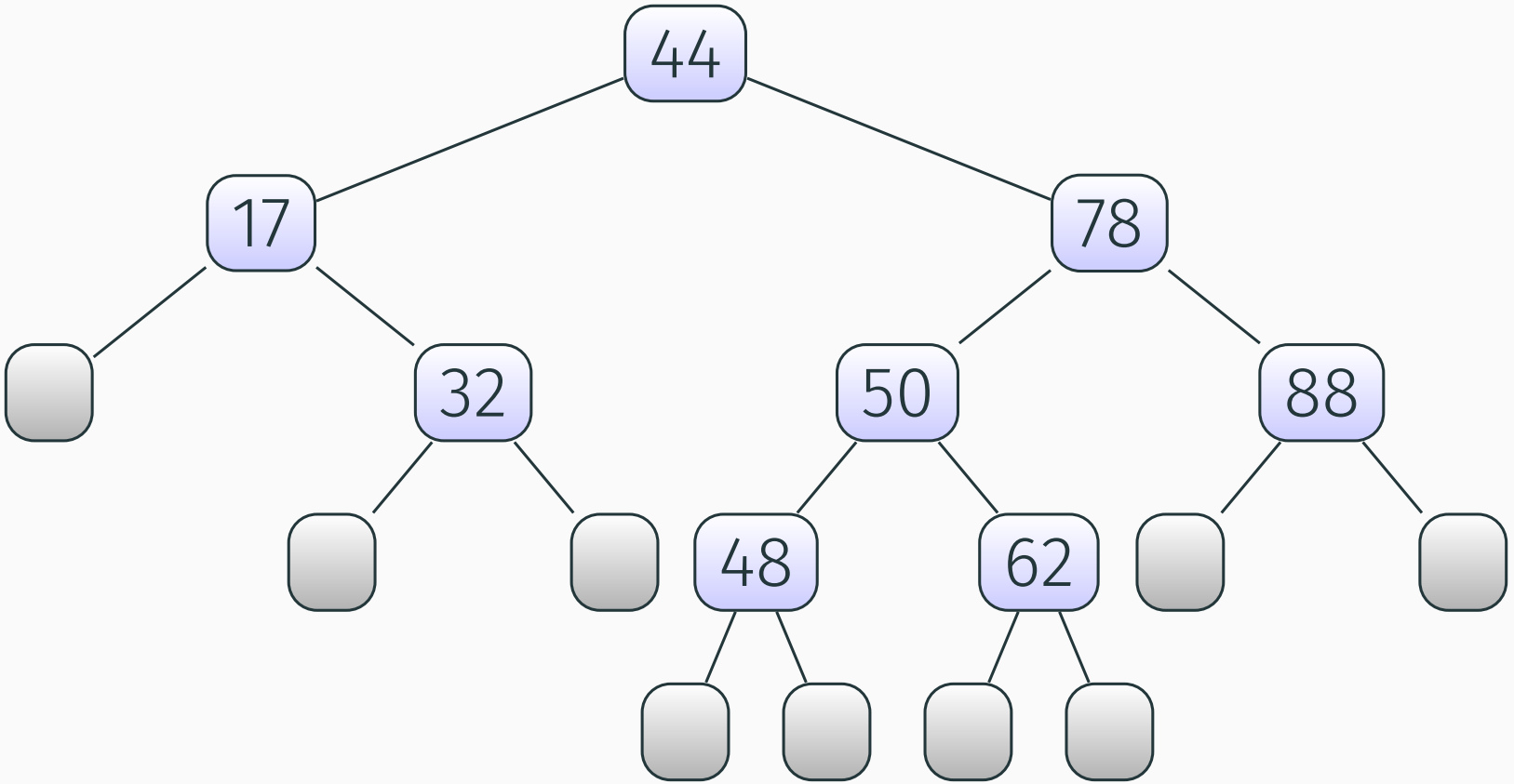


AVL Tree

AVL Tree

AVL Tree

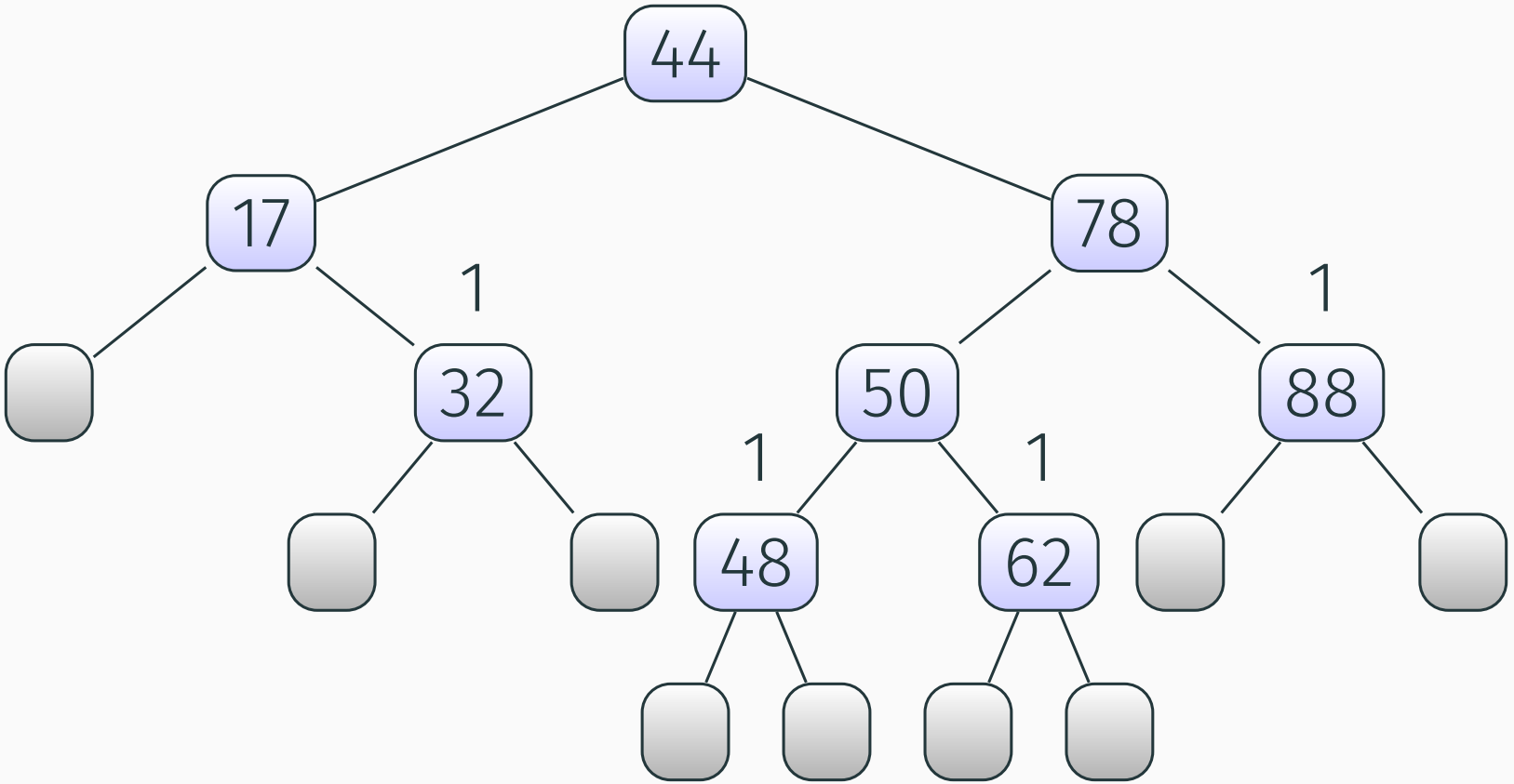
An AVL Tree is a **balanced binary search tree** such that for every internal node v of T, the heights of the children of v can differ by at most 1.



AVL Tree

AVL Tree

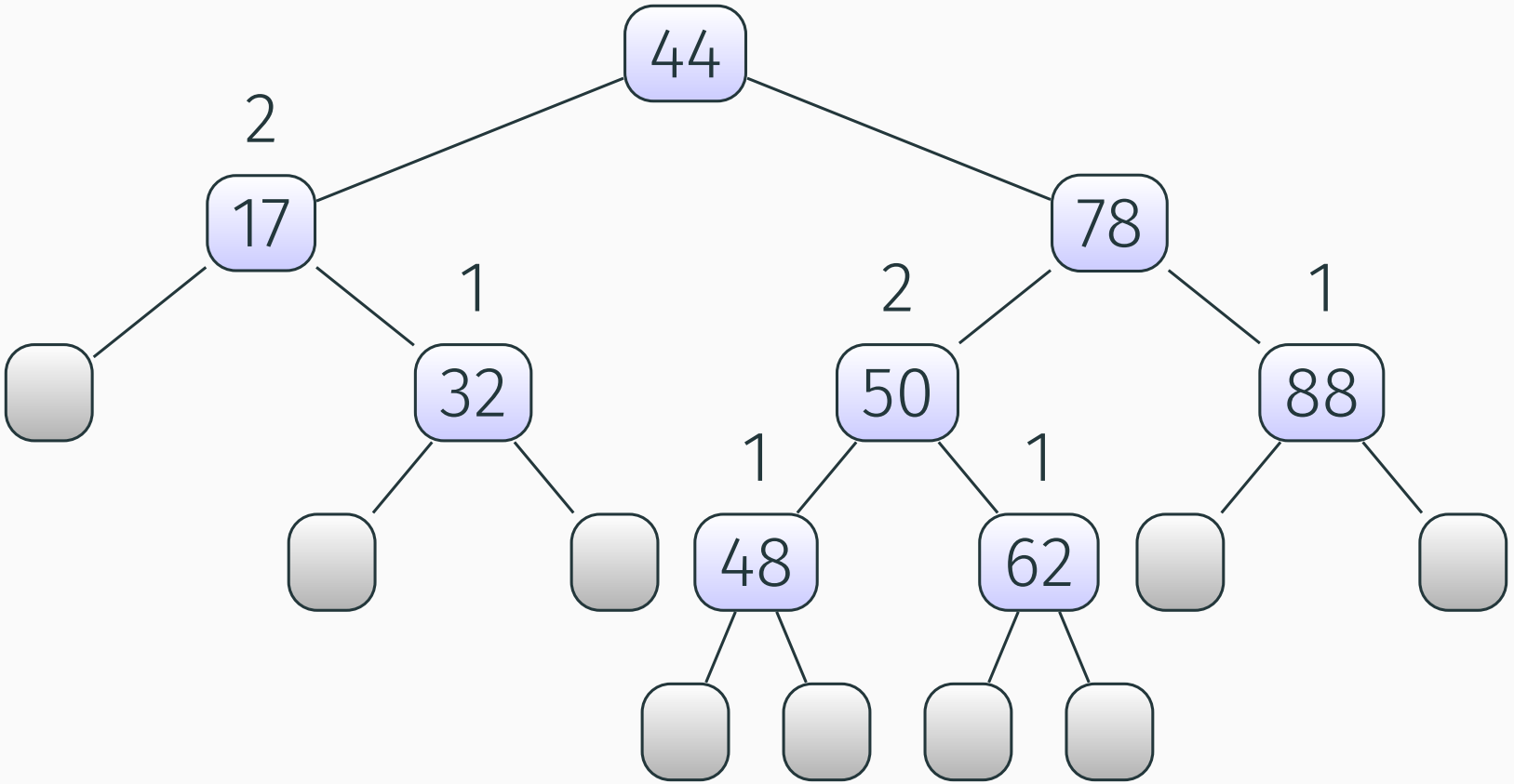
An AVL Tree is a **balanced binary search tree** such that for every internal node v of T, the heights of the children of v can differ by at most 1.



AVL Tree

AVL Tree

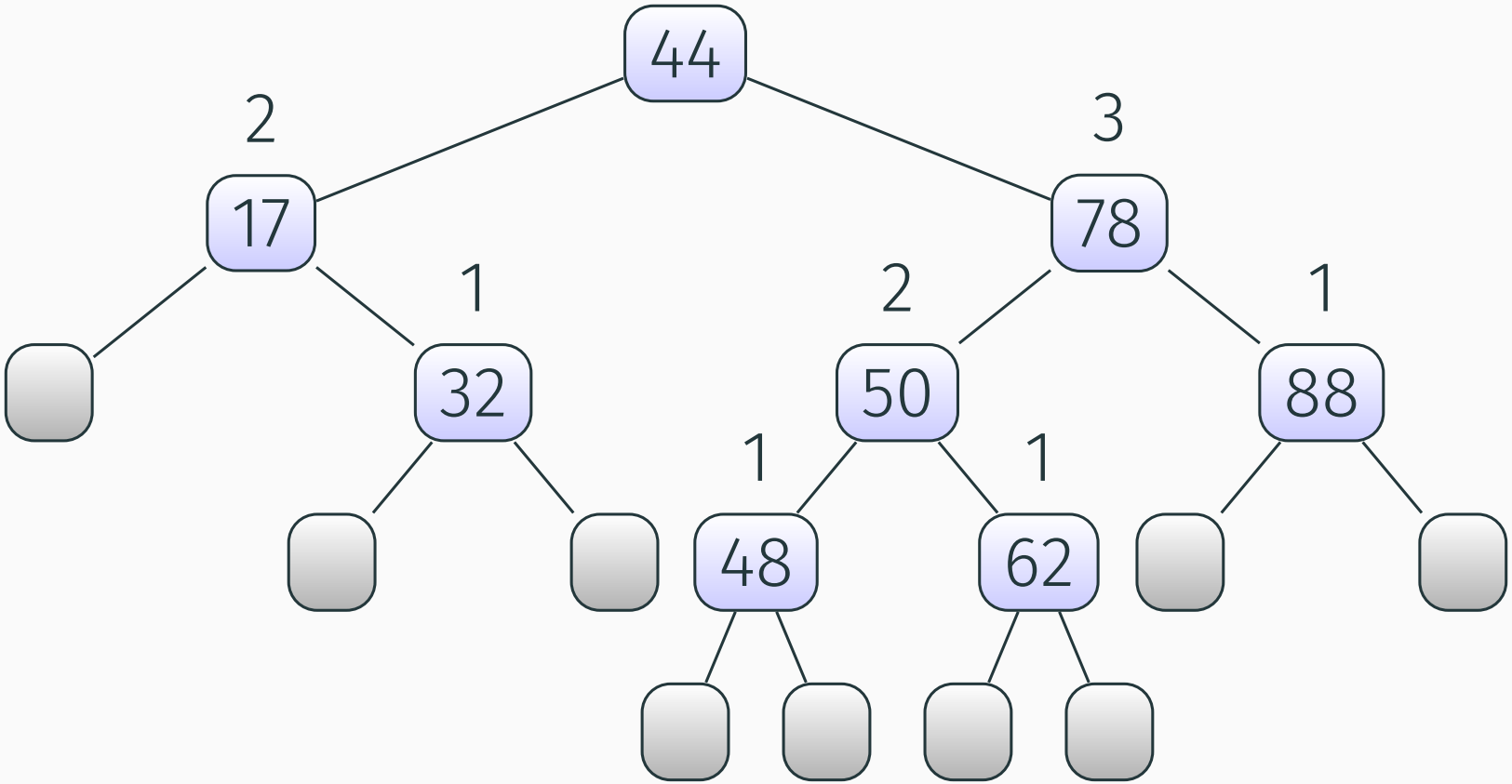
An AVL Tree is a **balanced binary search tree** such that for every internal node v of T, the heights of the children of v can differ by at most 1.



AVL Tree

AVL Tree

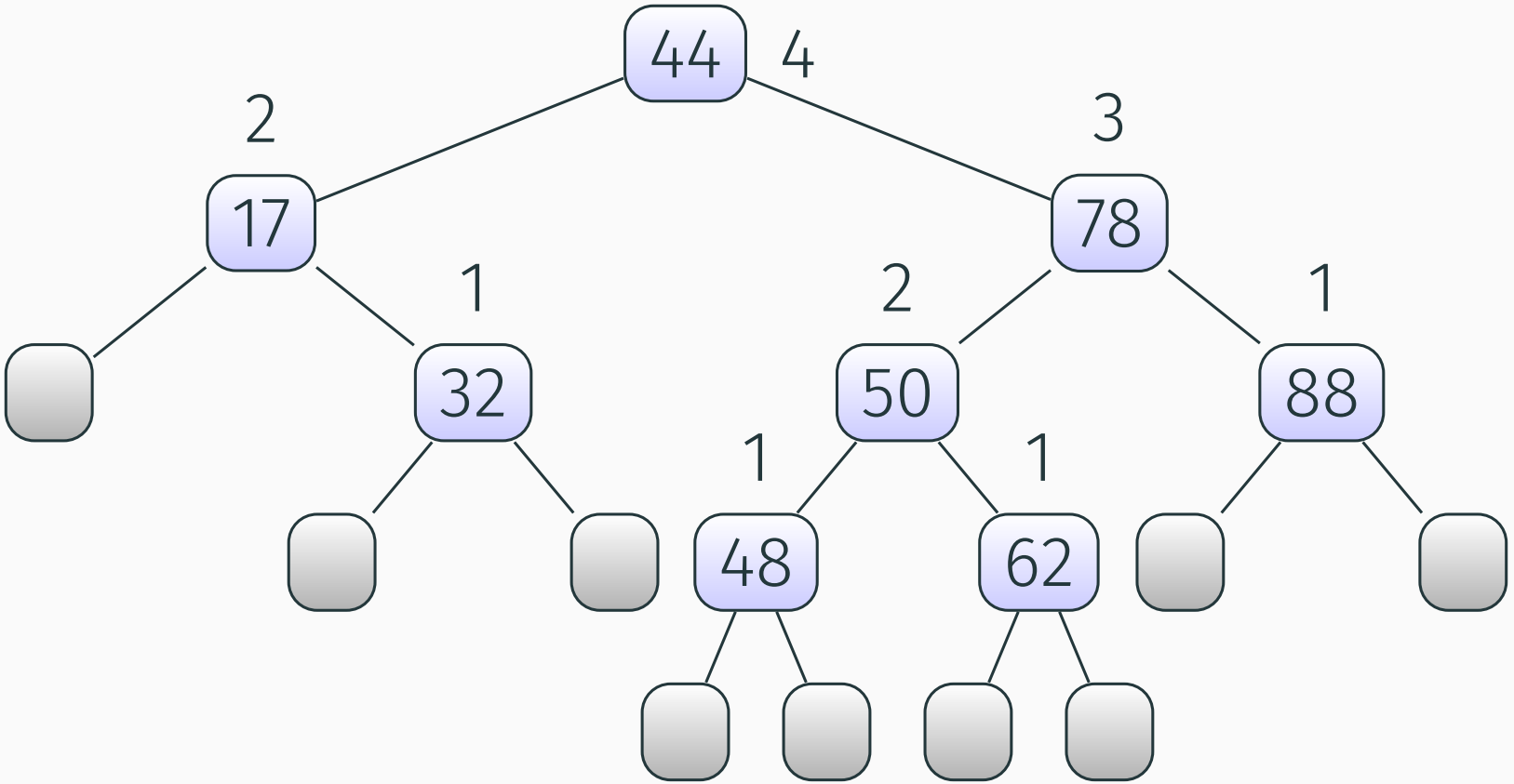
An AVL Tree is a **balanced binary search tree** such that for every internal node v of T, the heights of the children of v can differ by at most 1.



AVL Tree

AVL Tree

An AVL Tree is a **balanced binary search tree** such that for every internal node v of T, the heights of the children of v can differ by at most 1.



Height of an AVL Tree (Optional)

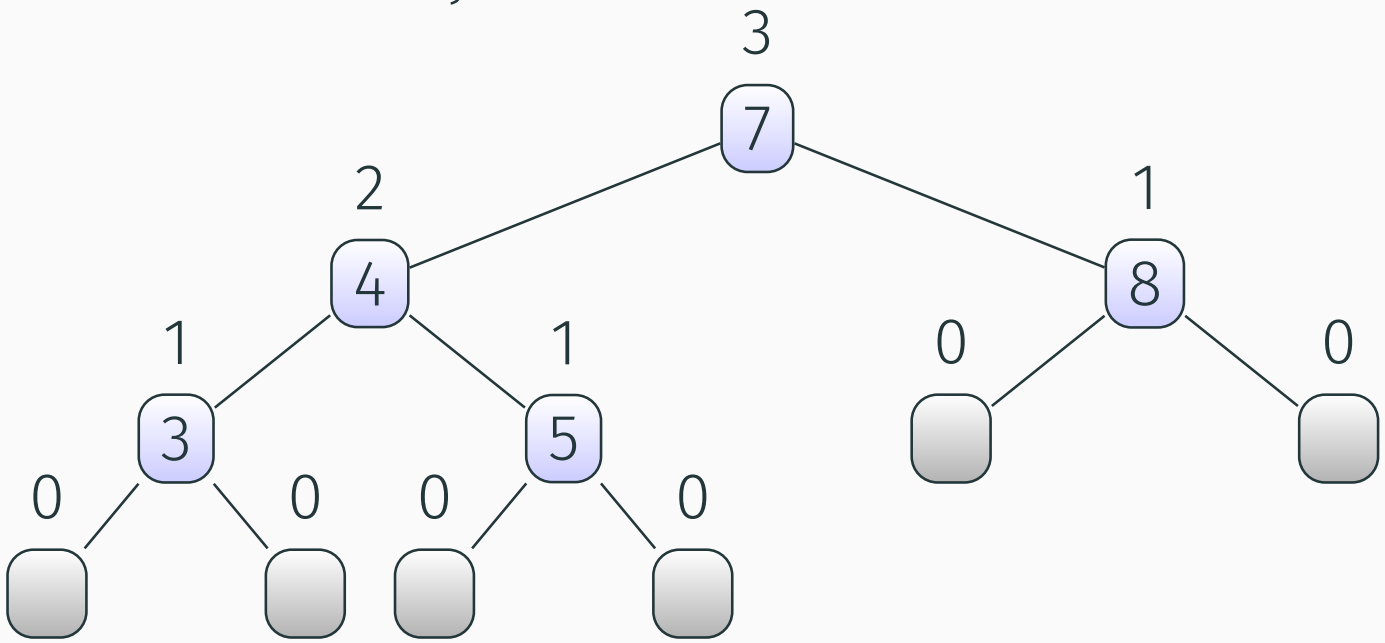
The height of an AVL tree storing n keys is $O(\log n)$.

Proof (by induction): Let us bound $n(h)$: the minimum number of internal nodes of an AVL tree of height h.

- $n(1) = 1$ and $n(2) = 2$
- For $n > 2$, an AVL tree of height h contains the root node, one AVL subtree of height $n - 1$ and another of height $n - 2$.
- That is, $n(h) = 1 + n(h - 1) + n(h - 2)$
- Knowing $n(h - 1) > n(h - 2)$, we get $n(h) > 2n(h - 2)$.
- $n(h) > 2n(h - 2) > 4n(h - 4) > 8n(n - 6) > \ldots > 2^i n(h - 2i)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: $h < 2 \log n(h) + 2$
- Thus the height of an AVL tree is $O(\log n)$

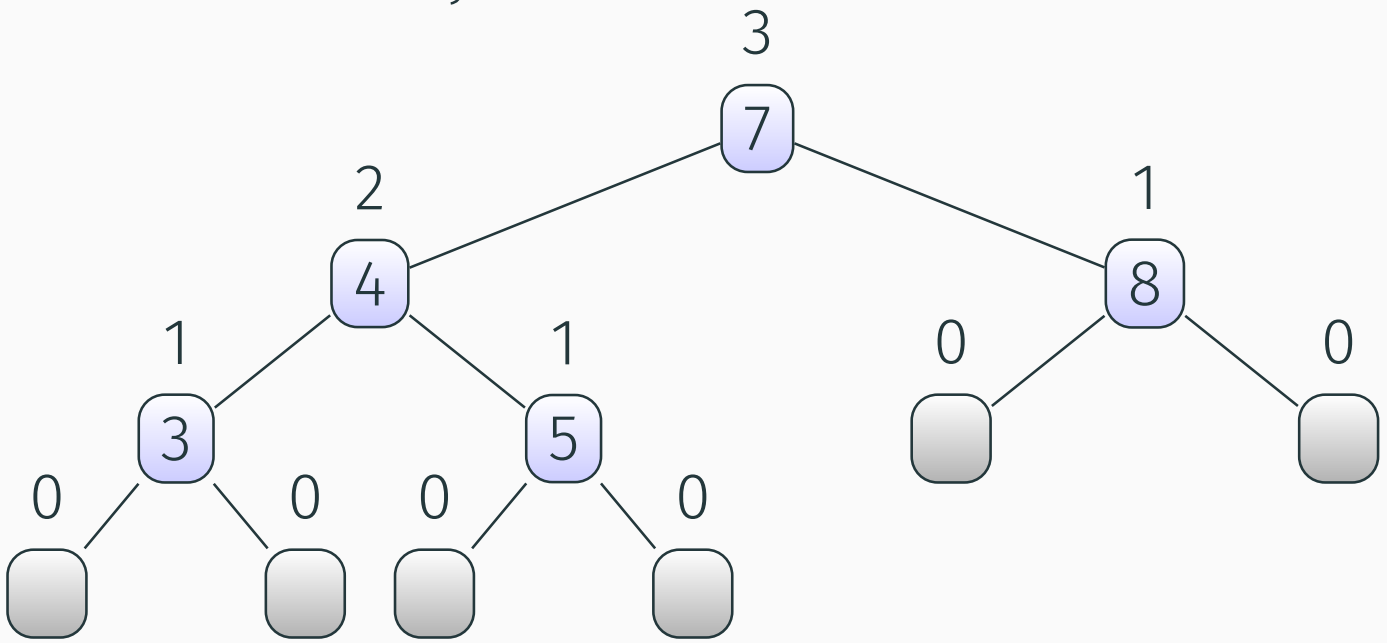
Insertion

- Insertion is as in a binary search tree



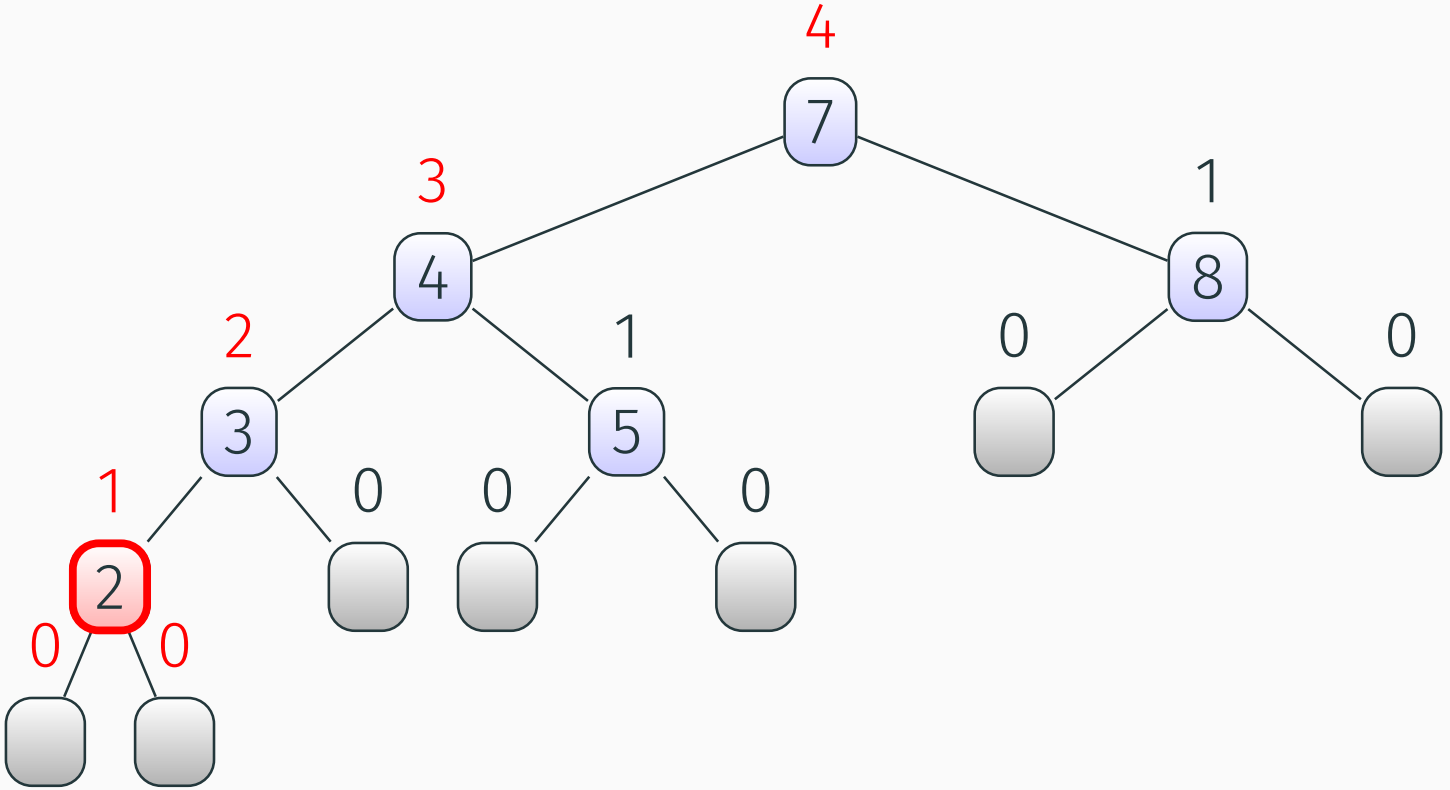
Insertion

- Insertion is as in a binary search tree insert 2



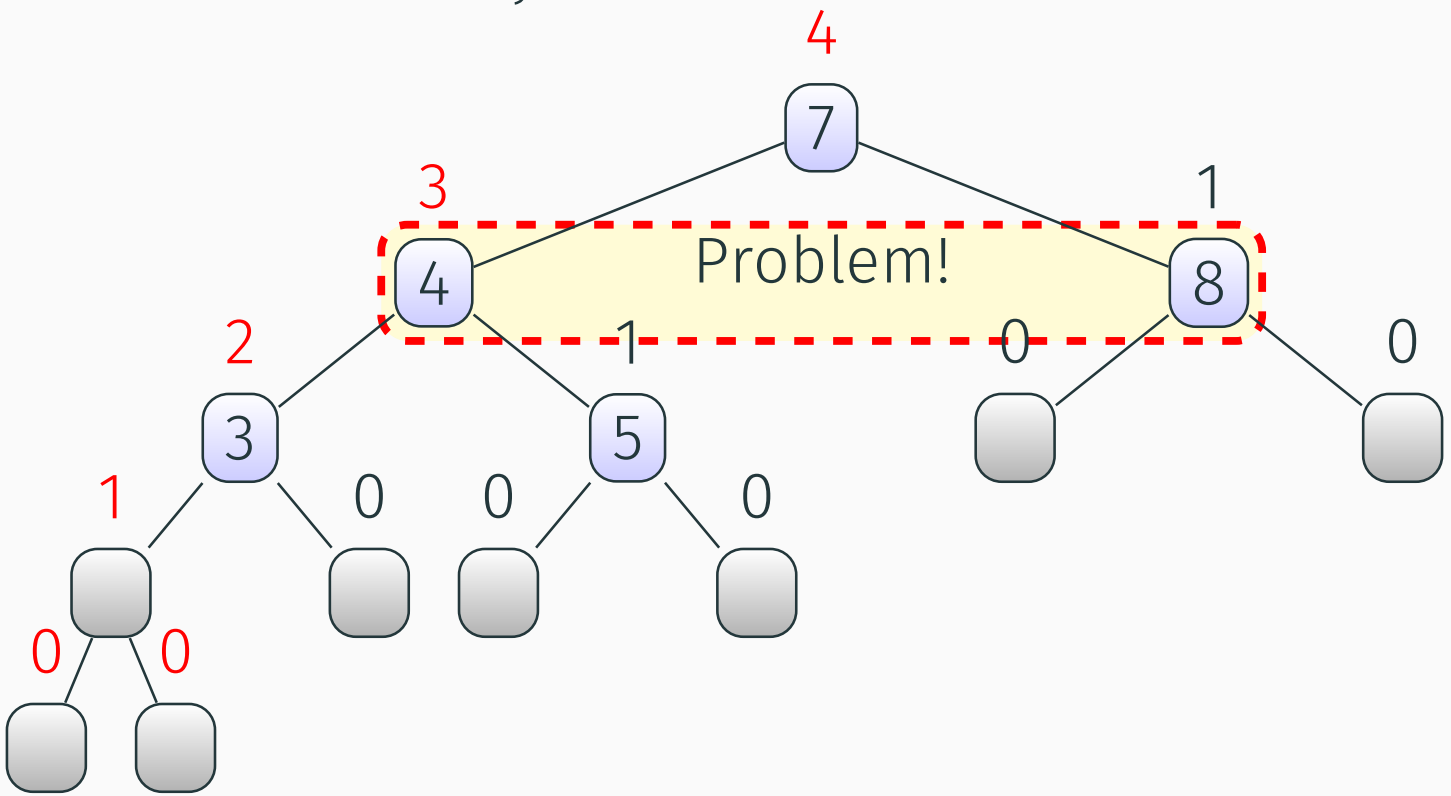
Insertion

- Insertion is as in a binary search tree insert 2



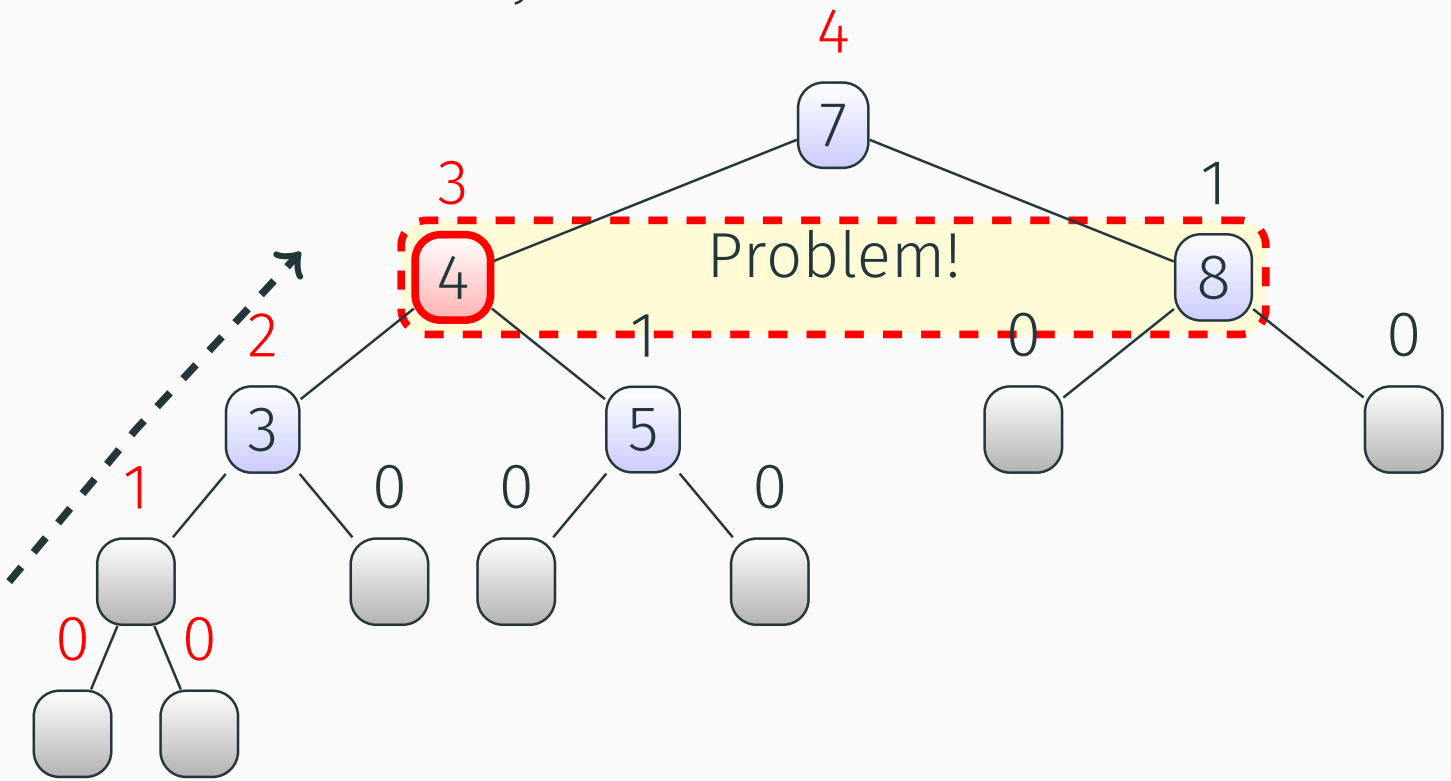
Insertion

- Insertion is as in a binary search tree insert 2



Insertion

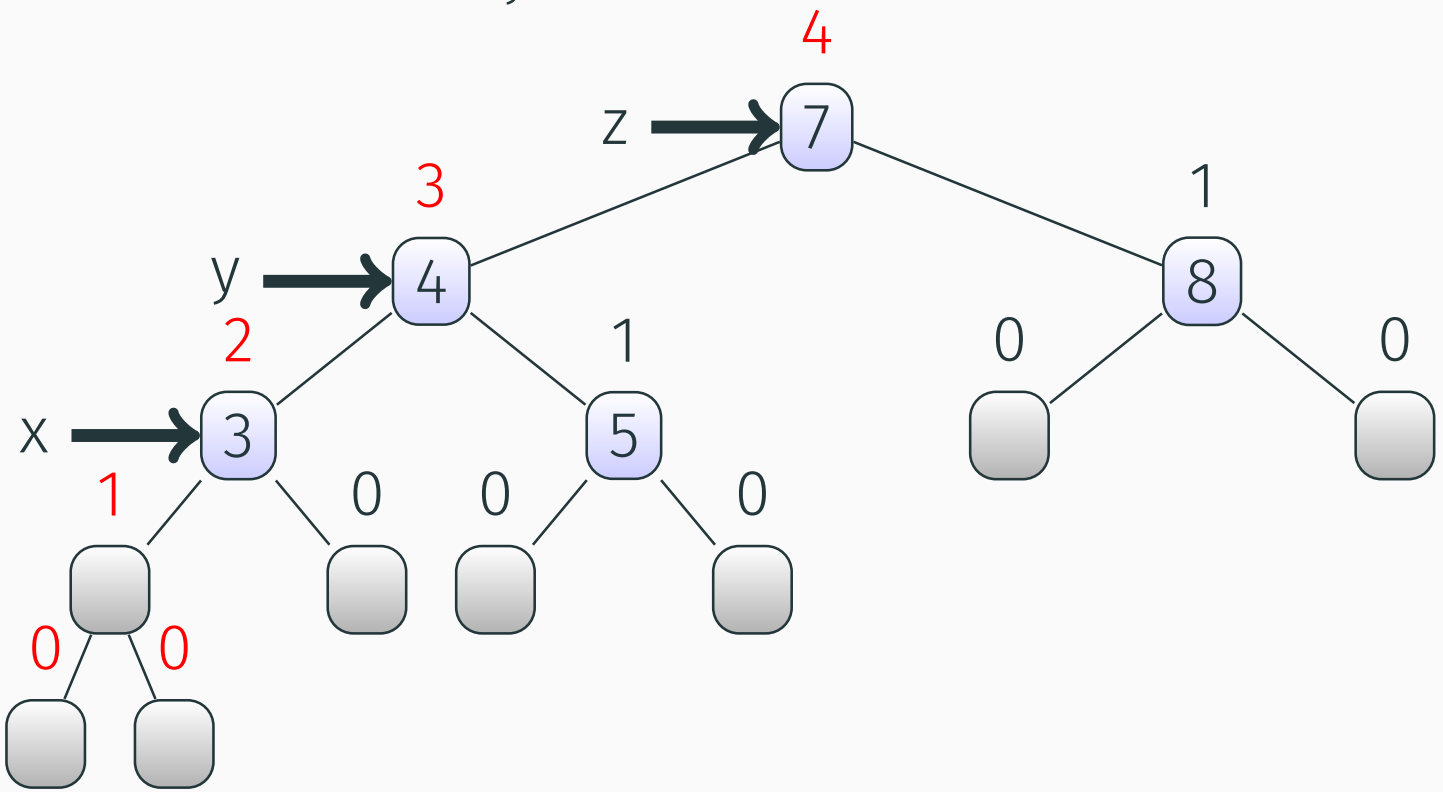
- Insertion is as in a binary search tree insert 2



Search: Starting at the inserted node, traverse toward the root until an imbalance is discovered.

Insertion

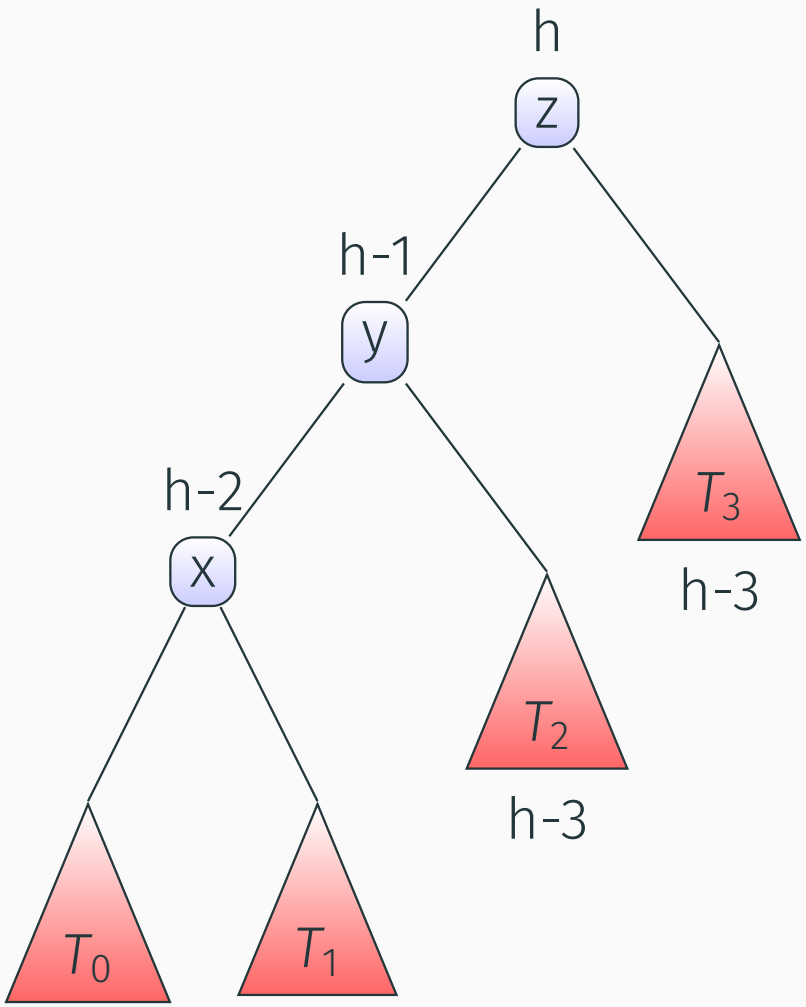
- Insertion is as in a binary search tree insert 2



Repair (trinode restructuring): 3 nodes x, y and z are distinguished:

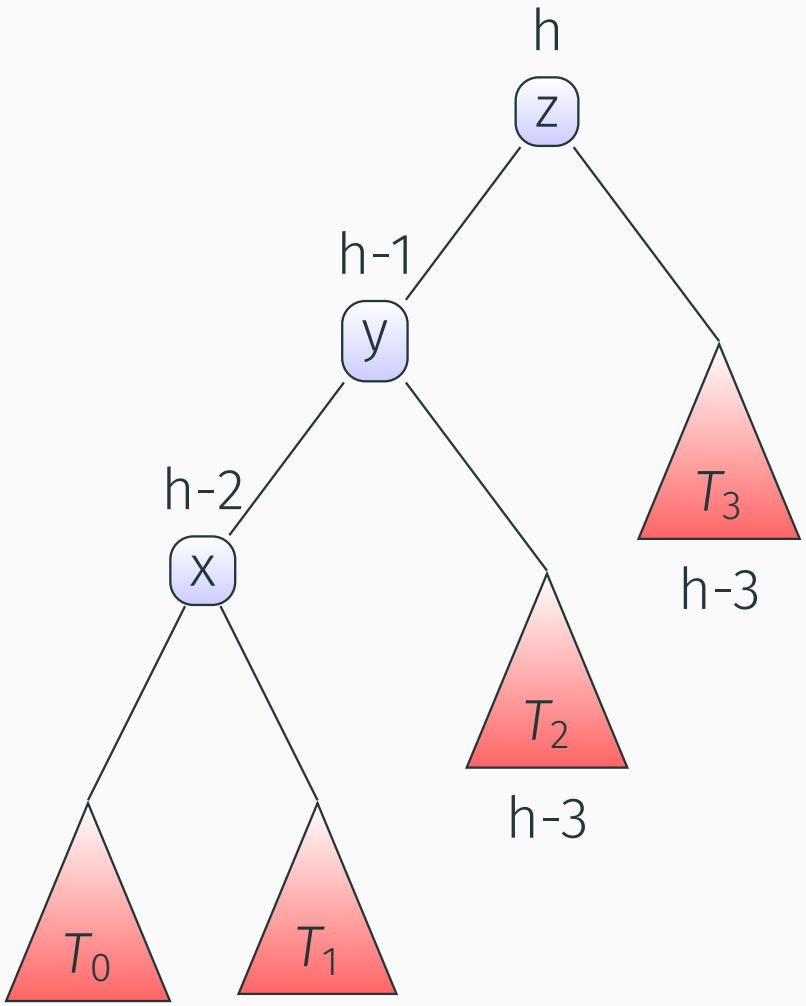
- z = the parent of the high sibling
- y = the high sibling
- x = the high child of the high sibling

Insertion: Trinode Restructuring – Case 1: $x \leq y \leq z$

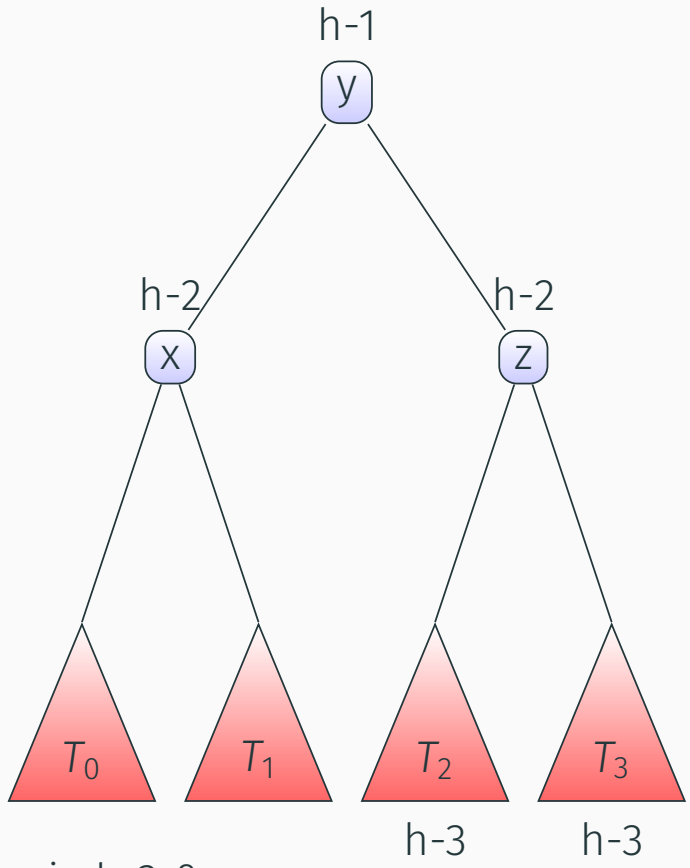


one is h-3 & one is h-4

Insertion: Trinode Restructuring – Case 1: $x \leq y \leq z$

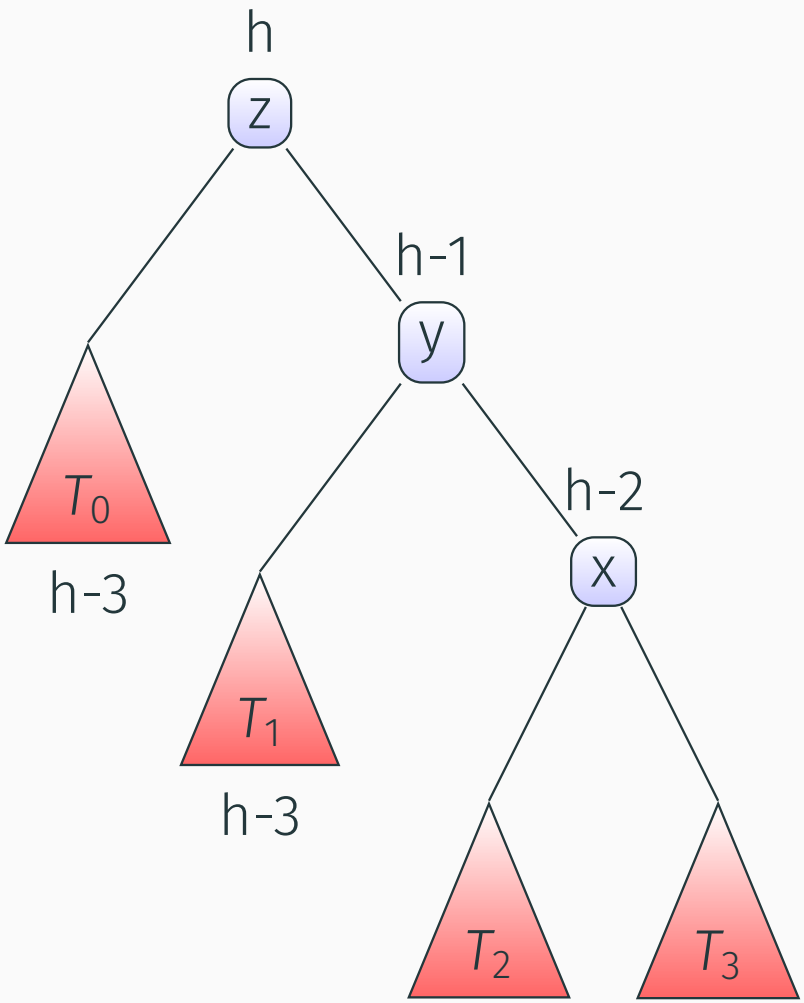


one is h-3 & one is h-4



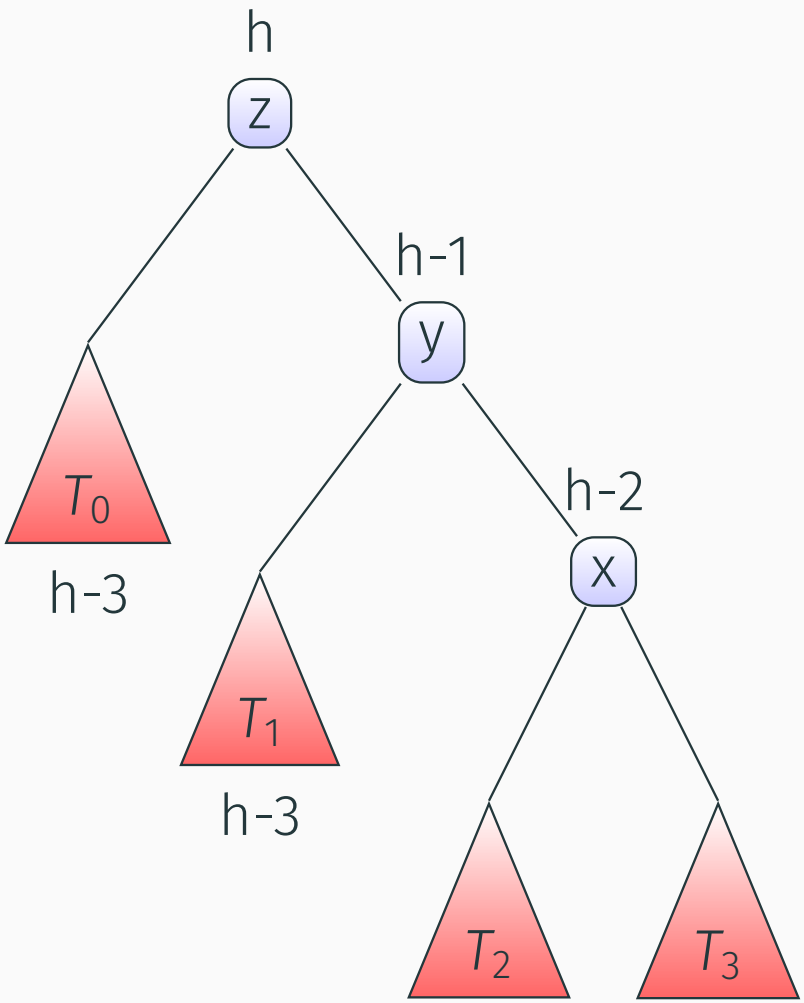
one is h-3 &
one is h-4

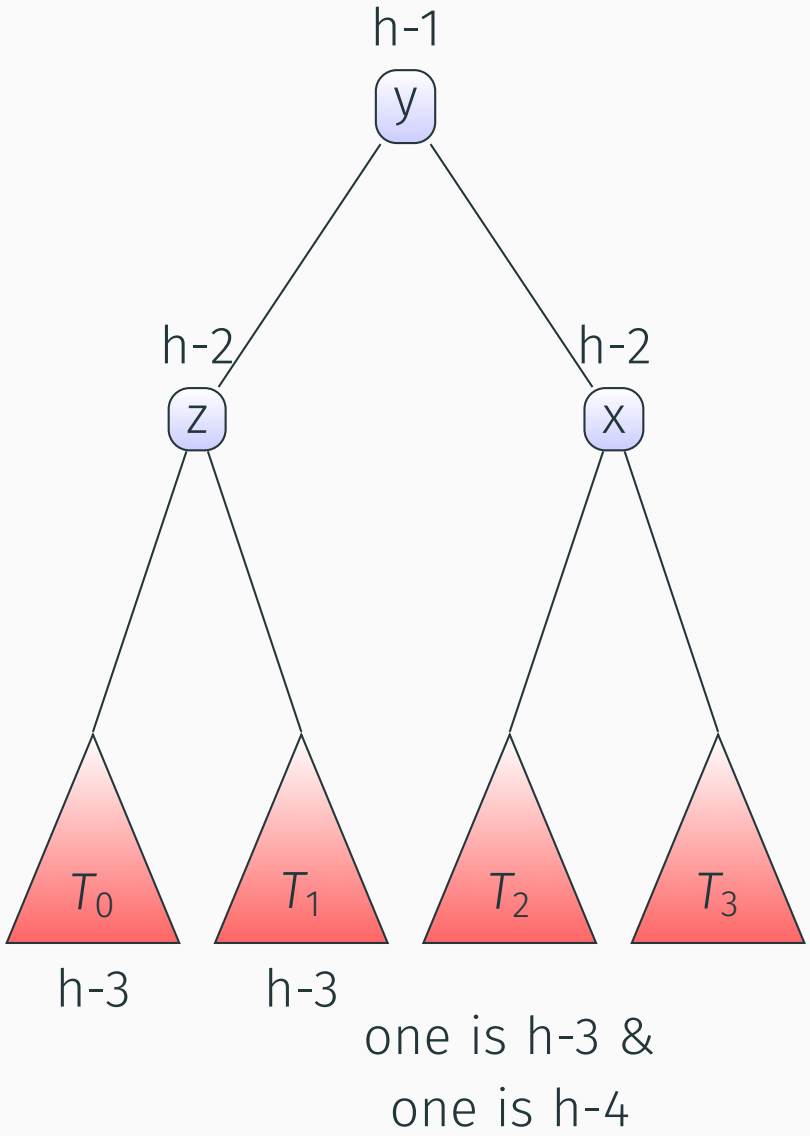
Insertion: Trinode Restructuring – Case 2: $z \leq y \leq x$



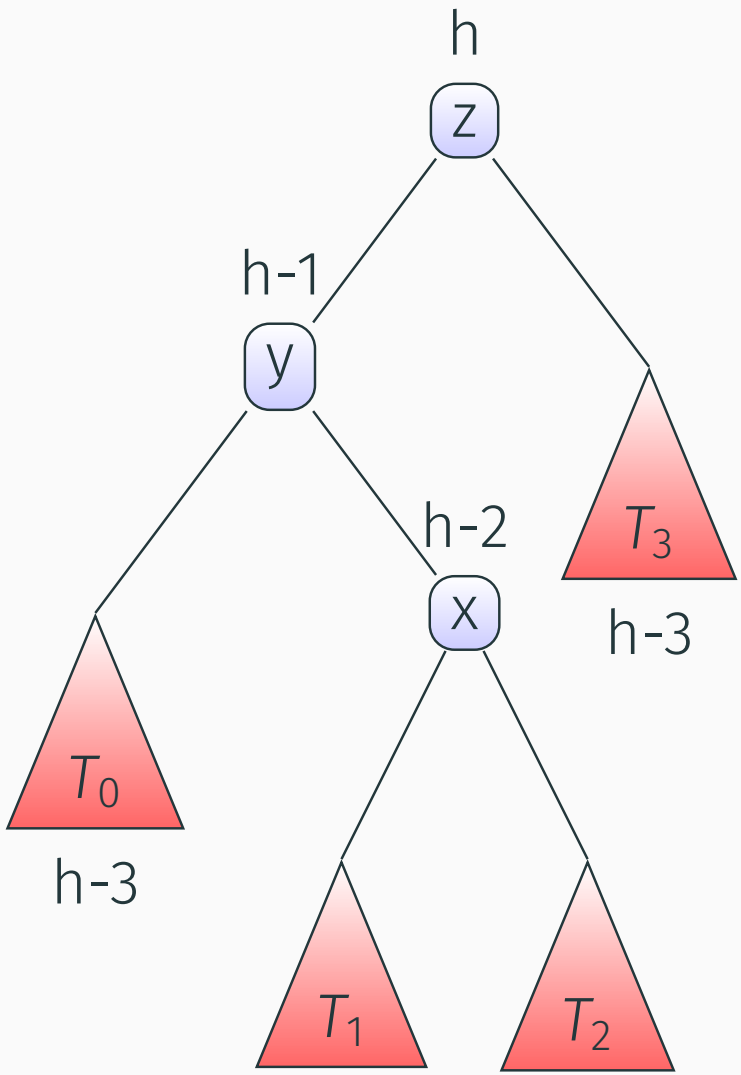
one is h-3 & one is h-4

Insertion: Trinode Restructuring – Case 2: $z \leq y \leq x$



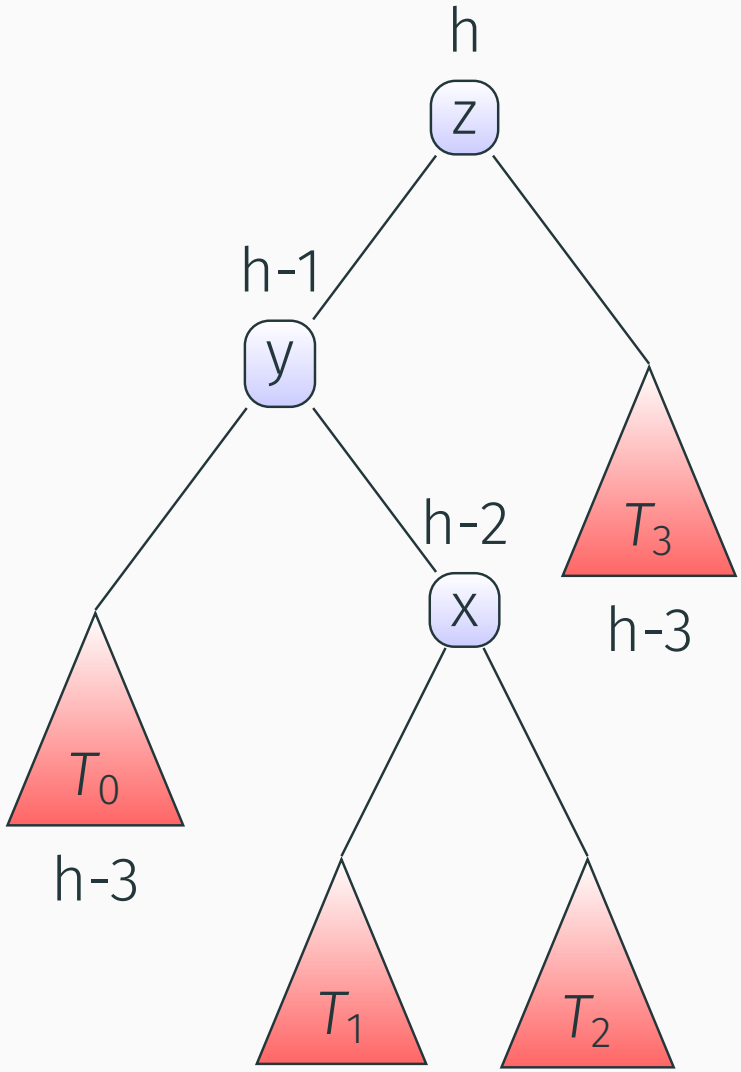


Insertion: Trinode Restructuring – Case 3: $y \leq x \leq z$

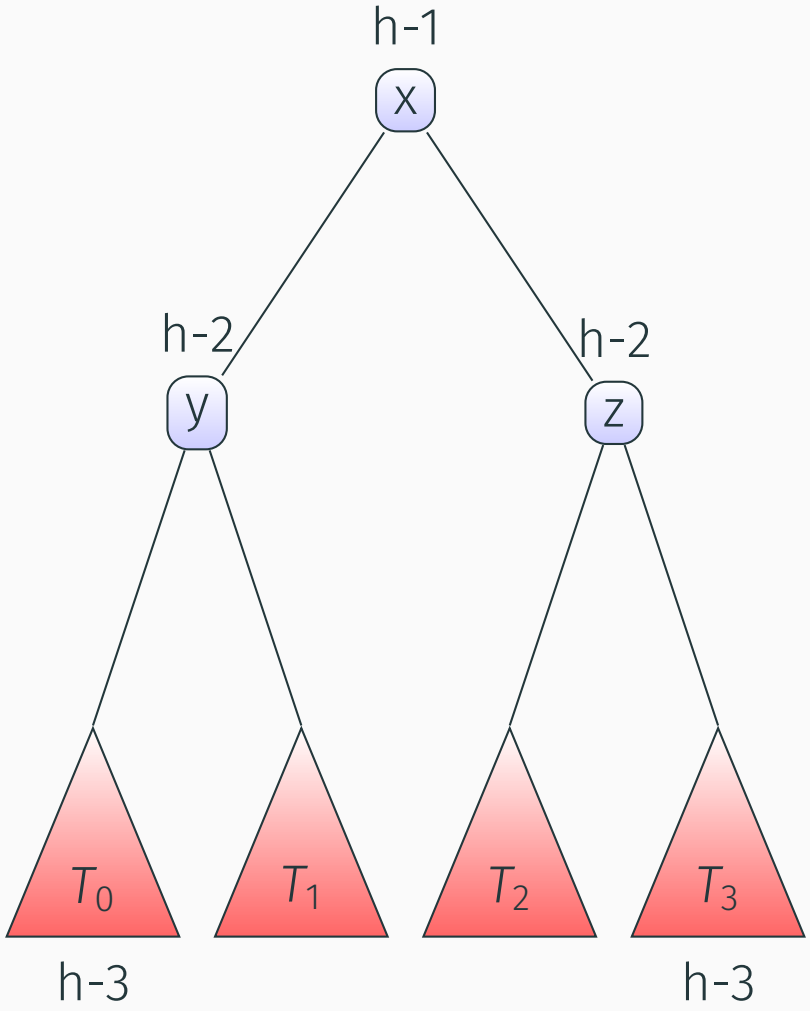


one is h-3 & one is h-4

Insertion: Trinode Restructuring – Case 3: $y \leq x \leq z$

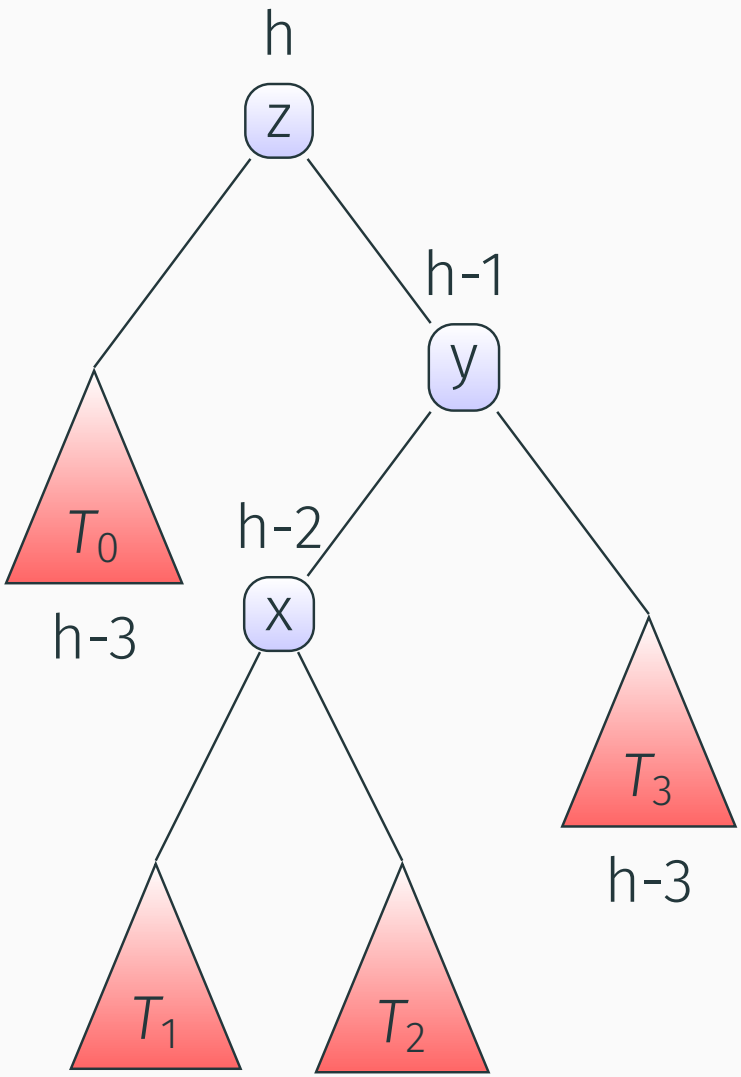


one is h-3 & one is h-4



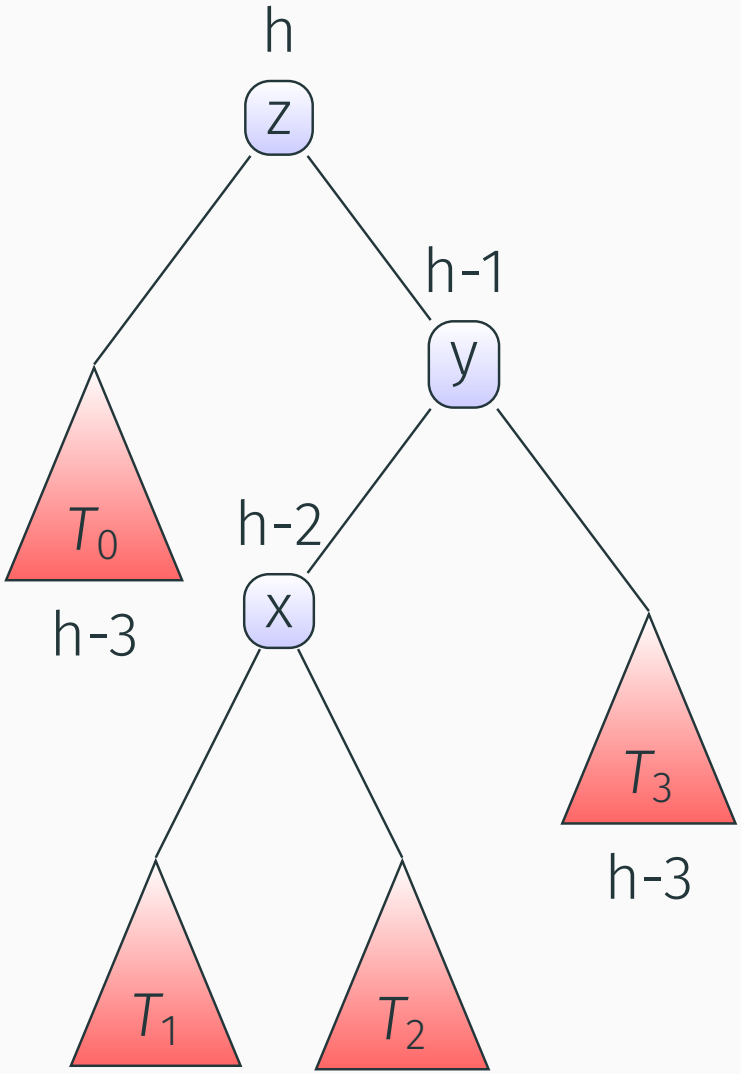
one is h-3 & one is h-4

Insertion: Trinode Restructuring – Case 4: $z \leq x \leq y$

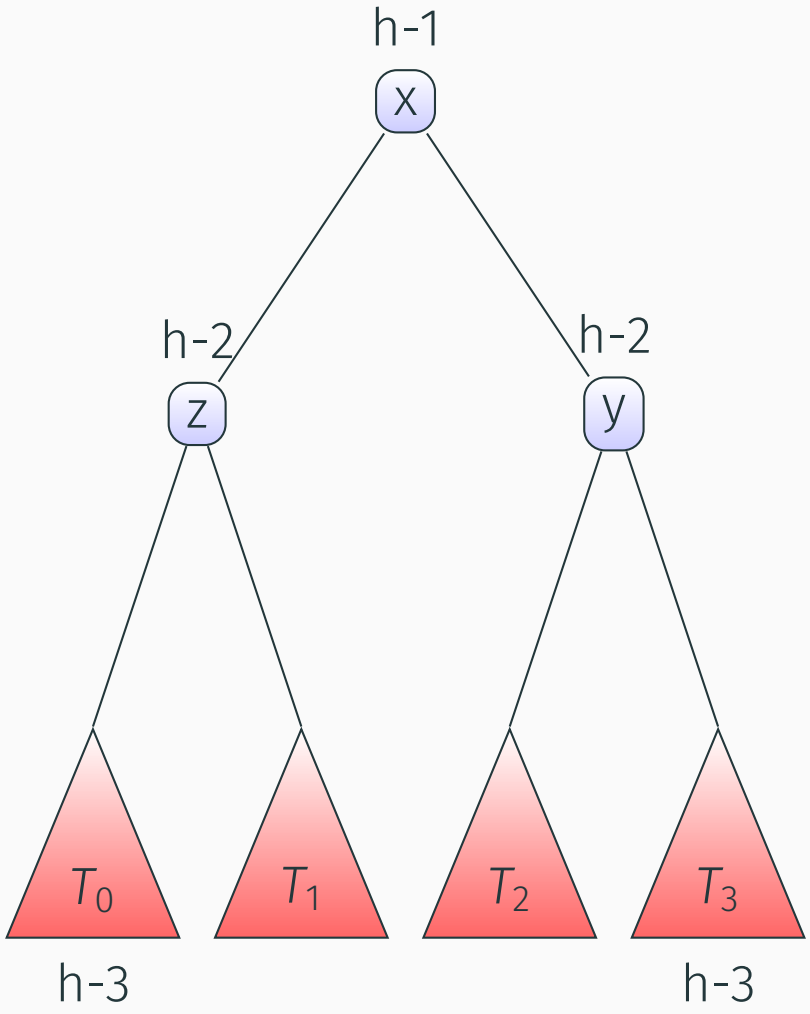


one is h-3 & one is h-4

Insertion: Trinode Restructuring – Case 4: $z \leq x \leq y$



one is h-3 & one is h-4



one is h-3 & one is h-4

Insertion: Trinode Restructuring - the Whole tree

Do we have to repeat this process further up the tree?

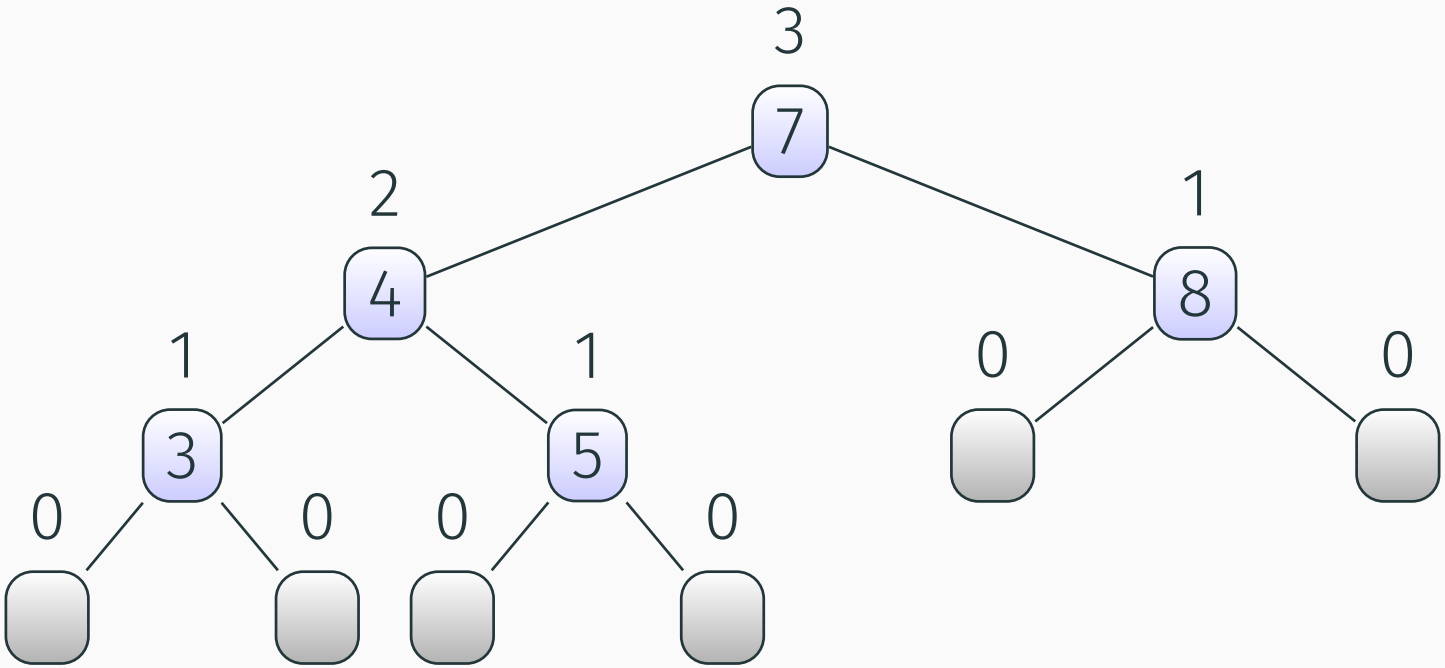
Insertion: Trinode Restructuring - the Whole tree

Do we have to repeat this process further up the tree?

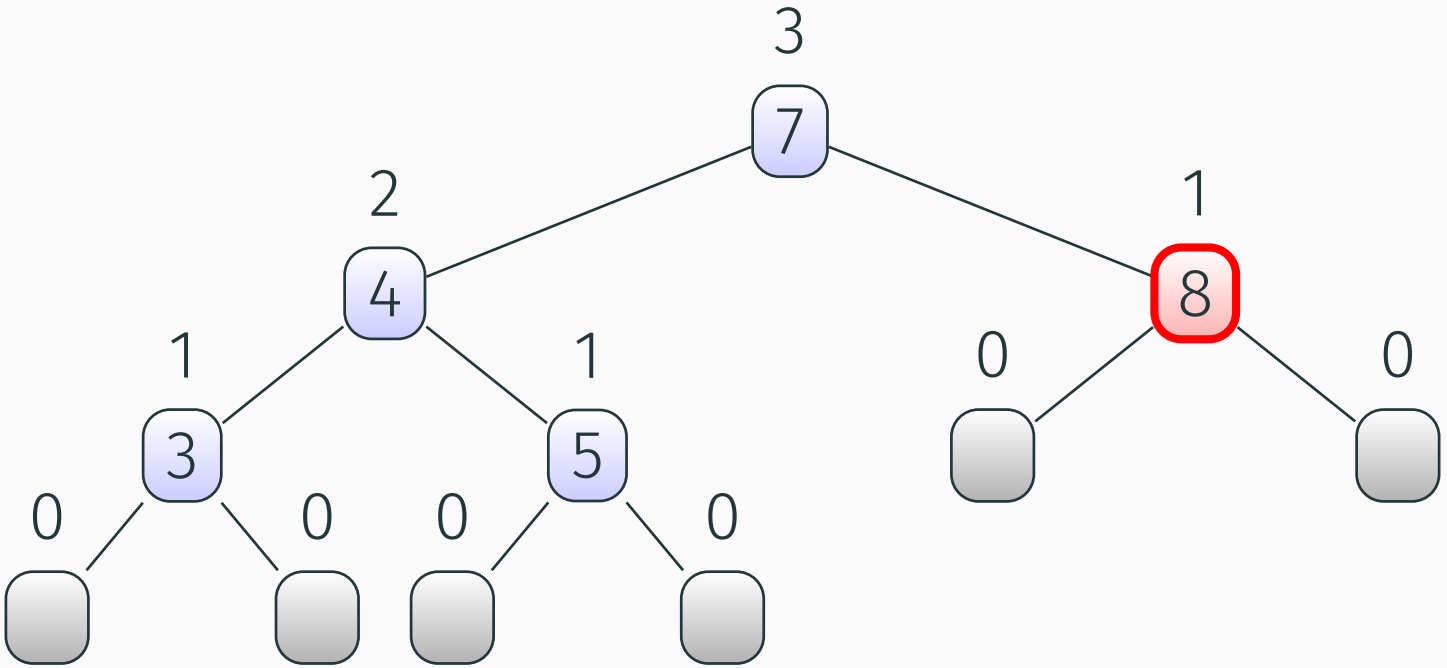
No!

- The tree was balanced before the insertion.
- Insertion raised the height of the subtree by 1.
- Rebalancing lowered the height of the subtree by 1.
- Thus the whole tree is still balanced.

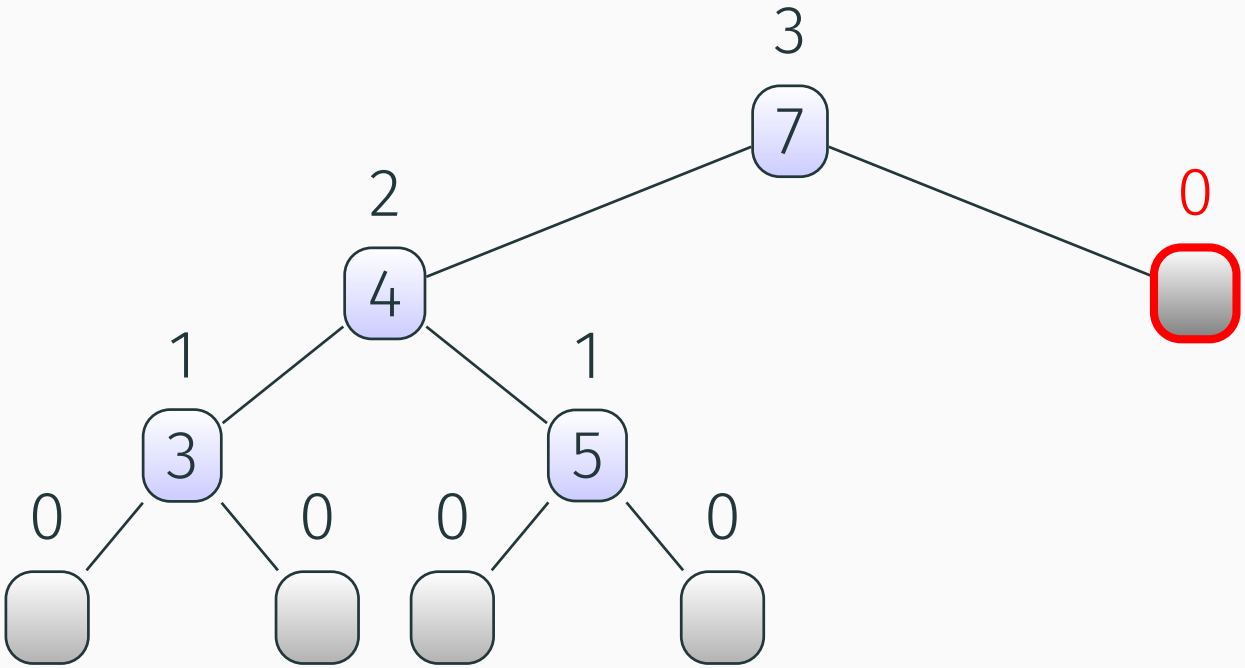
Deletion



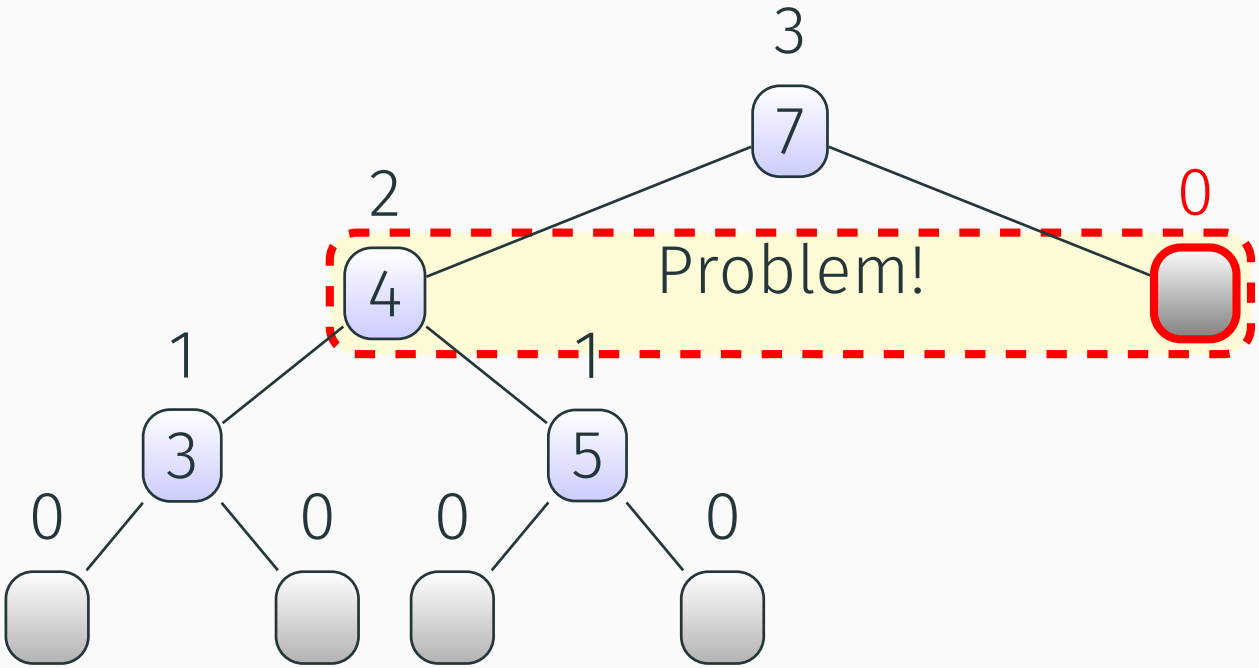
Deletion



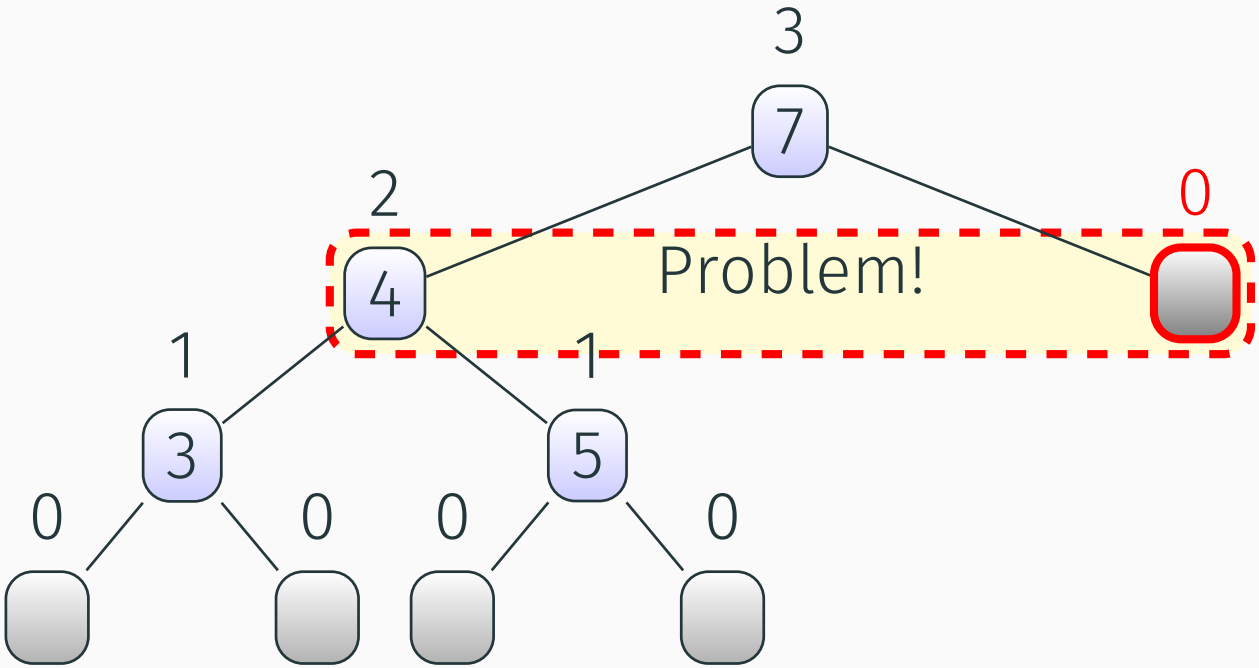
Deletion



Deletion



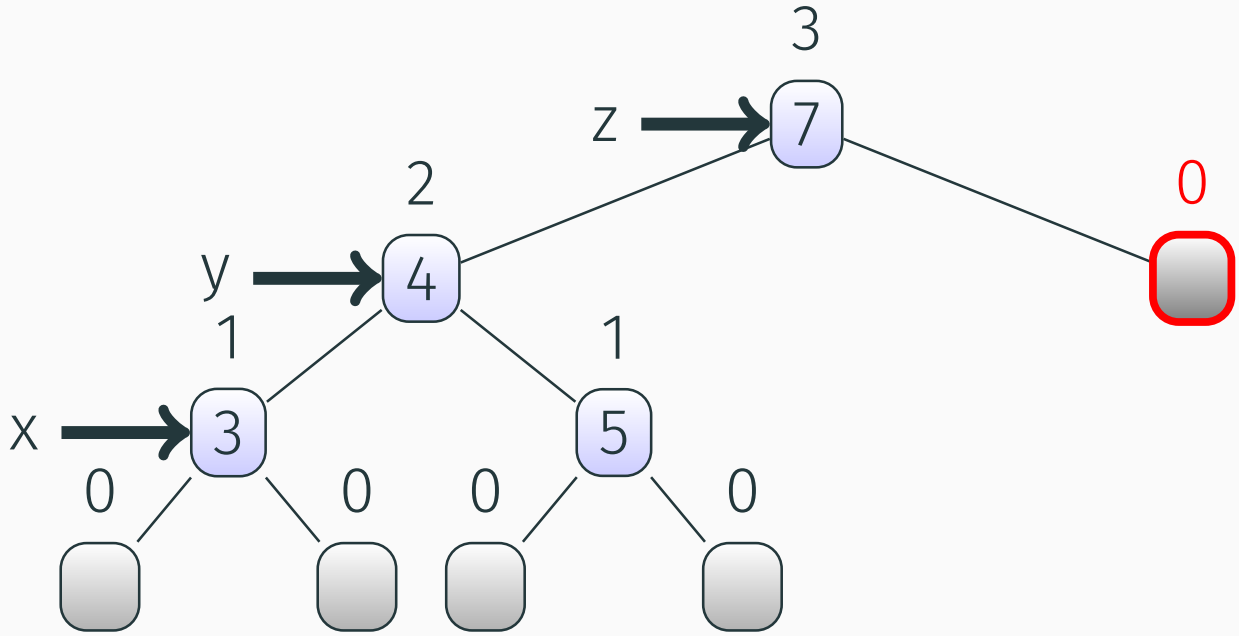
Deletion



Search:

- Let w be the node actually removed (*i.e.*, the node matching the key if it has a leaf child, otherwise the node following in an in-order traversal.)
- Starting at w, traverse toward the root until an imbalance is discovered.

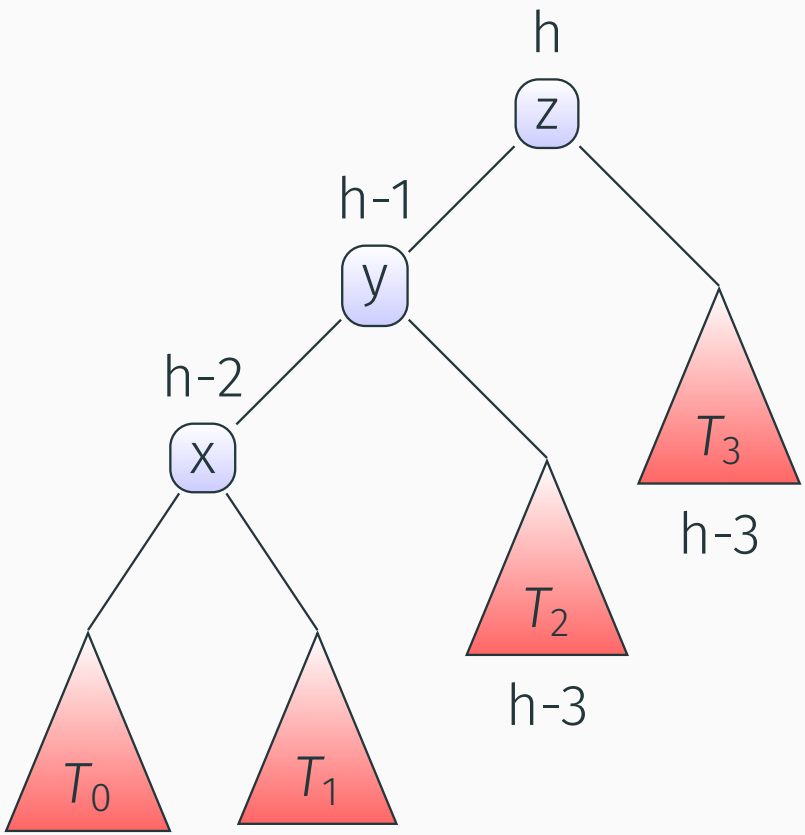
Deletion



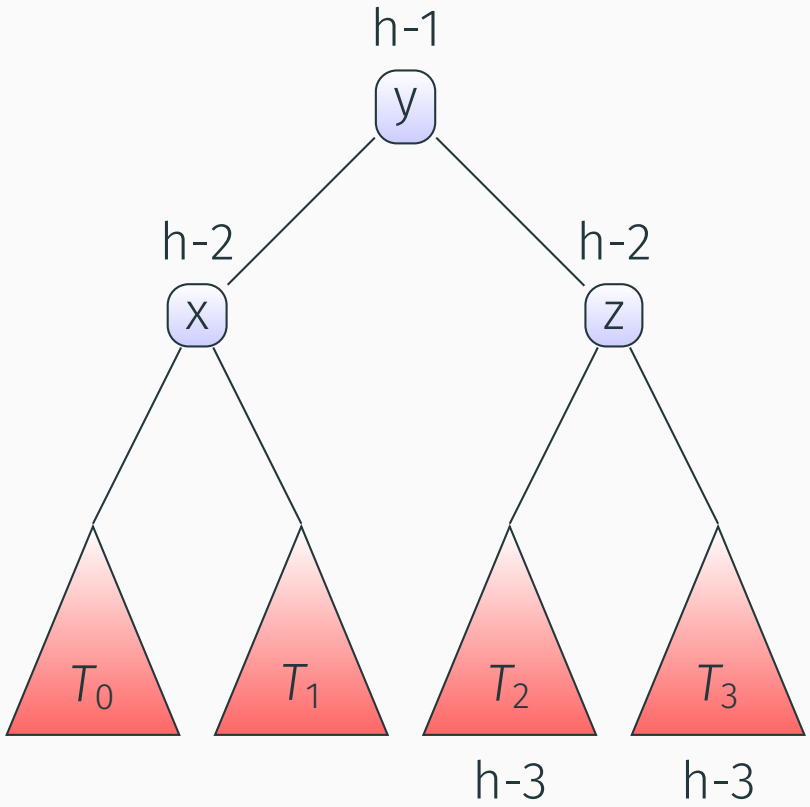
Repair (trinode restructuring): 3 nodes x, y and z are distinguished:

- z = the parent of the high sibling
- y = the high sibling
- x = the high child of the high sibling

Deletion: Trinode Restructuring



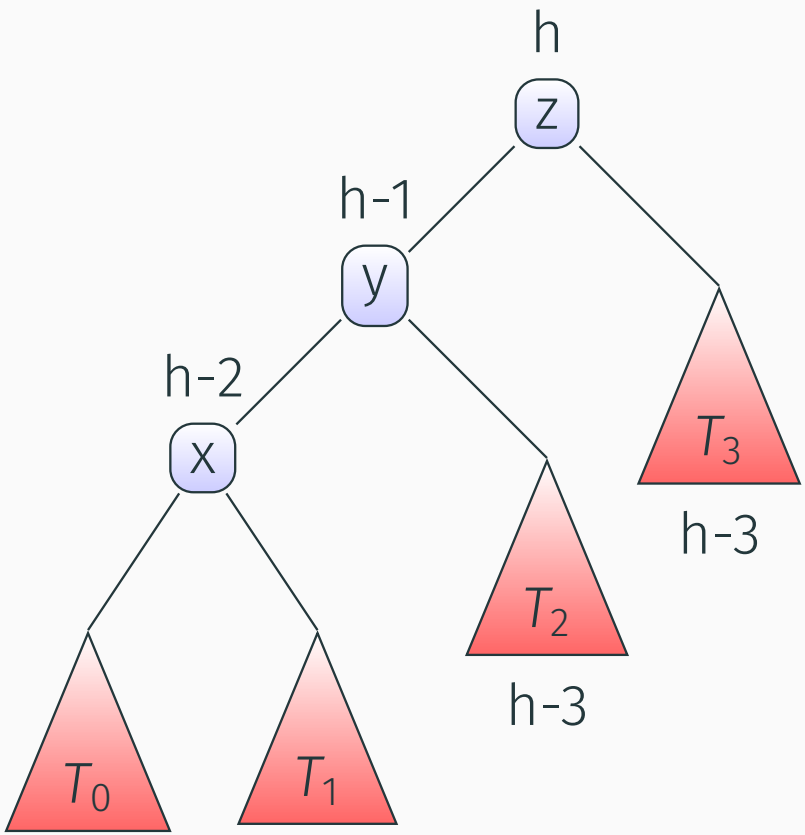
one is h-3 & one is h-4



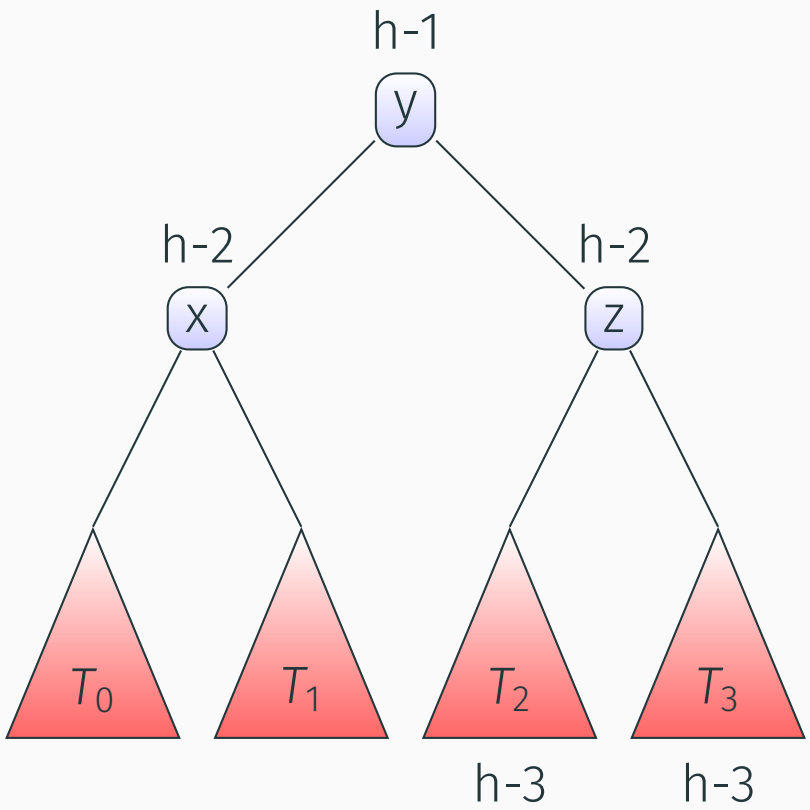
one is h-3 & one is h-4

Do we have to repeat this process further up the tree?

Deletion: Trinode Restructuring



one is h-3 & one is h-4



one is h-3 & one is h-4

Do we have to repeat this process further up the tree? YES!

- Unfortunately, trinode restructuring may reduce the height of the subtree, causing another imbalance further up the tree.

AVL Tree Performance

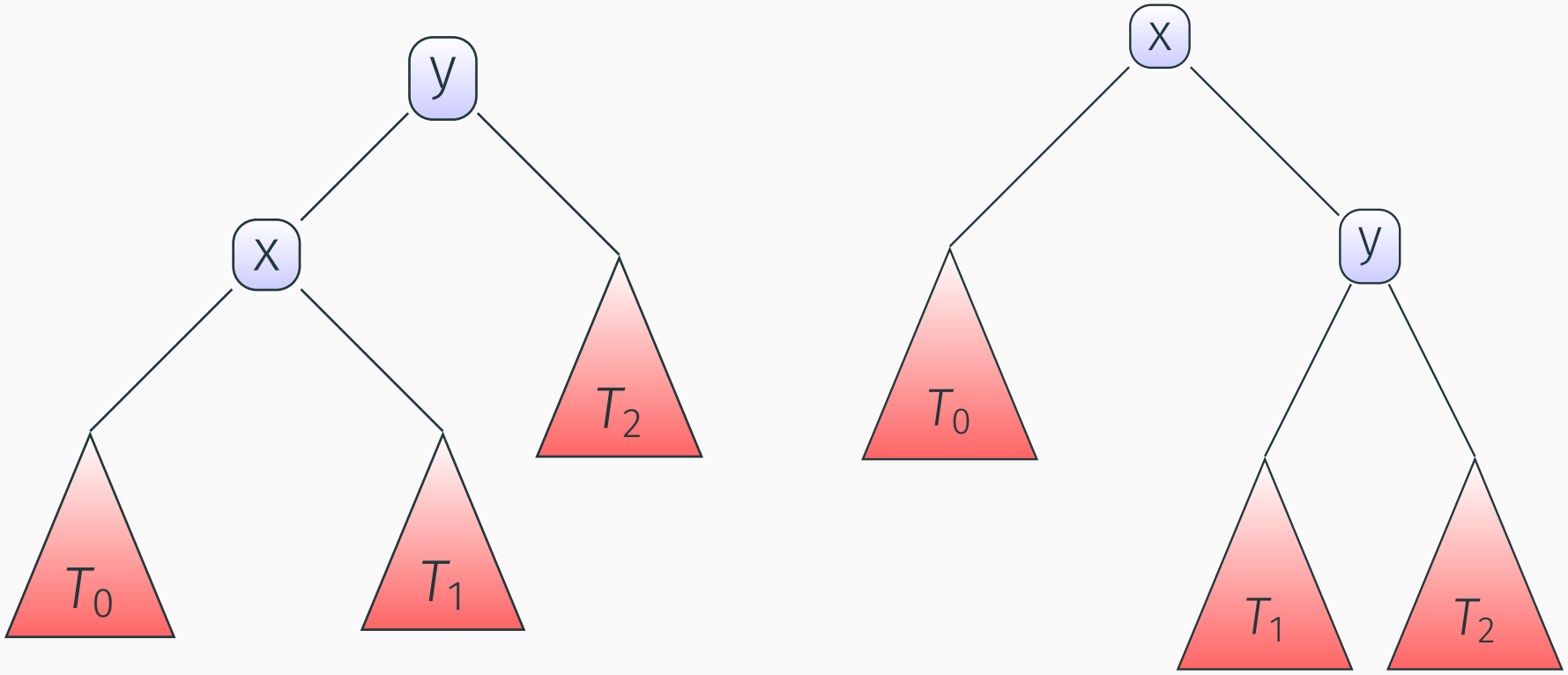
Suppose an AVL tree storing n items

- The data structure uses $O(n)$ space
- A single restructuring takes $O(1)$ time
  - using a linked-structure binary tree
- Searching takes $O(\log n)$ time
  - height of tree is $O(\log n)$, no restructures needed
- Insertion takes $O(\log n)$ time
  - initial find is $O(\log n)$
  - restructuring up the tree, maintaining heights is $O(1)$
- Removal takes $O(\log n)$ time
  - initial find is $O(\log n)$
  - restructuring up the tree, maintaining heights is $O(\log n)$

Red-Black Tree

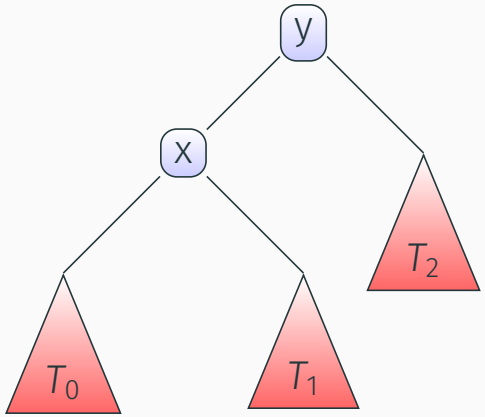
Rotations

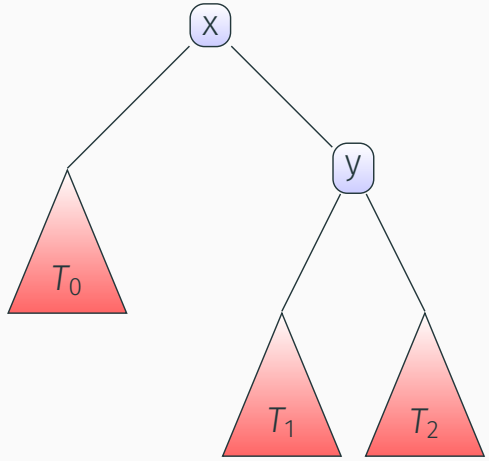
$\Longrightarrow$: Right-rotation; $\Longleftarrow$: Left-rotation



- Right-Rotate: the old-root becomes the right child of the new root.
- Left-Rotate: the old-root becomes the left child of the new root.

Left-rotation (`Right-Rotate` pseudo-code is symmetric)

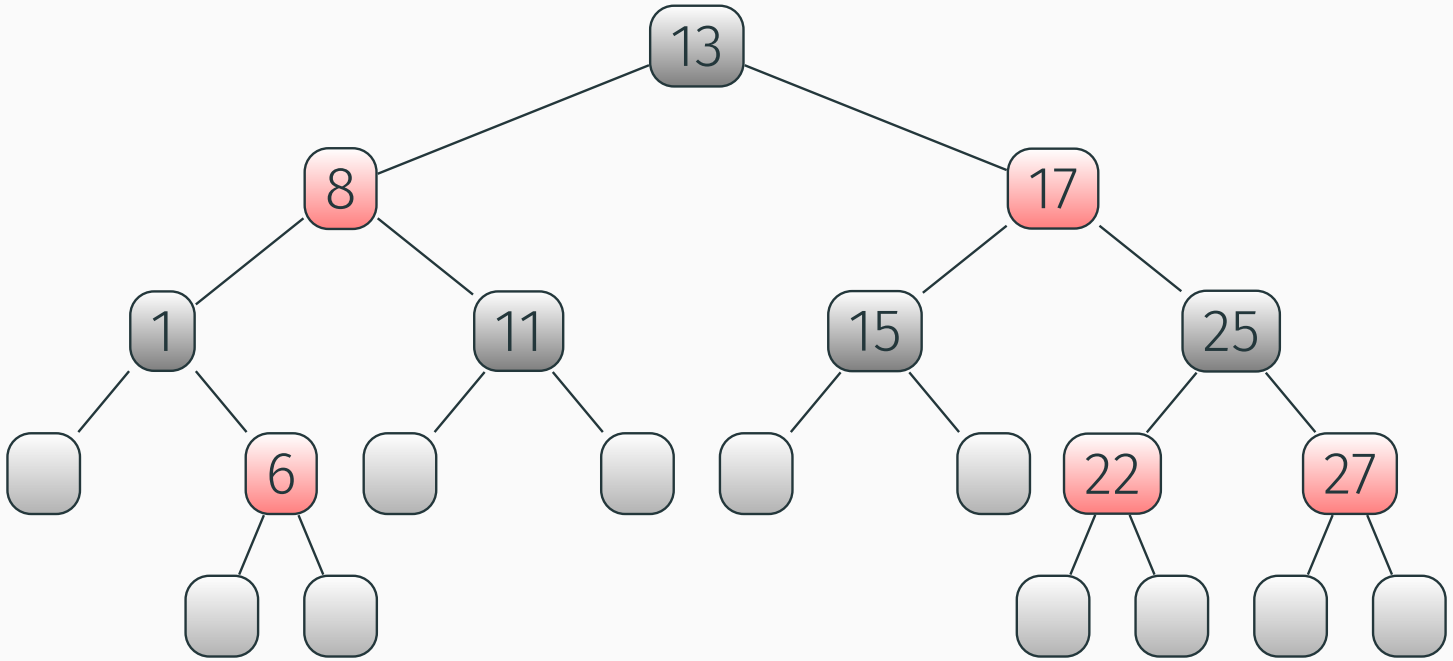





```
1  // leftRotate(Node x):
2  y = x.right
3
4  // build links between x and T1
5  x.right = y.left
6  if y.left is not null
7    y.left.parent = x
8
9  // build links between y and x.
       parent
10 y.parent = x.parent
11 if x is root
12   root = y
13 else if x is the left child
14   x.parent.left = y
15 else
16   x.parent.right = y
17
18 y.left = x
19 x.parent = y
```

Red-Black Tree

1. Every node has a color: either red or black.
2. Root and dummy leaves are colored black.
3. No red node is a parent of another red node.
4. Each root→leaf path in the tree has the same number of black nodes. (Black heights matter!!)



Height (Optional)

Claim: A RB-tree with n nodes height at most $2\log_2(n+1)$.

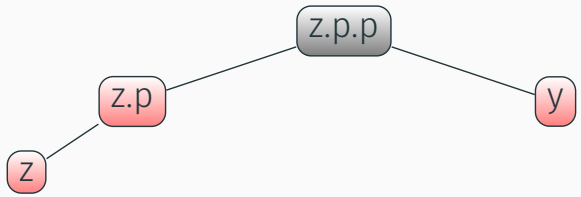
Proof: Fix any RB-tree, let h denote its height and n its size.

- On the root→leaf path of length h, there are $h+1 \geq h$ nodes.
- Clearly, the path has no two consecutive red nodes (property 3). So the number of black nodes is $\geq \frac{h}{2}$.
- So property 4 assures that all root→leaf paths have at least $\frac{h}{2}$ black nodes.
- The first $\frac{h}{2}$ layers in the tree are full. Thus

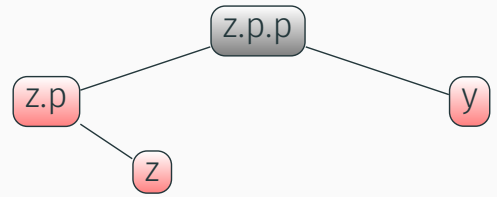
$$n \geq 1+2+4+8+\ldots+2^{\frac{h}{2}-1} = 2^{\frac{h}{2}}-1 \quad \Rightarrow \quad \log_2(n+1) \geq \frac{h}{2} \quad \square$$

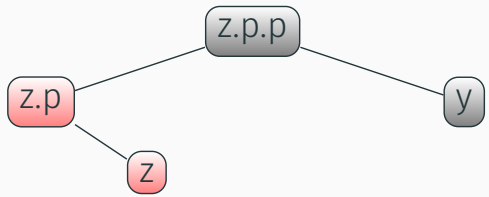
Insert

- A new node is colored in red.
- If it's the very first node (T's root) — just color it black. Done.
- Only property 3 can be violated: so if the new leaf is z, then its parent $z.p$ is red and we need to fix it up.
- Due to property 2, the red parent $z.p$ has to have a parent $z.p.p$. Due to property 3, $z.p.p$ must be black.
- So now it comes to z's uncle y (the non-$z.p$ child of $z.p.p$) — 3 cases:

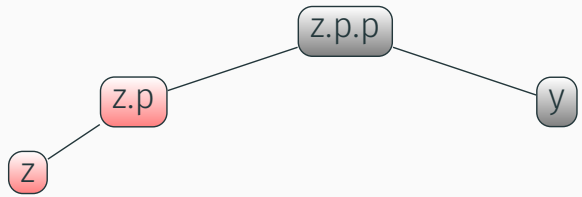


Case 1





Case 2

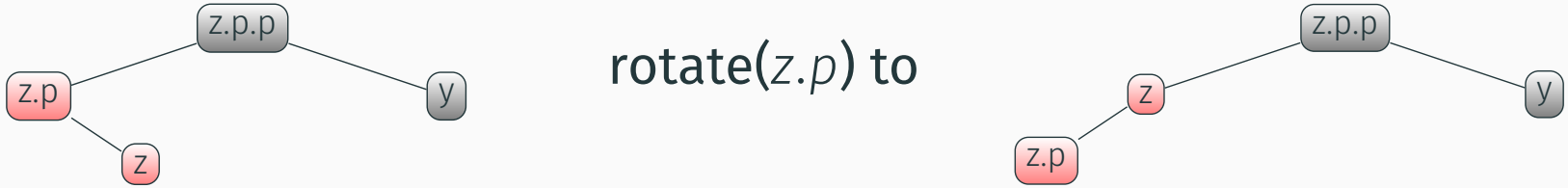


Case 3

Insert

Case 1: color $z.p$ and y black and color $z.p.p$ red, and recurse up the fix-up.

Case 2: rotate to Case 3



Case 3: rotate($z.p.p$) to make $z.p$ the parent, then flip colors of $z.p.p$ and $z.p$.

Delete

Eventually, a node with at least one child missing is deleted.

- If this node is red — no violation occurs.
- If this node is black, we start the fix-up with the deleted node's child x (could be a black dummy leaf)
  - If x is red — color it black.
  - If x is the root of the tree — just color it black.
  - O/w (i.e., x is black & not a root), we're looking into x's sibling w

Case 1. w is red;

Case 2. w is black and it has only black children;

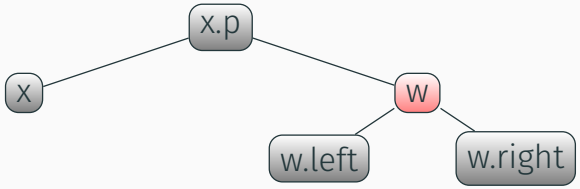
Case 3. w is black and its child in the reverse direction is red;

Case 4. w is black and its child in the same direction is red.

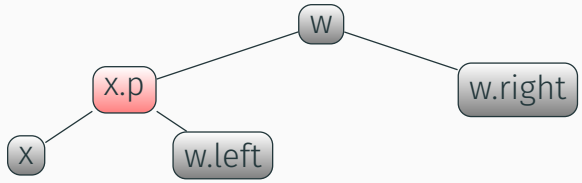
Delete: a black node was deleted

Child x is black & not a root, we're looking into x's sibling w

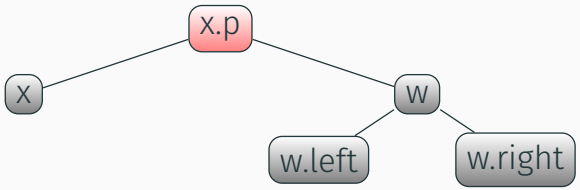
Case 1: w is red; Rotate so that x's sibling is black (check cases 2, 3 or 4)

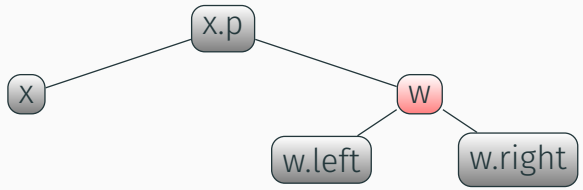


rotate to

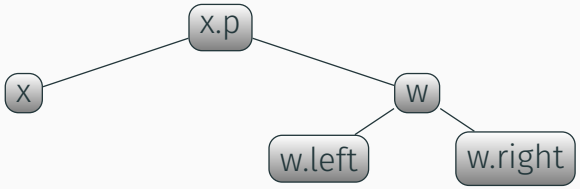


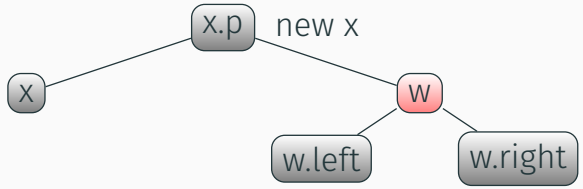
Case 2: w is black and it has only black children;





If w's (and x's) parent is red, color w's parent black and this compensates for the black node we removed;



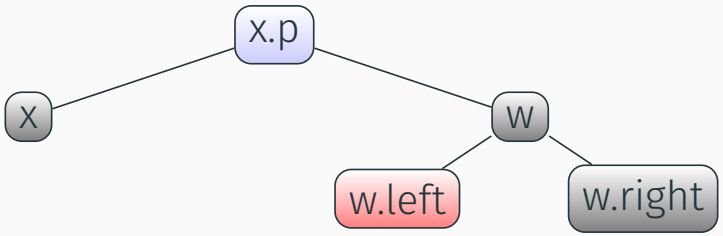


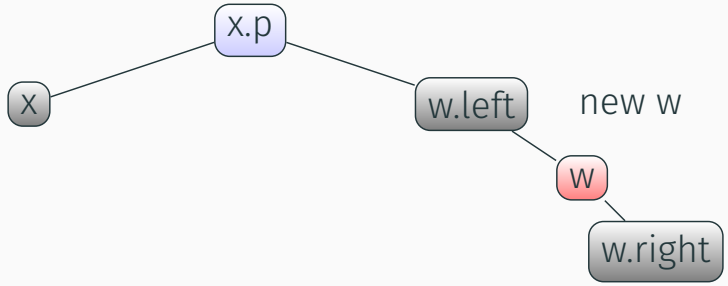
If w's (and x's) parent is black, recurse the fix-up on it.

Delete: a black node was deleted

Child x is black & not a root, we're looking into x's sibling w

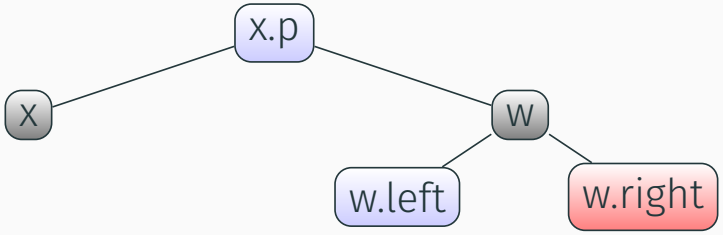
Case 3: w is black and its child in the reverse direction is red;

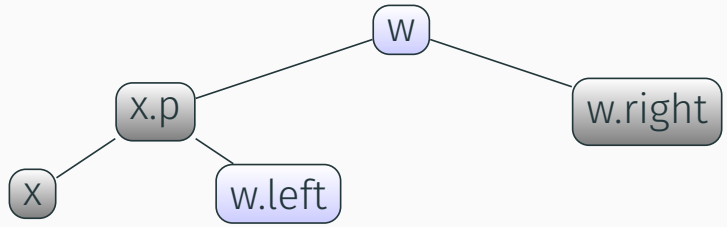




Rotate on w and flip w and its (new) parent color to make it case 4.

Case 4: w is black and its child in the same direction is red.





Give w the color of w's parent; rotate w's parent (so w is the new root of this subtree); color both children of w black

Thank you!

Questions?