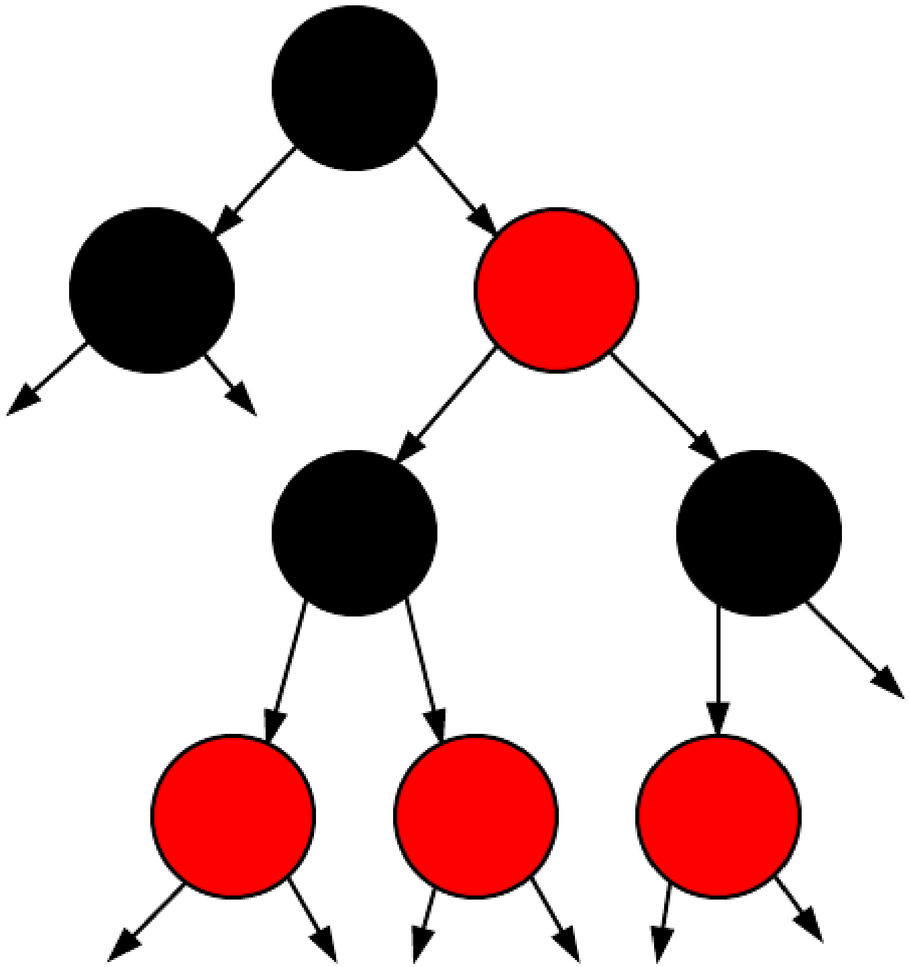
Red-Black Tree



A red-black tree is a kind of self-balancing binary search tree.

Each node stores an extra bit representing "color" (**red** or **black**),

These colours are used to ensure that the tree remains balanced during insertions and deletions.

Why **red** and **black**?

- The color “red” was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC.

- Another response from Guibas states that it was because of the red and black pens available to them to draw the trees.

The path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.

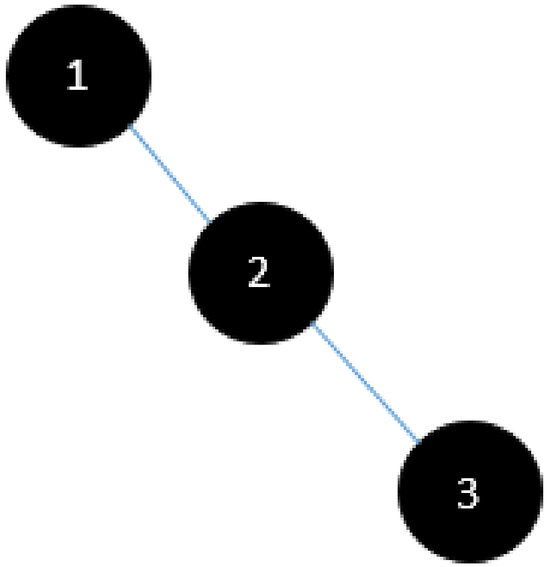
Requirements:

1. Each node is either **red** or **black**.
2. All **NIL** nodes are considered **black**.
3. A red node does not have a red child.
4. Every path from a given node to any of its descendant **NIL** nodes goes through the same number of black nodes.

- The root of the tree is always black. *(Optional)*

How does a Red-Black Tree ensure balance?

Try to recolour.



"A red-black tree with n internal (non-Nil) nodes has height at most 2log(n+1)."

— Introduction to Algorithms, Lemma 13.1

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause **more rotations** during insertion and deletion.

So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred.

And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Applications:

- Most of the self-balancing BST library functions like **map** and **set** in C++ (OR TreeSet and TreeMap in Java) use Red-Black Tree.
- It is used to implement CPU Scheduling Linux.
- Moreover, MySQL also uses the Red-Black tree for indexes on tables.
- Besides they are used in the K-mean clustering algorithm for reducing time complexity.

Terminology

- The **black depth** of a node is defined as the number of black nodes from the root to that node (i.e. the number of black ancestors).
- The **black height** of a red-black tree is the number of black nodes in any path from the root to the leaves.

- Violation of requirement 3 is called **red-violation**.
- Violation of requirement 4 is called **black-violation**.

Insertion

In the Red-Black tree, we use two tools to do the balancing.

1. Recoloring (Repairing)
2. Rotation

Let's denote:

- N - current node
- P - parent node
- G - grandparent node
- U - uncle node

The rebalancing loop has the following invariant:

- N is **red** at the beginning of each iteration.
- Requirement 3 is satisfied for all pairs N←P with the possible exception N←P when P is also **red** (a red-violation at N).
- All other properties (including requirement 4) are satisfied throughout the tree.

Case 1: If P is **black**

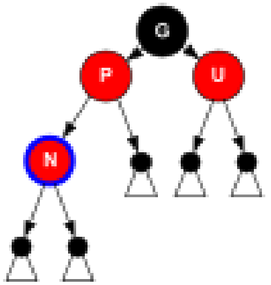
Requirement 3 holds!

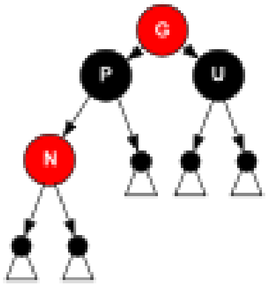
Case 2: If P and U are **red**

Both of them can be repainted black and the grandparent G becomes red for maintaining requirement 4.

Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed.

However, the grandparent G may now violate requirement 3, if it has a red parent. After relabeling G to N the loop invariant is fulfilled so that the rebalancing can be iterated on one black level (= 2 tree levels) higher.





Case 3: If N is the root

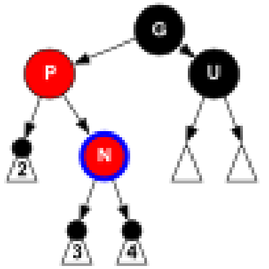
Case 2 has been executed for (h-1)/2 times and the total height of the tree has increased by 1, now being h. The current node N is the (red) root of the tree, and all RB-properties are satisfied.

Case 4: If P is the root and is **red**

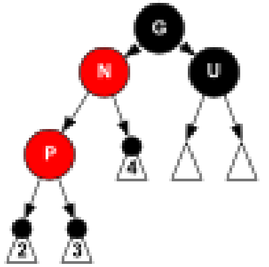
Because N is also red, requirement 3 is violated. But after switching P’s color the tree is in RB-shape. The black height of the tree increases by 1.

Case 5:
If P is **red**, U is **black,** N is inner grandchild

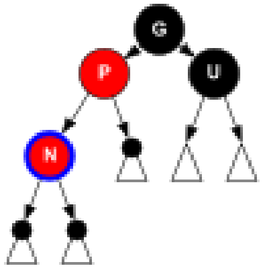
The ultimate goal is to rotate the parent node P to the grandparent position, but this will not work if N is an "inner" grandchild of G (i.e., if N is the left child of the right child of G or the right child of the left child of G).



A dir-rotation(RR/LL) at P switches the roles of the current node N and its parent P.



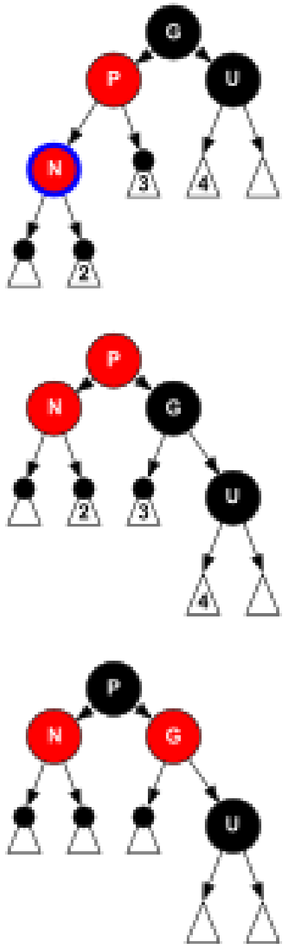
But both P and N are red, so requirement 4 is preserved. Requirement 3 is restored in case 6.



Case 6:

If P is **red**, U is **black,** N is outer grandchild

Now (1-dir)-rotate at G, putting P in place of G and making P the parent of N and G. G is black and its former child P is red, since requirement 3 was violated. After switching the colors of P and G the resulting tree satisfies requirement 3. Requirement 4 also remains satisfied, since all paths that went through the black G now go through the black P.



Insert diagram (from Wikipedia)

before				*case* →	*rota-tion*	*assig-nment*	*after*				→ *next*	*Δh*
P	**G**	**U**	*x*				**P**	**G**	**U**	*x*		
—				**I3**							→	
●				**I1**							→	
●	—			**I4**			●				→	
●	●	●		**I2**		**N:=G**	?				?	2
●	●	●	i	**I5**	**P↶N**	**N:=P**	●	●	●	o	**I6**	0
●	●	●	o	**I6**	**P↷G**		●	●	●		→	

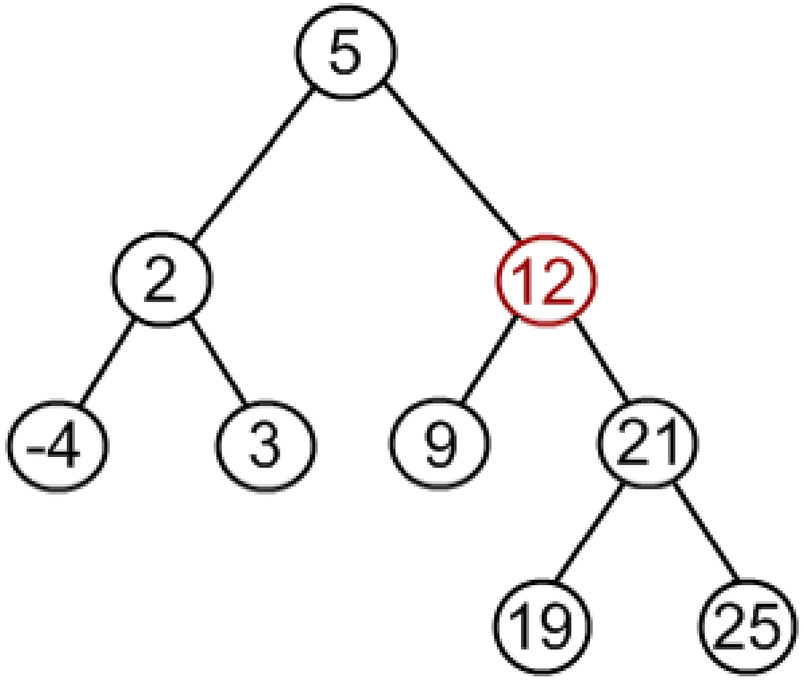
Insertion: This synopsis shows in its *before* column, that all possible cases[32] of constellations are covered.

Removal

Let's denote:

- **N** - current node
- **P** - parent node of N
- **S** - sibling node of N
- **C** - close nephew of N (S’s child in the same direction as N)
- **D** - distant nephew of N

Deletion from BST



RB tree removal: simple cases

- If N is the root that does not have a non-NIL child, it is replaced by a NIL node, after which the tree is empty—and in RB-shape.

- If N has two non-NIL children...
  - Is this case possible after replacement?

- If N has exactly one non-NIL child, it must be a red child (**black**-violation)

- If N is a **red** node, it cannot have a non-NIL child (**red**-violation)

- If N is a **black** node, it may have a red child or no non-NIL child at all.
  - If N has a red child, it is simply replaced with this child after painting the latter black.
  - If N has only NIL children..... !!!!!!


The complex case is
when N is not the root,
colored **black** and has
only NIL children.

In the first iteration,
N is replaced by NIL.

The rebalancing loop has the following invariant:

- At the beginning of each iteration, the **black height** of N equals the iteration number minus one, which means that in the first iteration it is zero and that N is a true black node in higher iterations.

- The number of **black** nodes on the paths through N is one less than before the deletion, whereas it is unchanged on all other paths so that there is a **black-violation** at P if other paths exist.

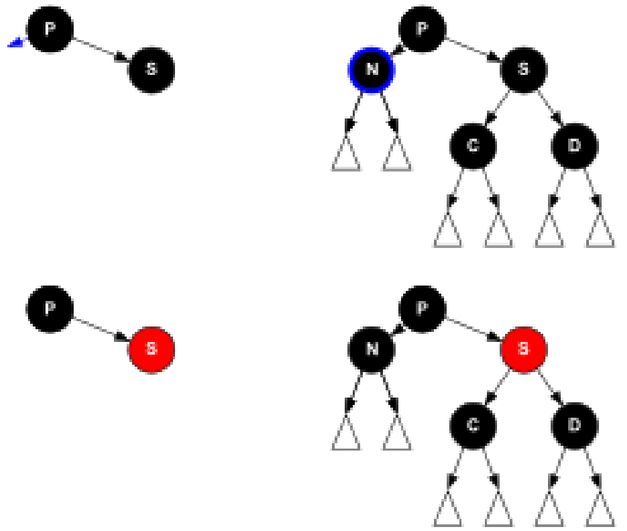
- All other properties (including requirement 3) are satisfied throughout the tree.

Case 1: P, S, S’s children are **black**

After painting S **red** all paths passing through S, which are precisely those paths not passing through N, have one less **black** node.

Now all paths in the subtree rooted by P have the same number of black nodes, but <u>one fewer than the paths that do not pass through P</u>, so we still have **black-violation**.

After relabeling P to N the loop invariant is fulfilled so that the rebalancing can be iterated on one black level (= 1 tree level) higher.



Case 2: N is the new root

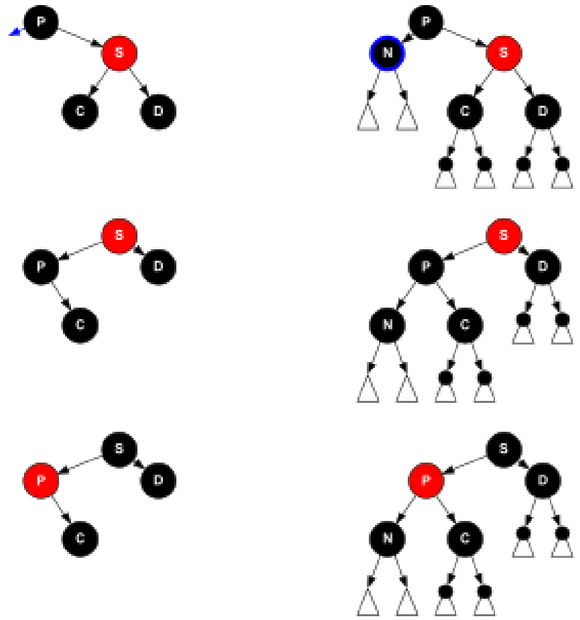
RB-properties are preserved

Case 3: S is **red**, P, C, D are **black**

A dir-rotation at P turns S into N's grandparent.

Then after reversing the colors of P and S, the path through N is still short one black node.

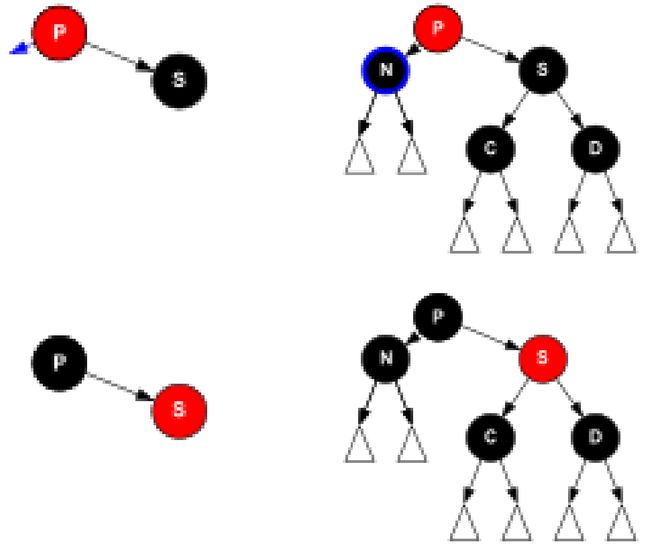
But N has a **red** parent P and a **black** sibling S, so the transformations in cases 4, 5, or 6 are able to restore the RB-shape.



Case 4: P is **red**, S, S's children are **black**

The sibling S and S’s children are black, but P is red.

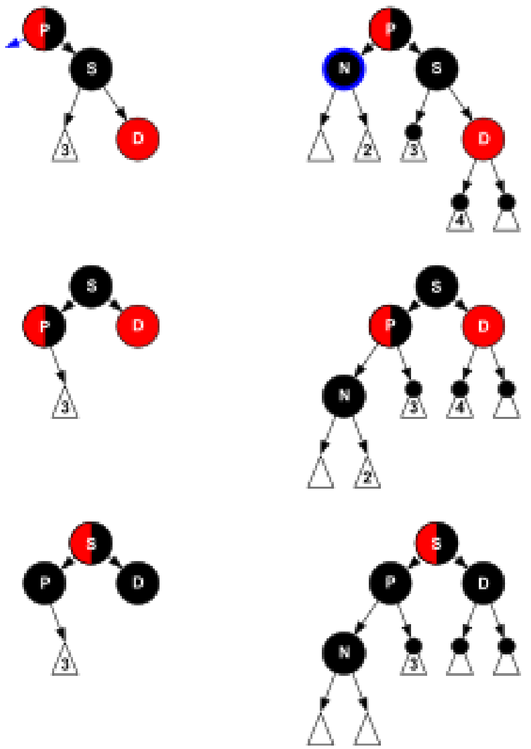
Exchanging the colors of S and P does not affect the number of black nodes on paths going through S, but it does add one to the number of black nodes on paths going through N, making up for the deleted black node on those paths.



Case 6: D is **red**, S is **black**

After a dir-rotation at P the sibling S becomes the parent of P and S’s distant child D.

The colors of P and S are exchanged, and D is made **black**. The subtree still has the same color at its root.

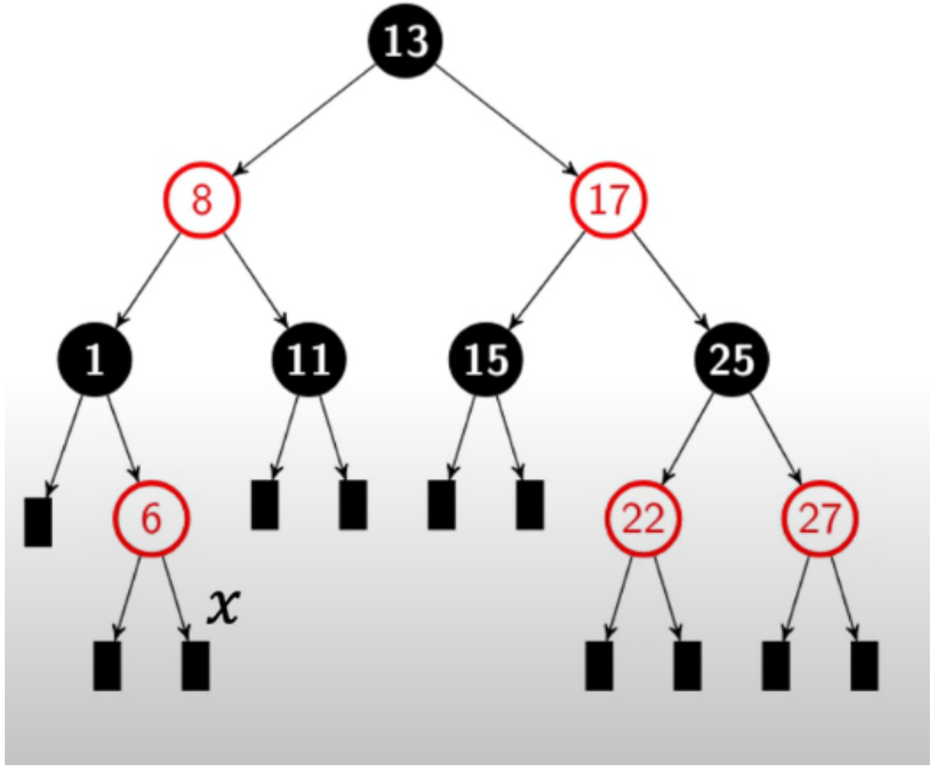


Delete diagram (from Wikipedia)

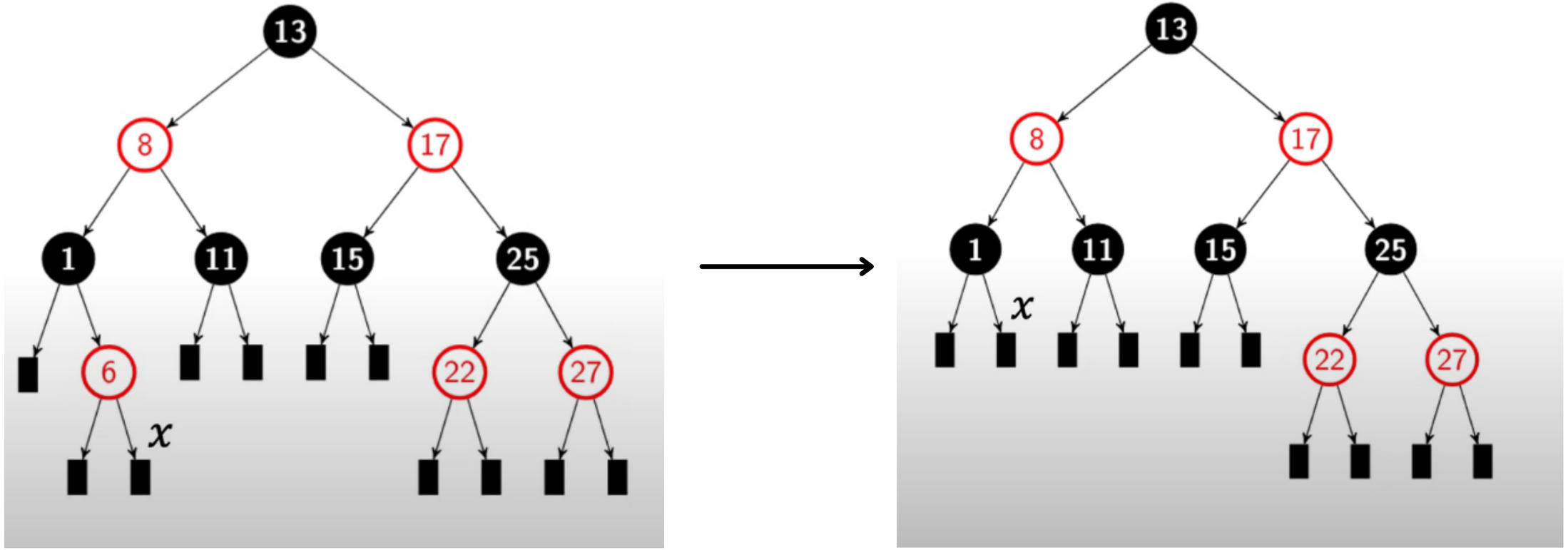
before				*case* →	*rota-tion*	*assig-nment*	*after*				→ *next*	*Δh*
P	**C**	**S**	**D**				**P**	**C**	**S**	**D**		
—				**D2**							→	
●	●	●	●	**D3**	**P↶S**	**N:=N**	●	?	●	?	?	0
●	●	●	●	**D1**		**N:=P**	?				?	1
●	●	●	●	**D4**			●	●	●	●	→	
◑	●	●	●	**D5**	**C↷S**	**N:=N**	◑		●	●	**D6**	0
◑		●	●	**D6**	**P↶S**		●		◑	●	→	

Deletion: This synopsis shows in its *before* column, that all possible cases[32] of color constellations are covered.

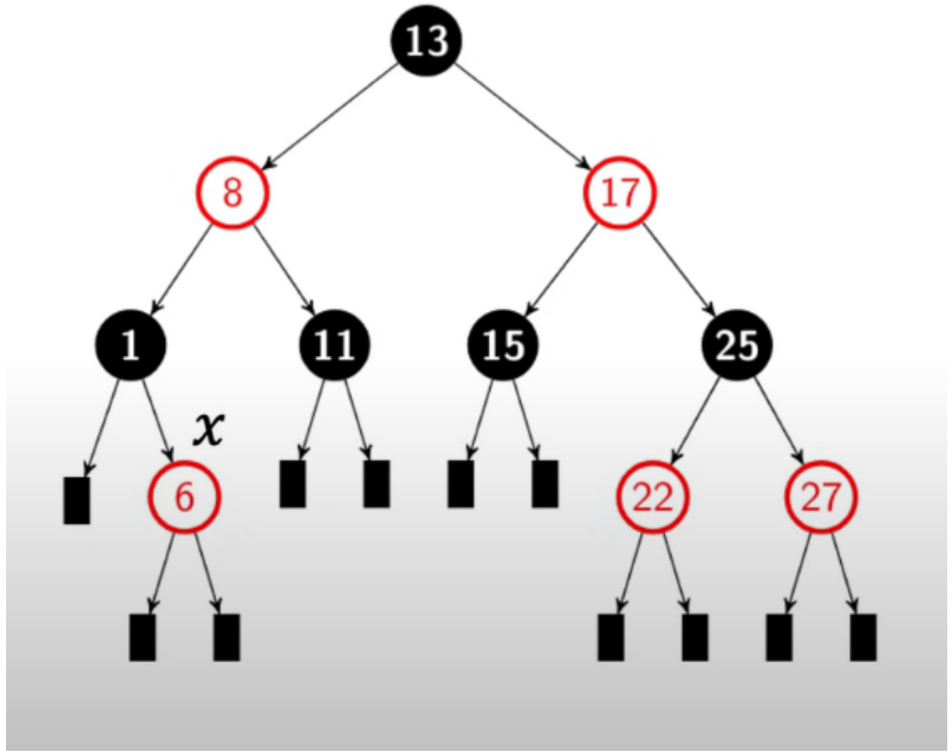
Example 1 (del 6)



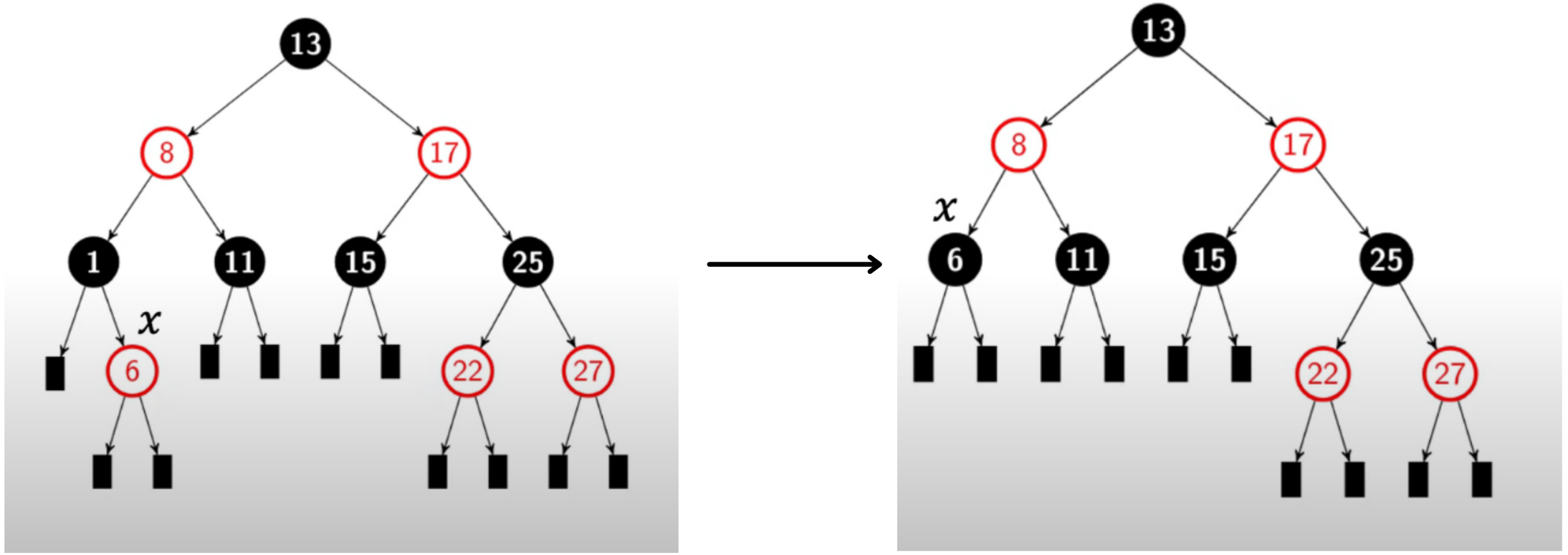
Example 1 (del 6)



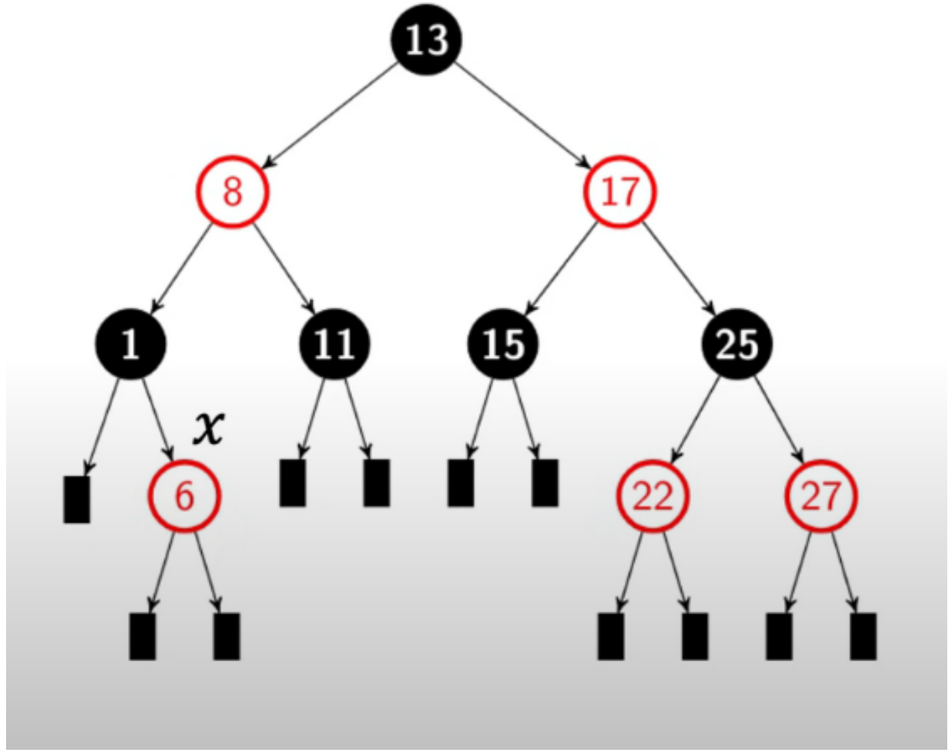
Example 2 (del 1)



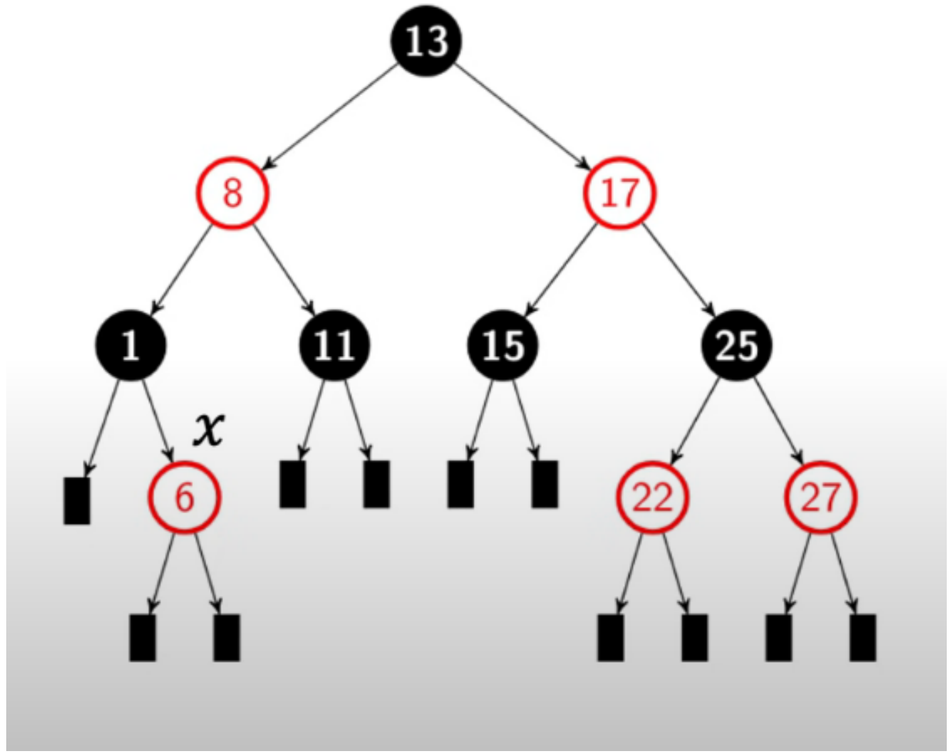
Example 2 (del 1)

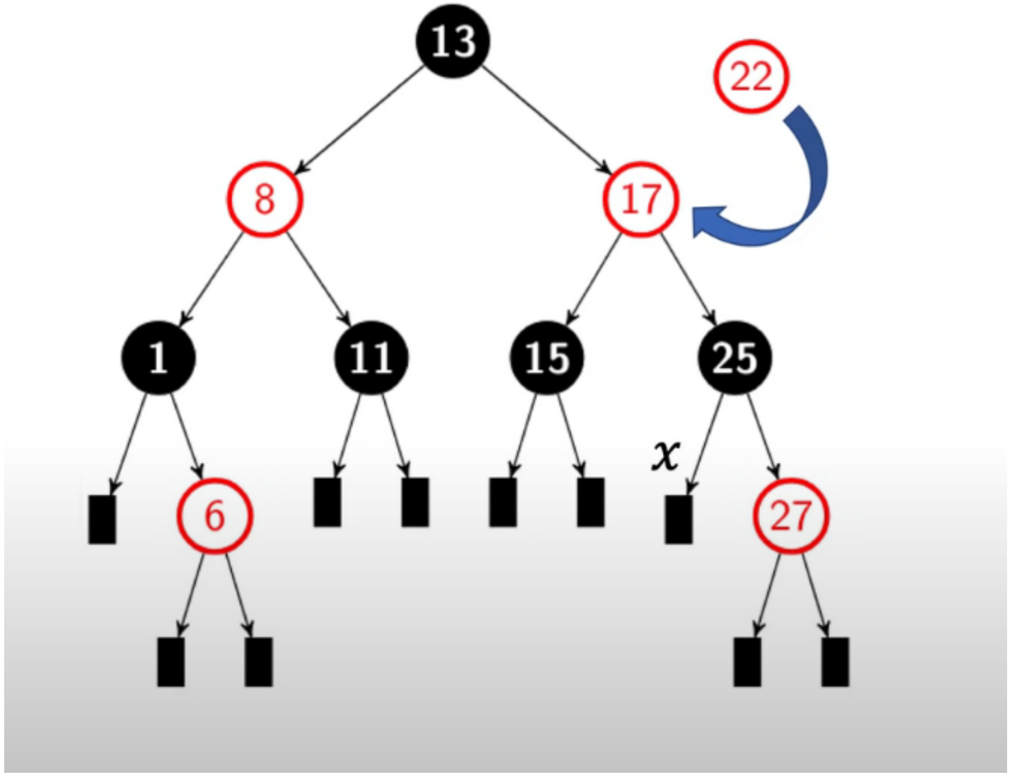


Example 3 (del 17)

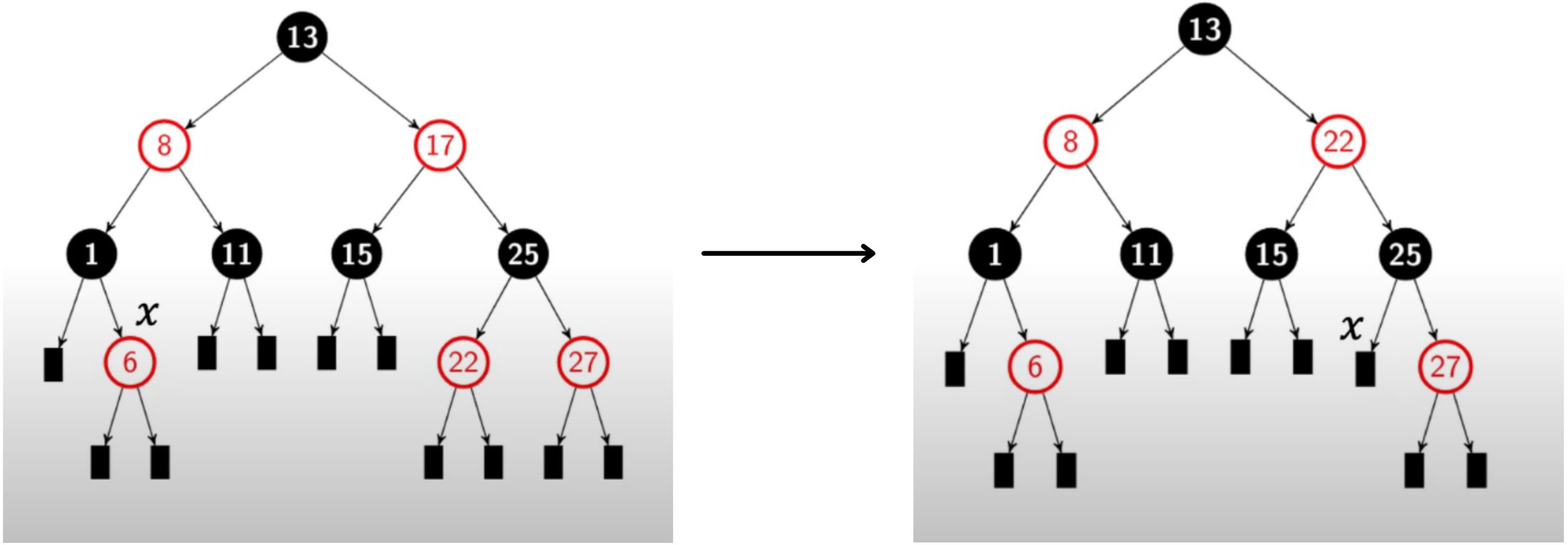


Example 3 (del 17)

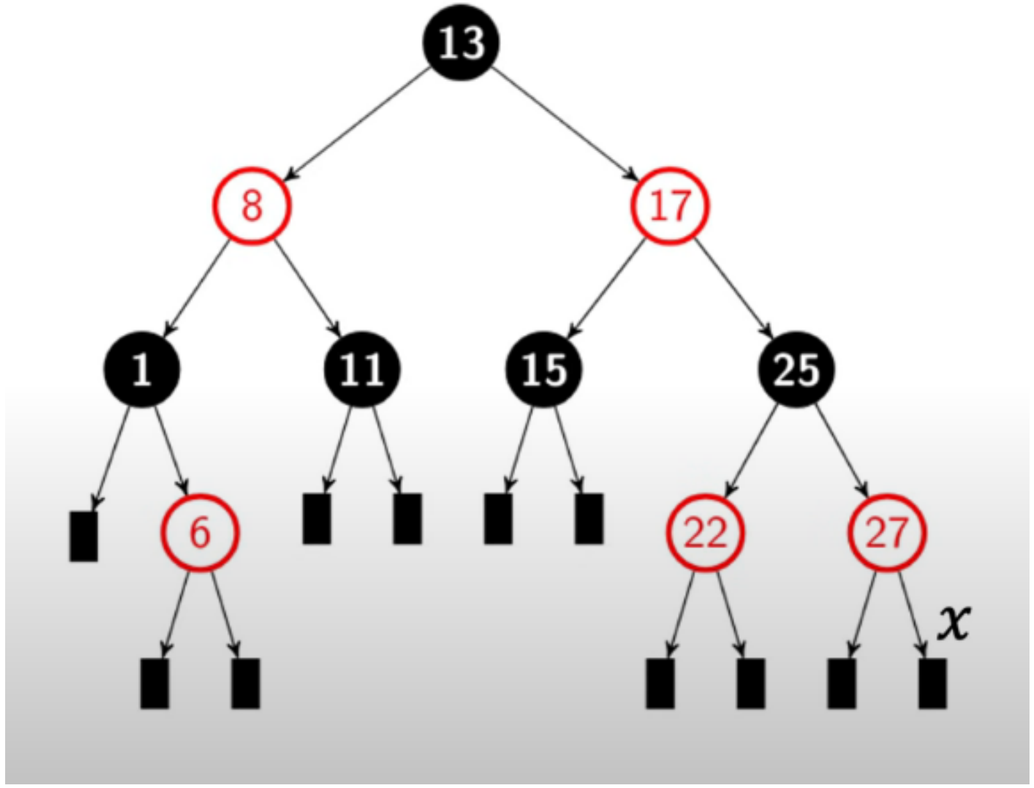




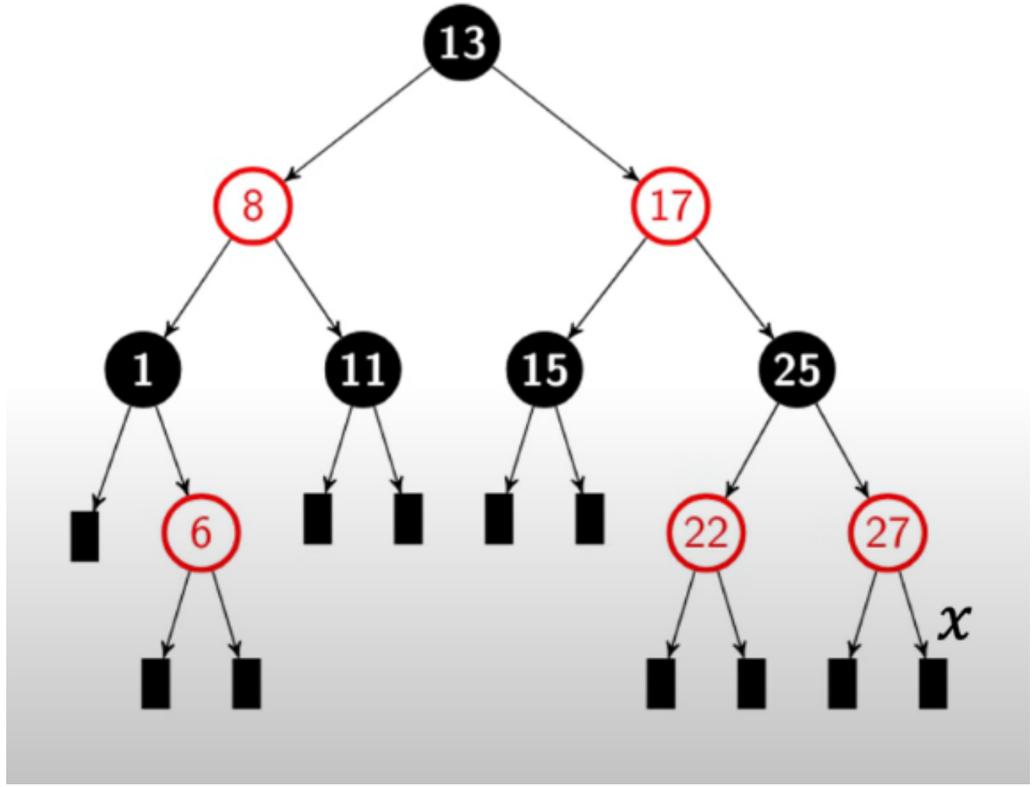
Example 3 (del 17)

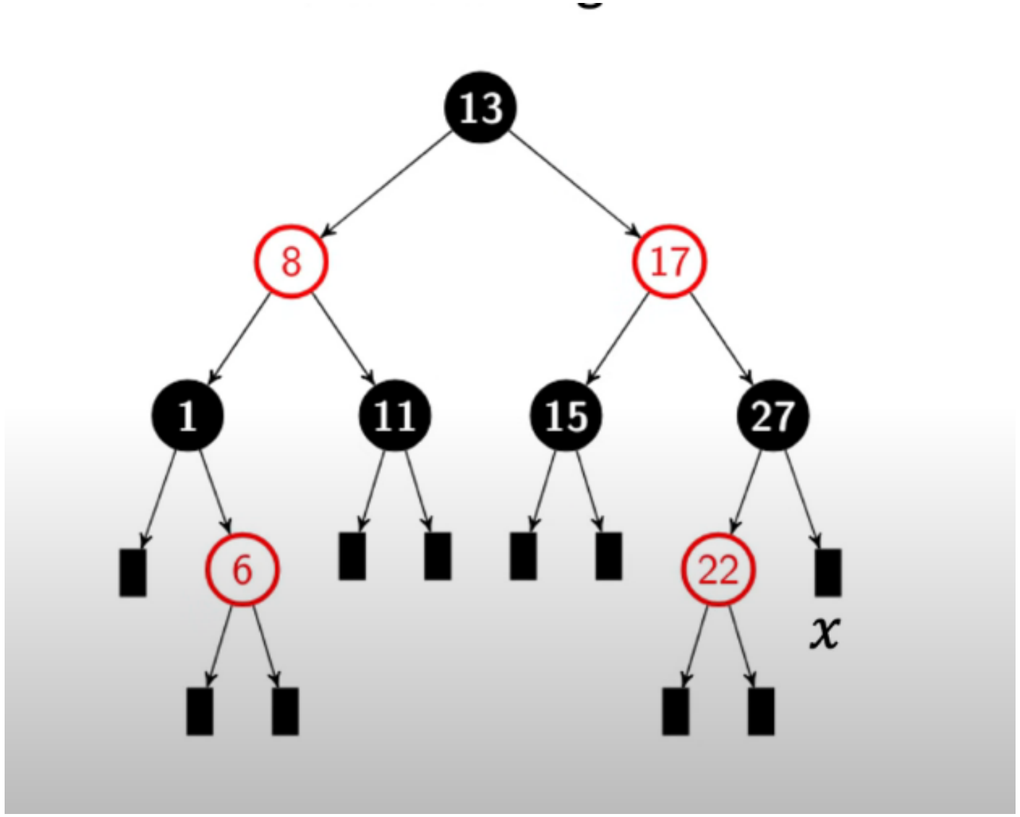


Example 4 (del 25)

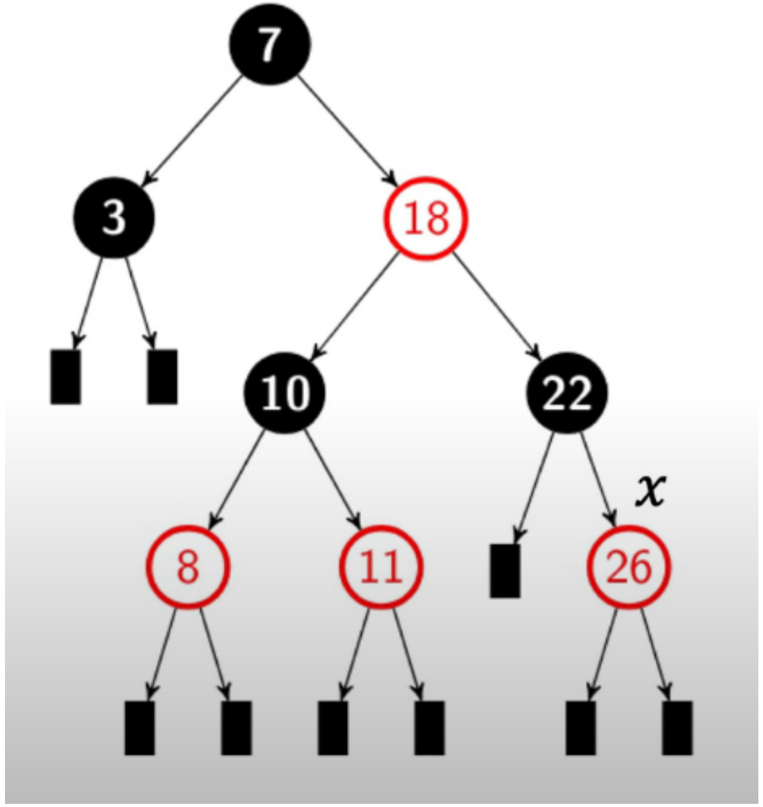


Example 4 (del 25)

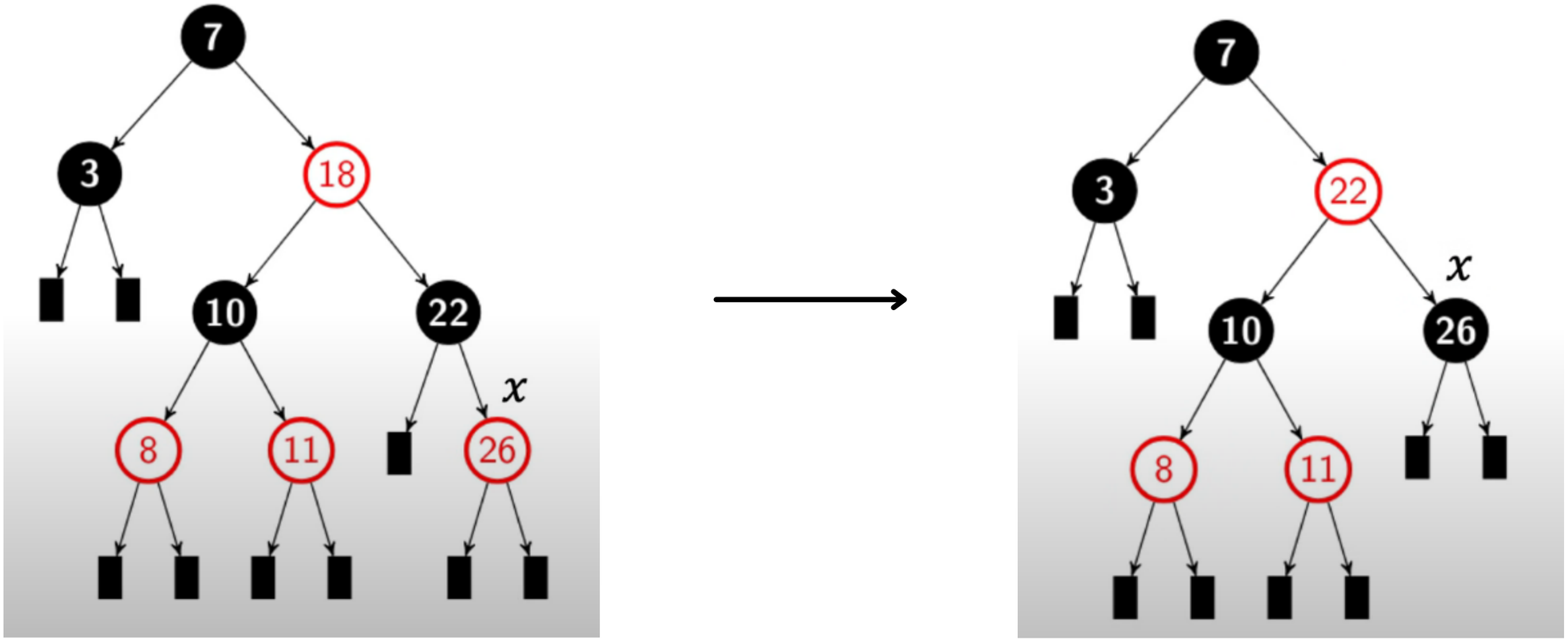




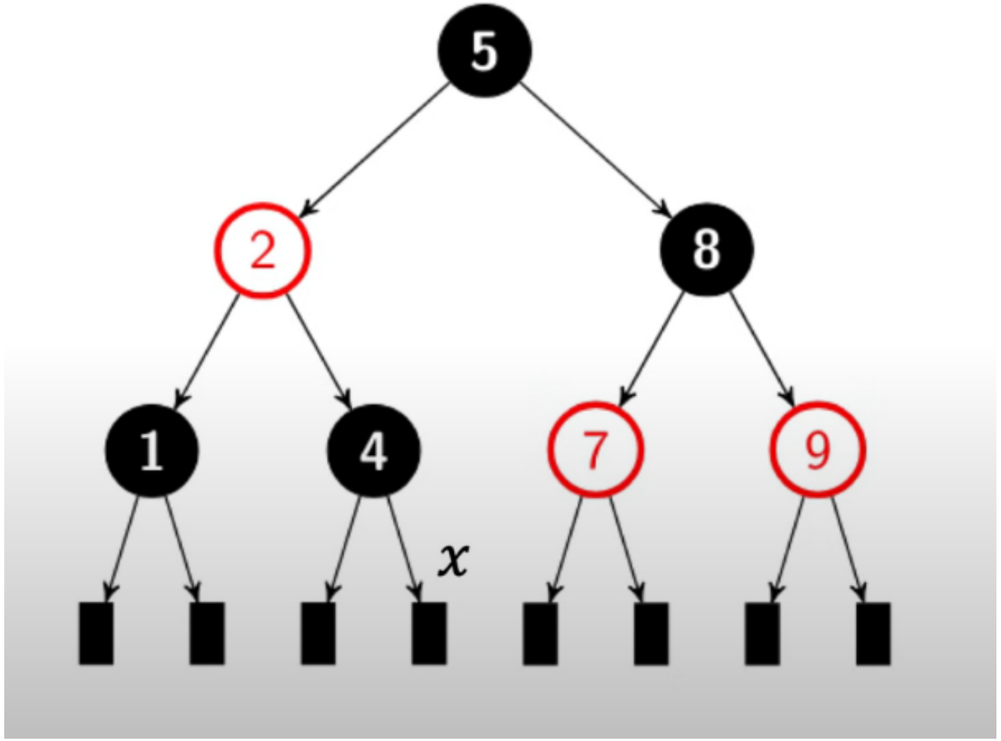
Example 5 (del 18)



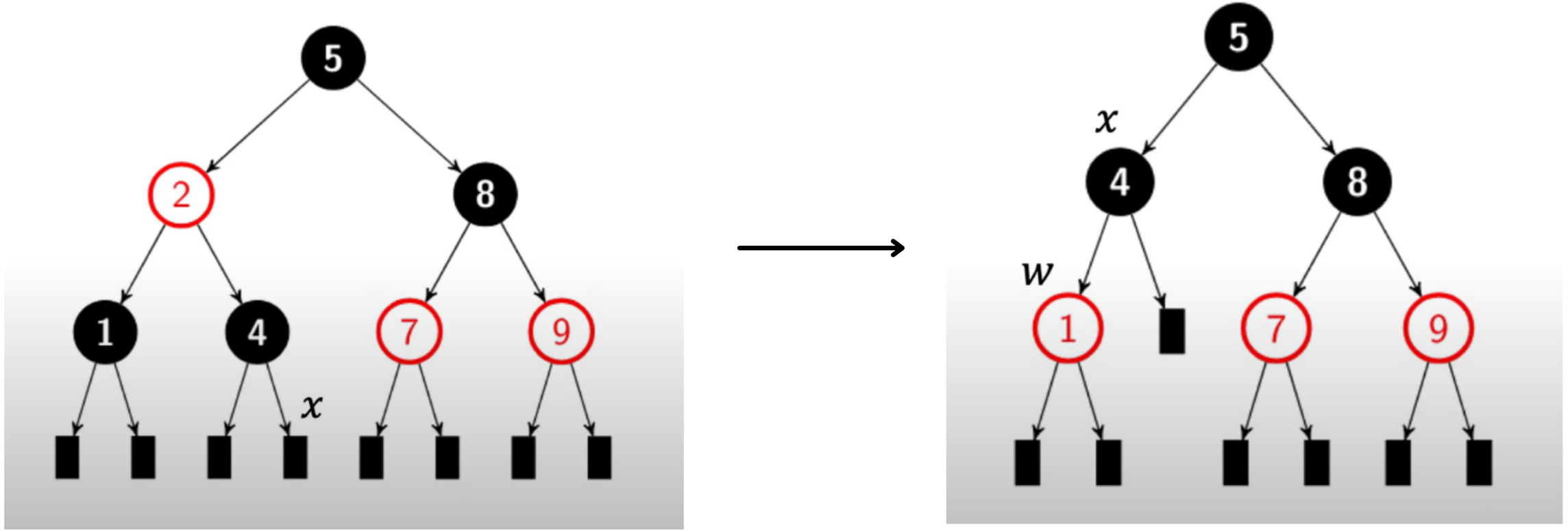
Example 5 (del 18)



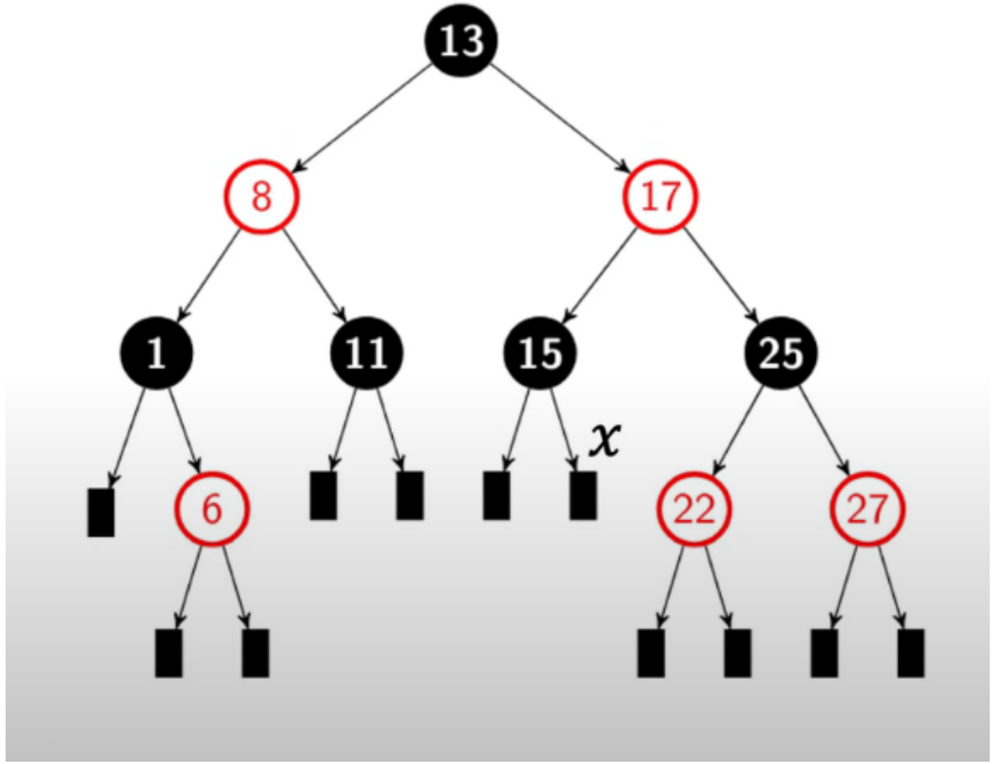
Example 6 (del 2)



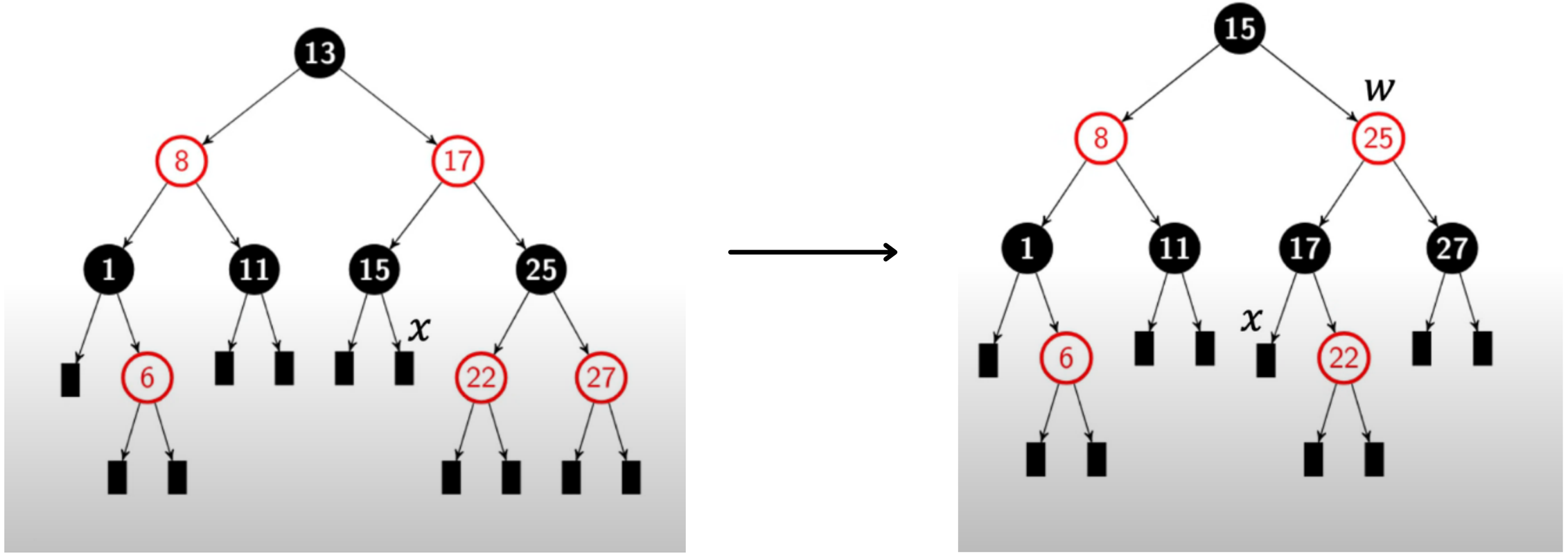
Example 6 (del 2)



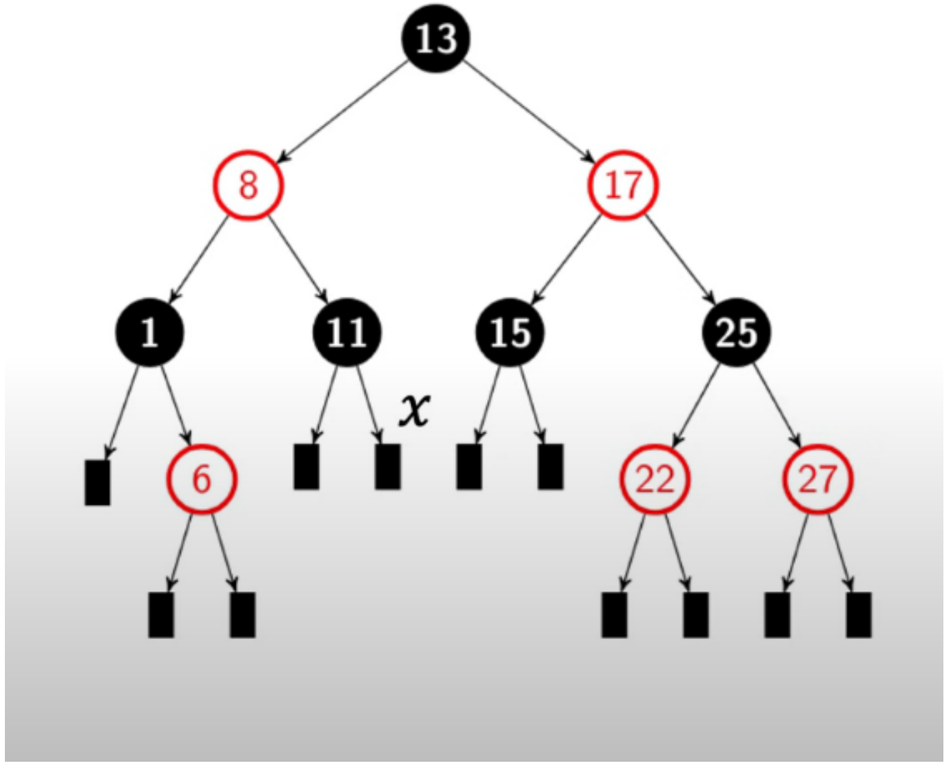
Example 7 (del 13)



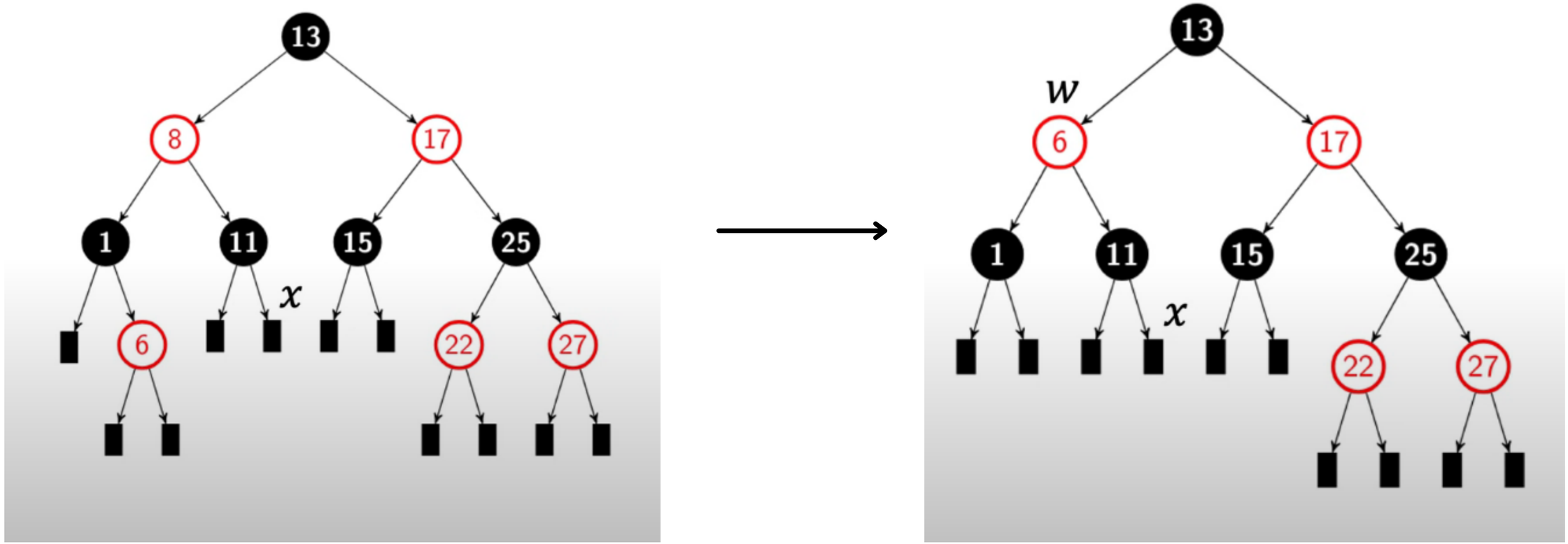
Example 7 (del 13)



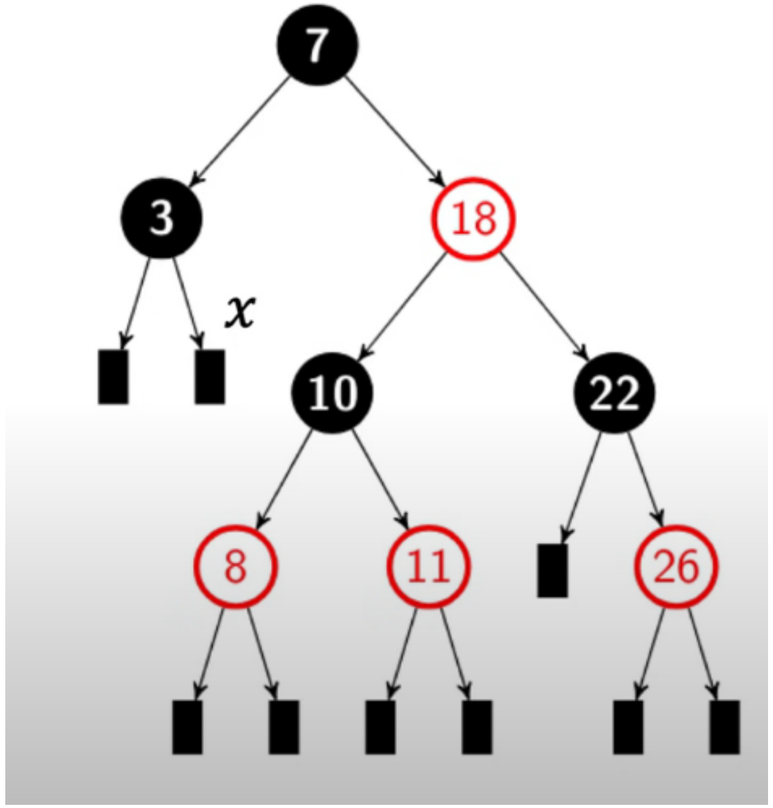
Example 8 (del 8)



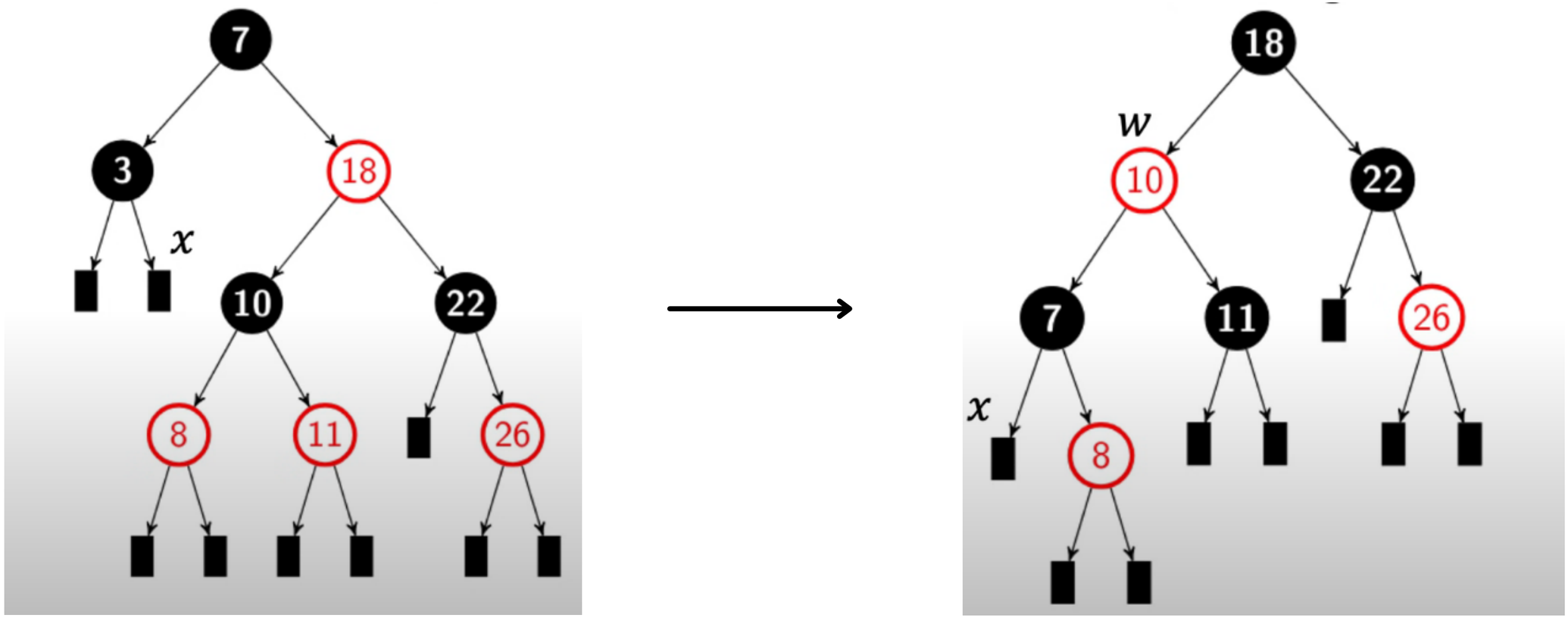
Example 8 (del 8)



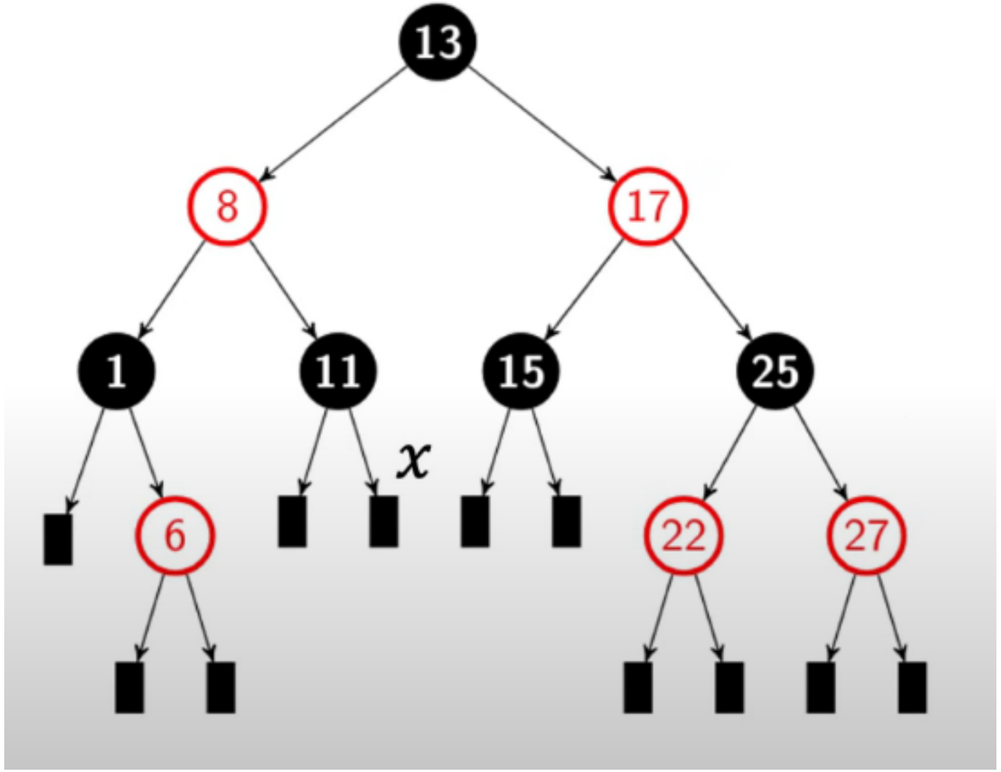
Example 9 (del 3)



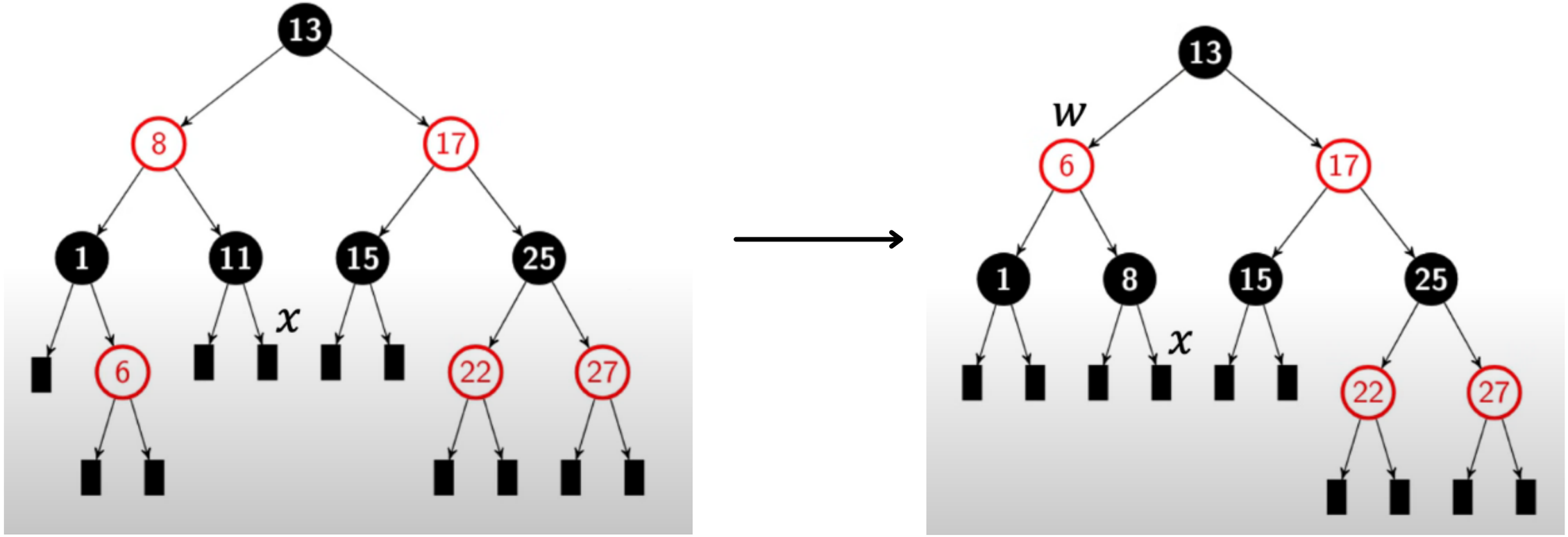
Example 9 (del 3)

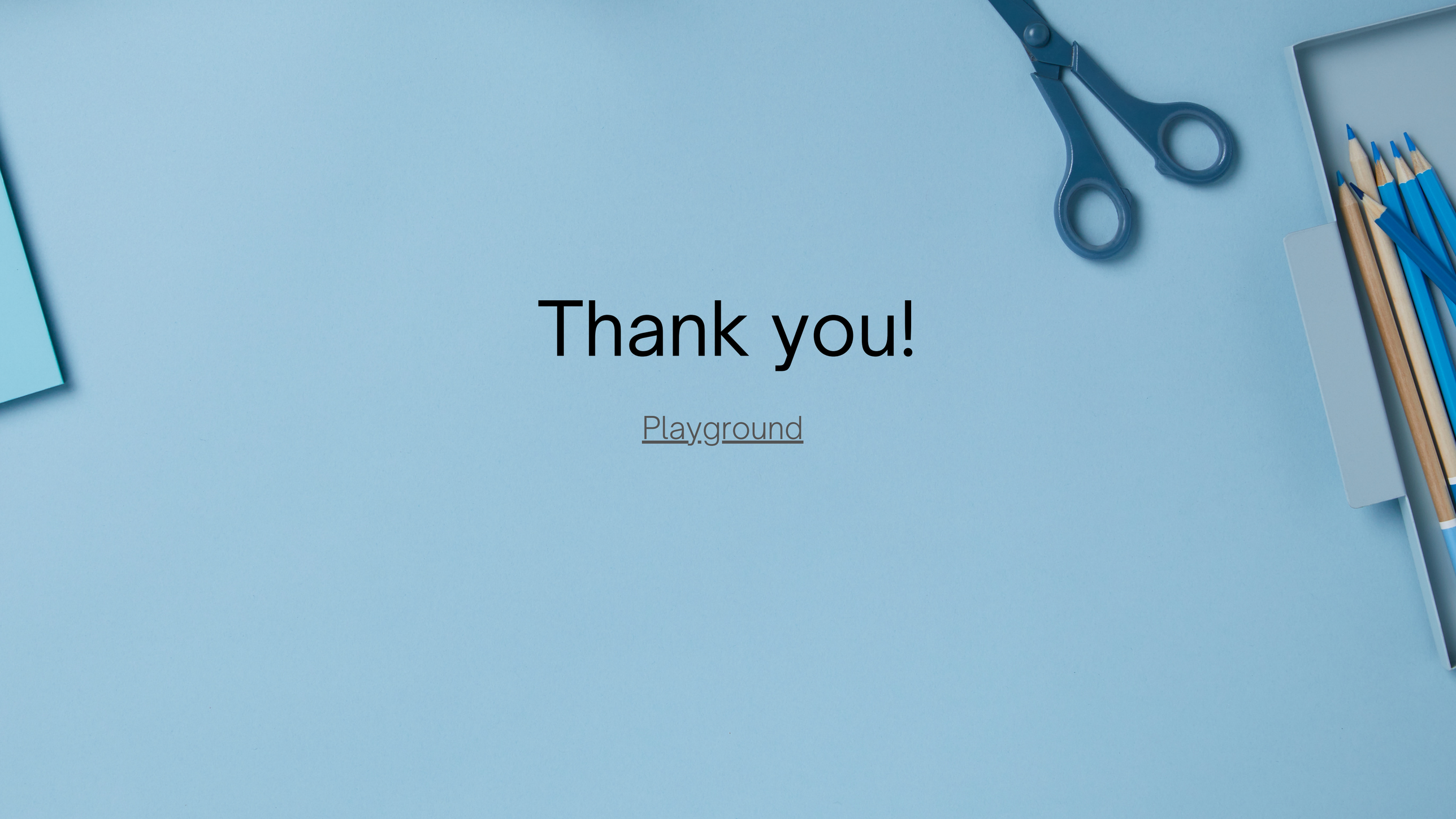


Example 10 (del 11)



Example 10 (del 11)





Thank you!

Playground