Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

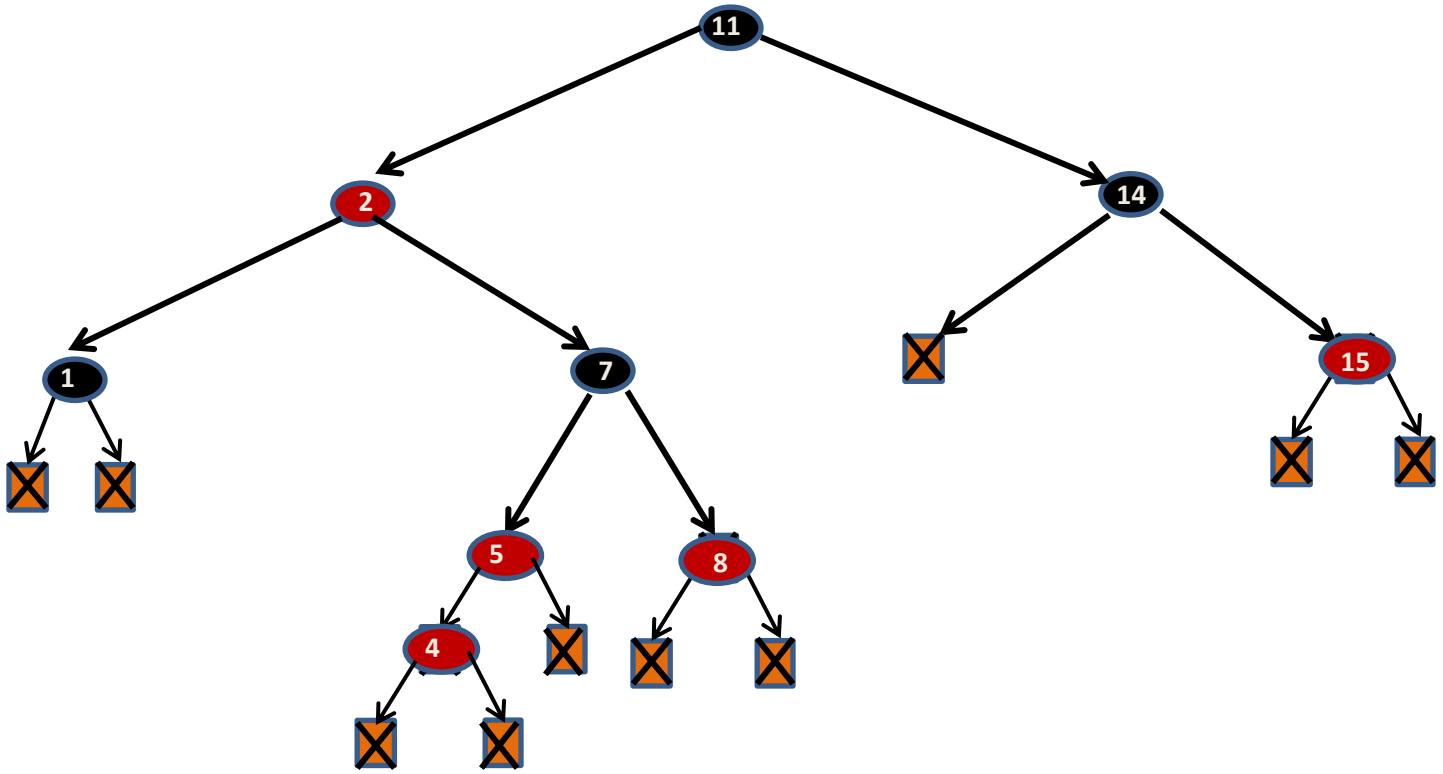
Lecture 20:

- **Solving some practice sheet problems**
- **Overview of the 2nd half of course.**

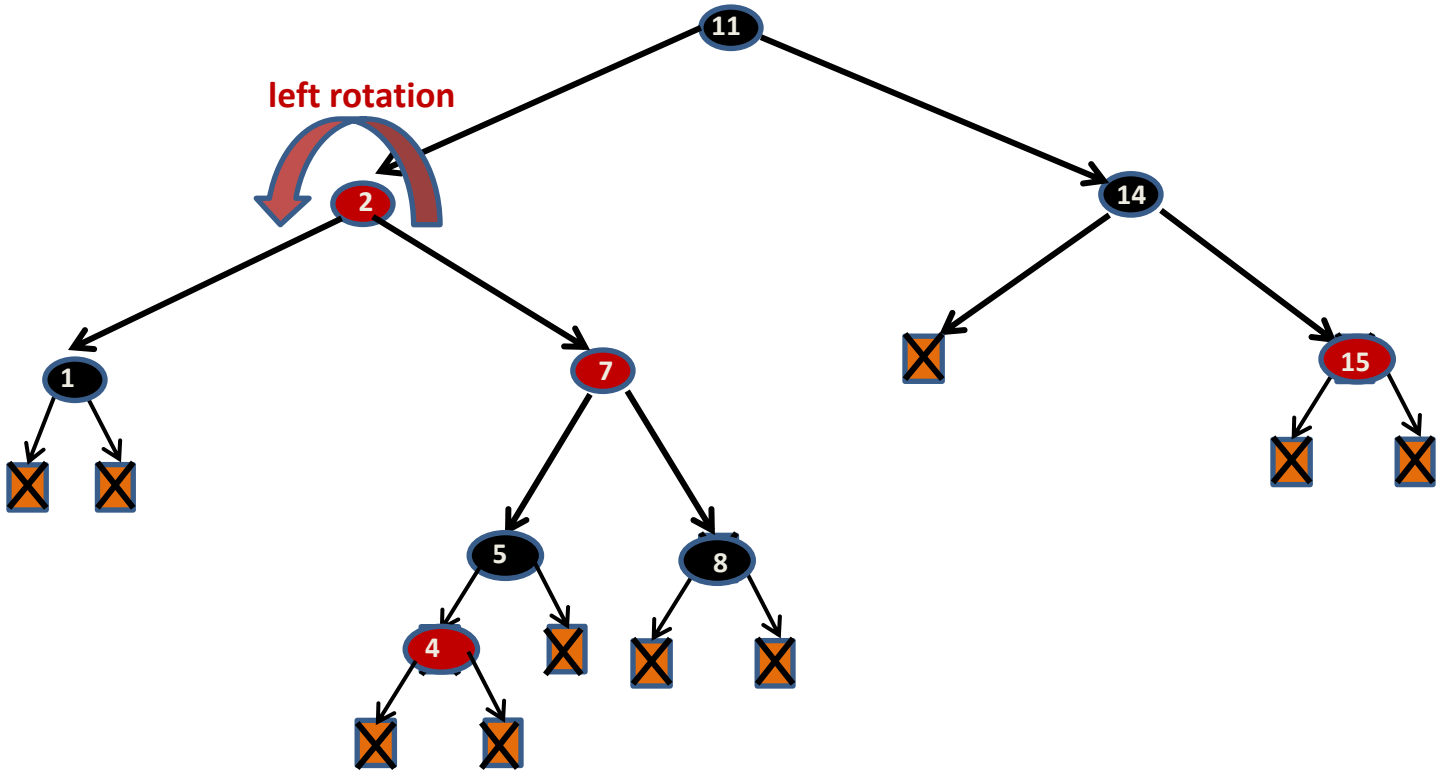
Practice problems

Sheet 4

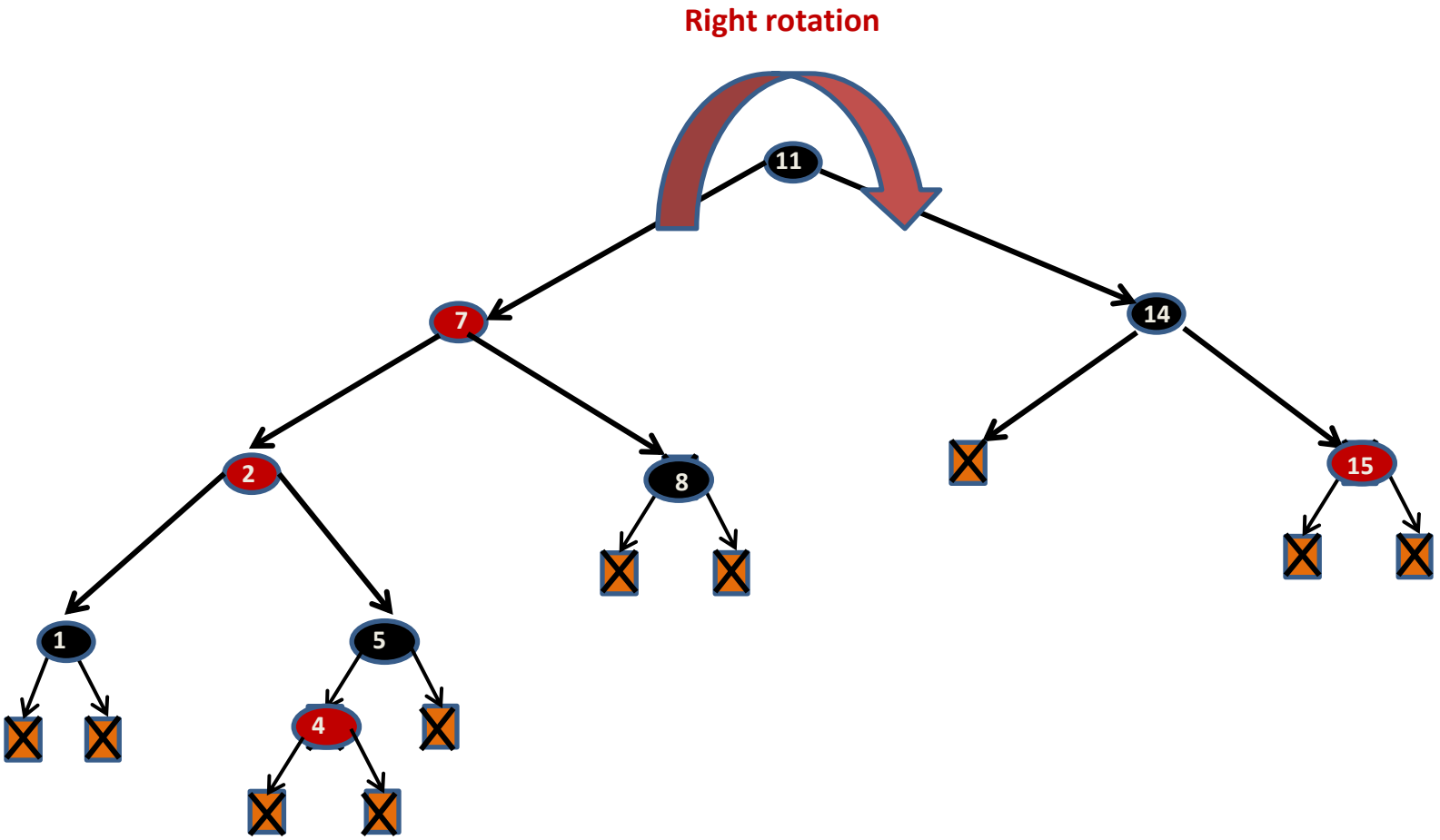
How to insert 4 ?



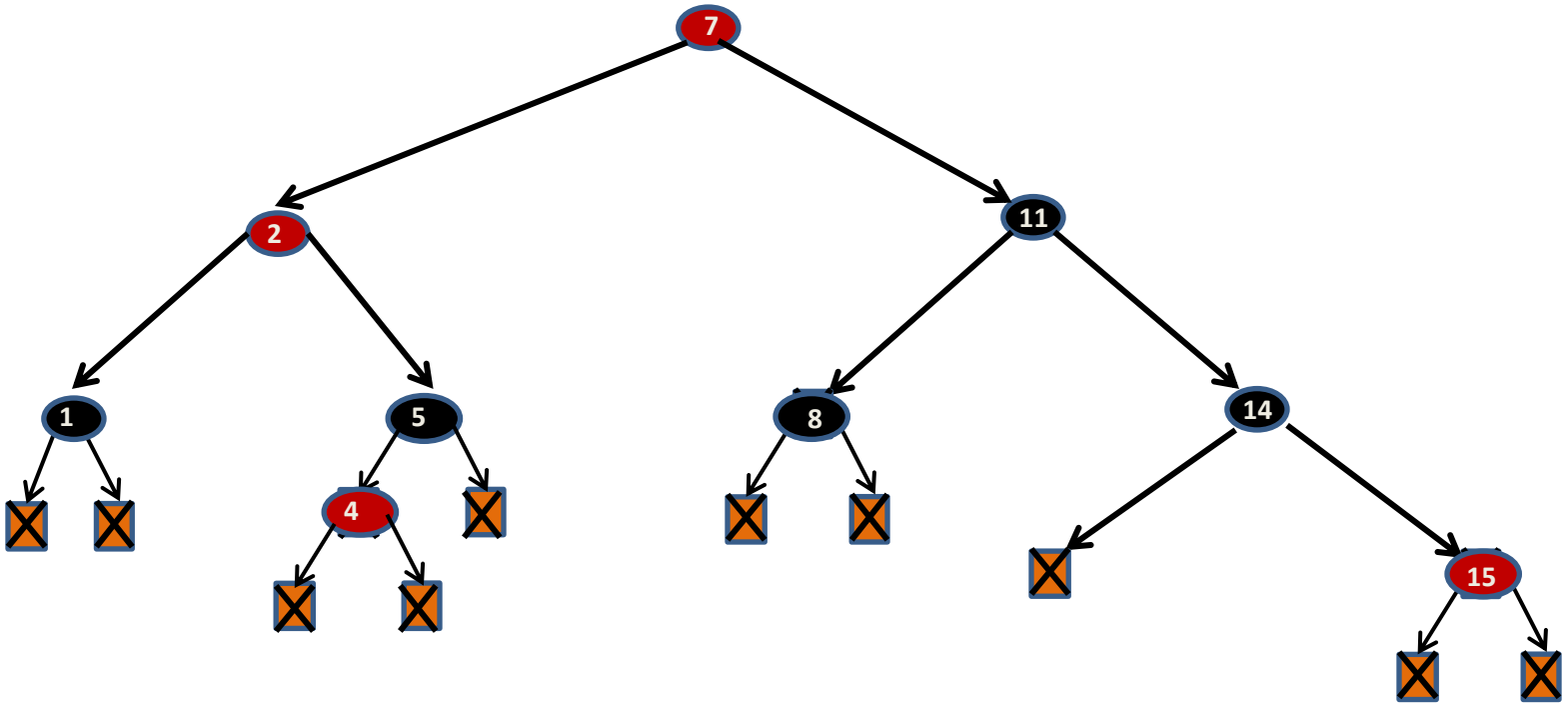
How to insert 4 ?



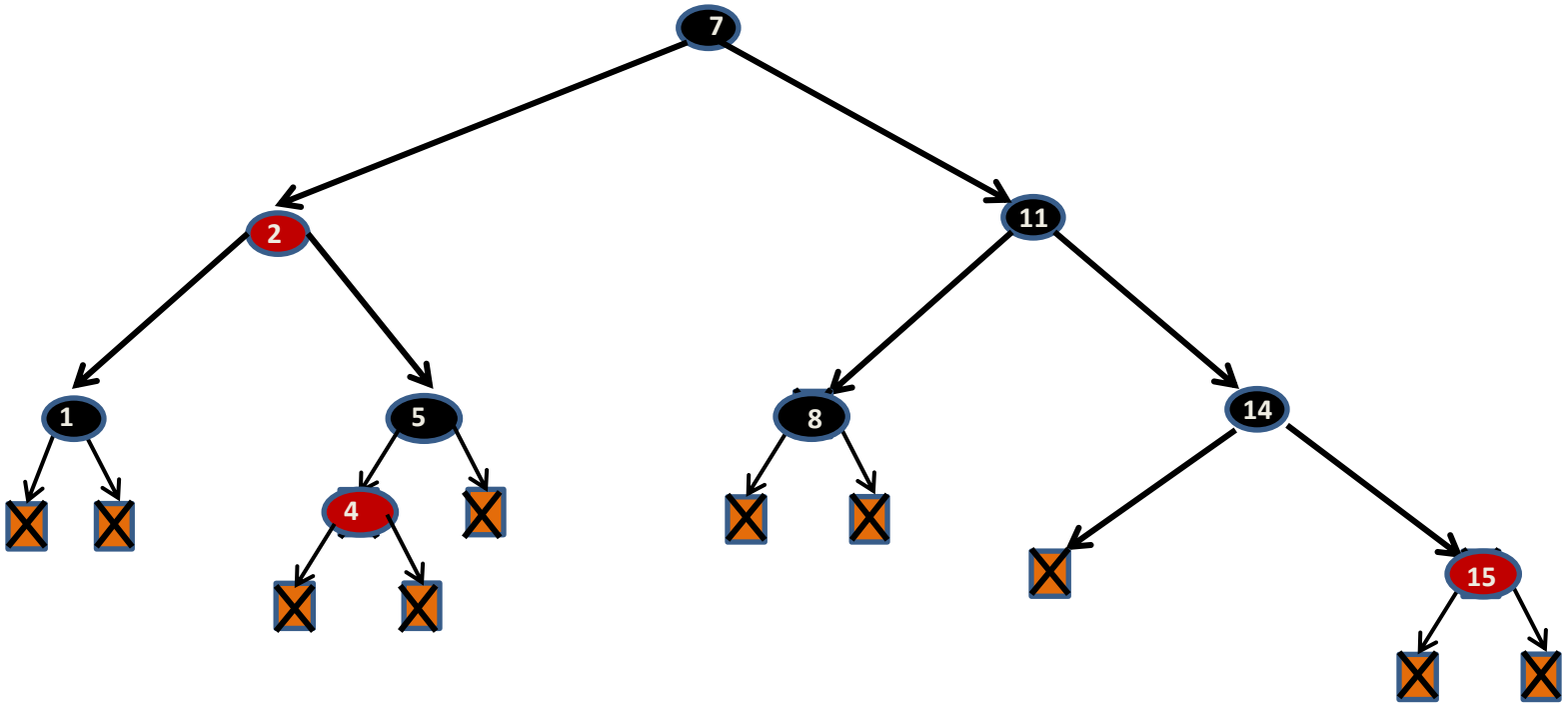
How to insert 4 ?



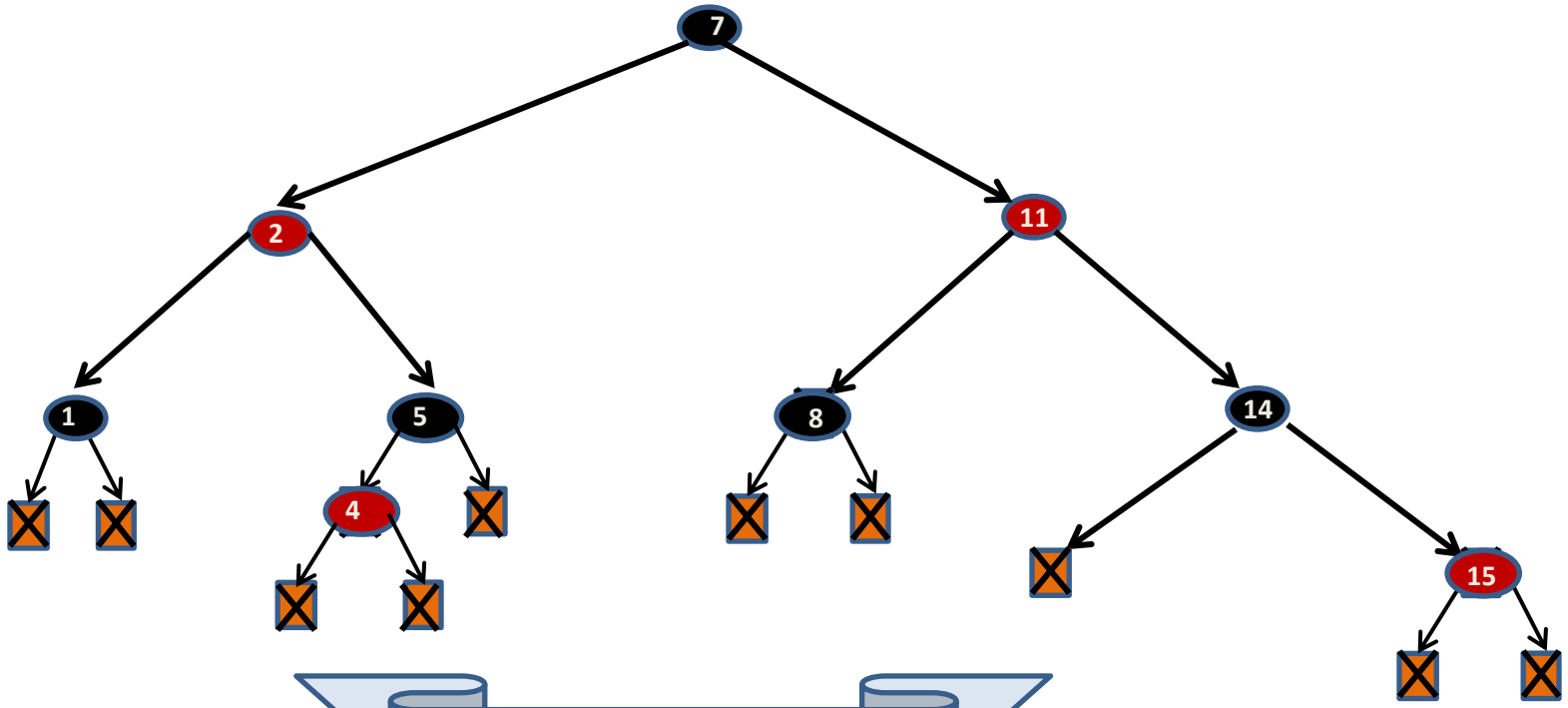
How to insert 4 ?



How to insert 4 ?

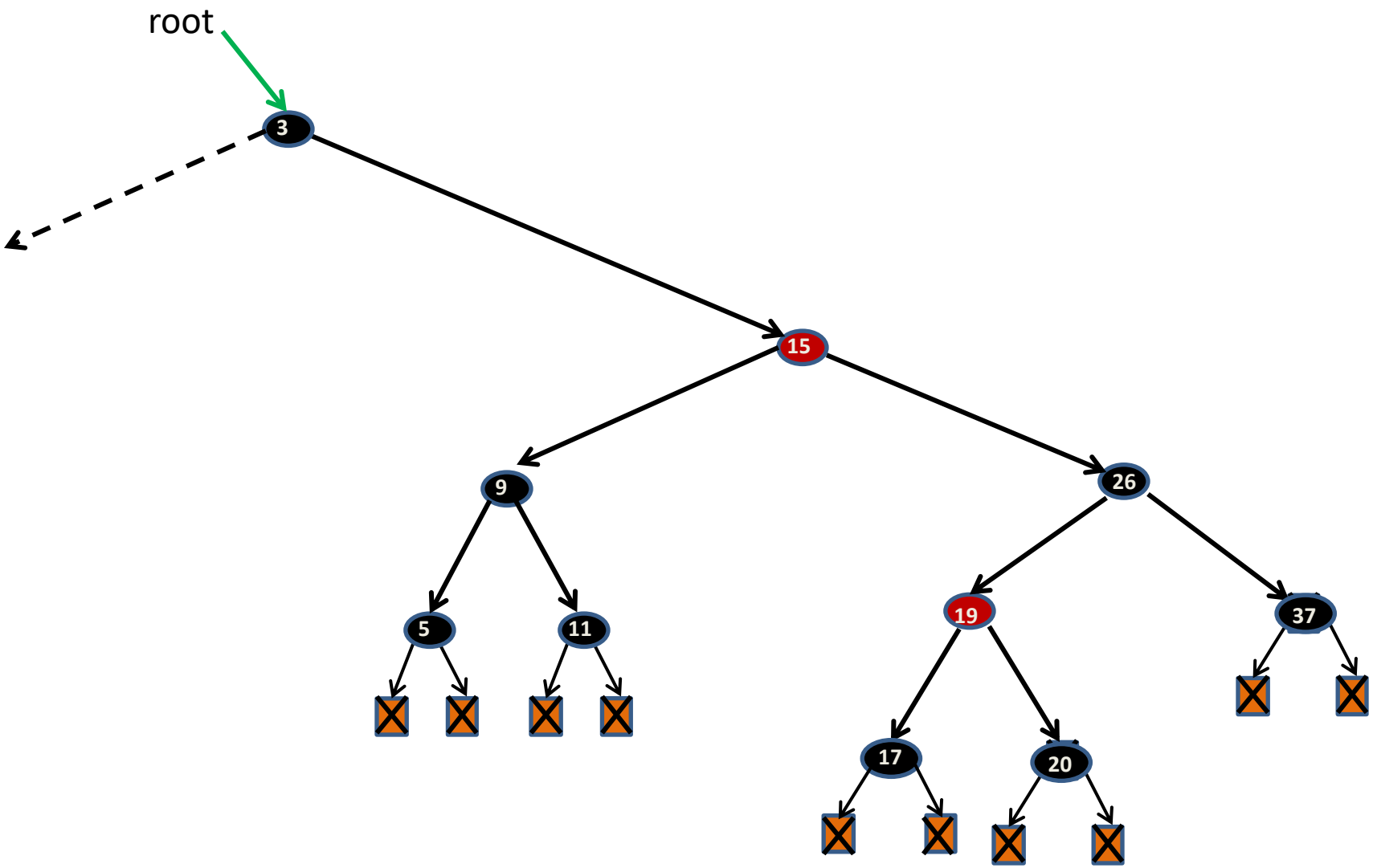


How to insert 4 ?

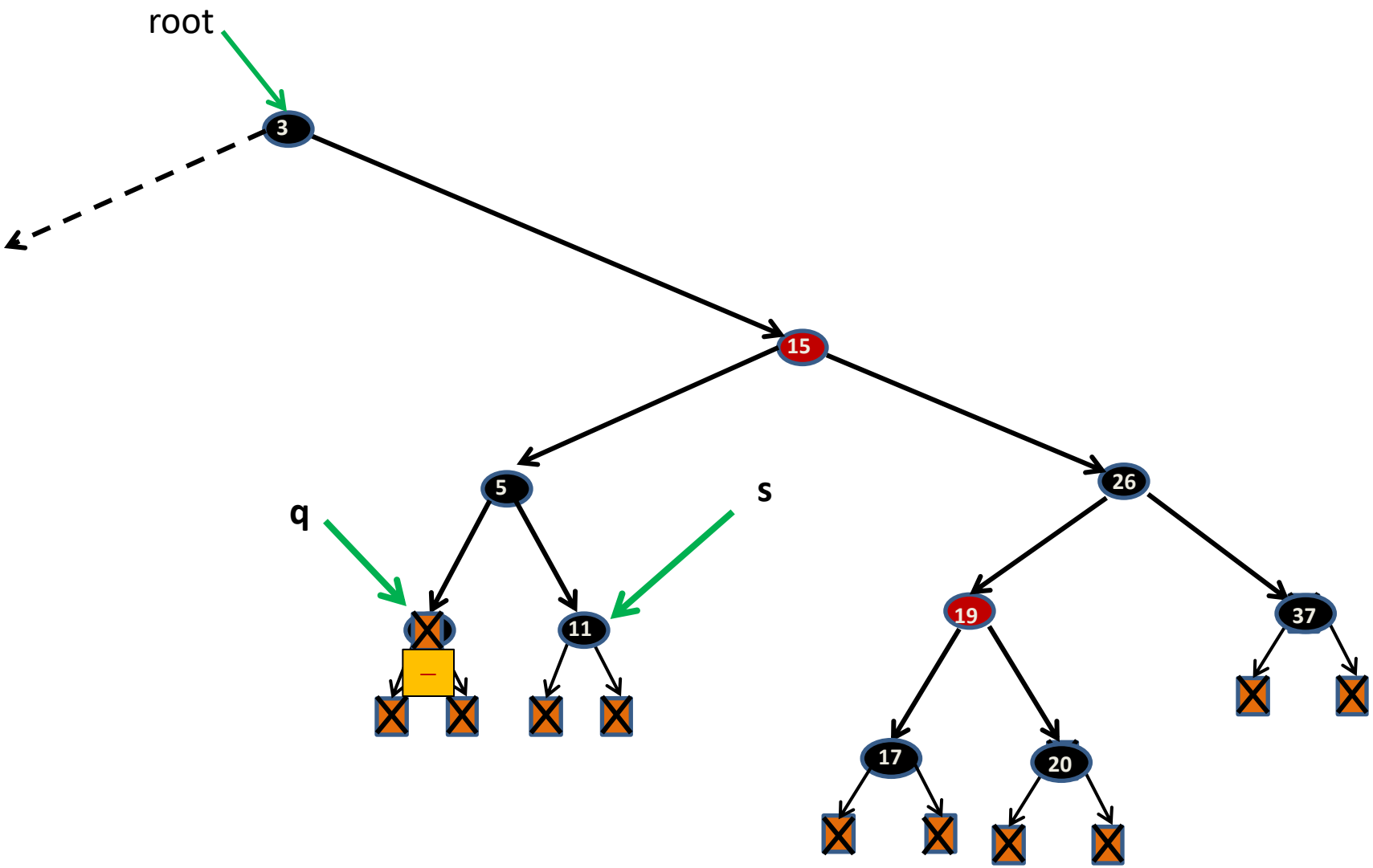


All properties of **red**-**black** tree are restored now☺

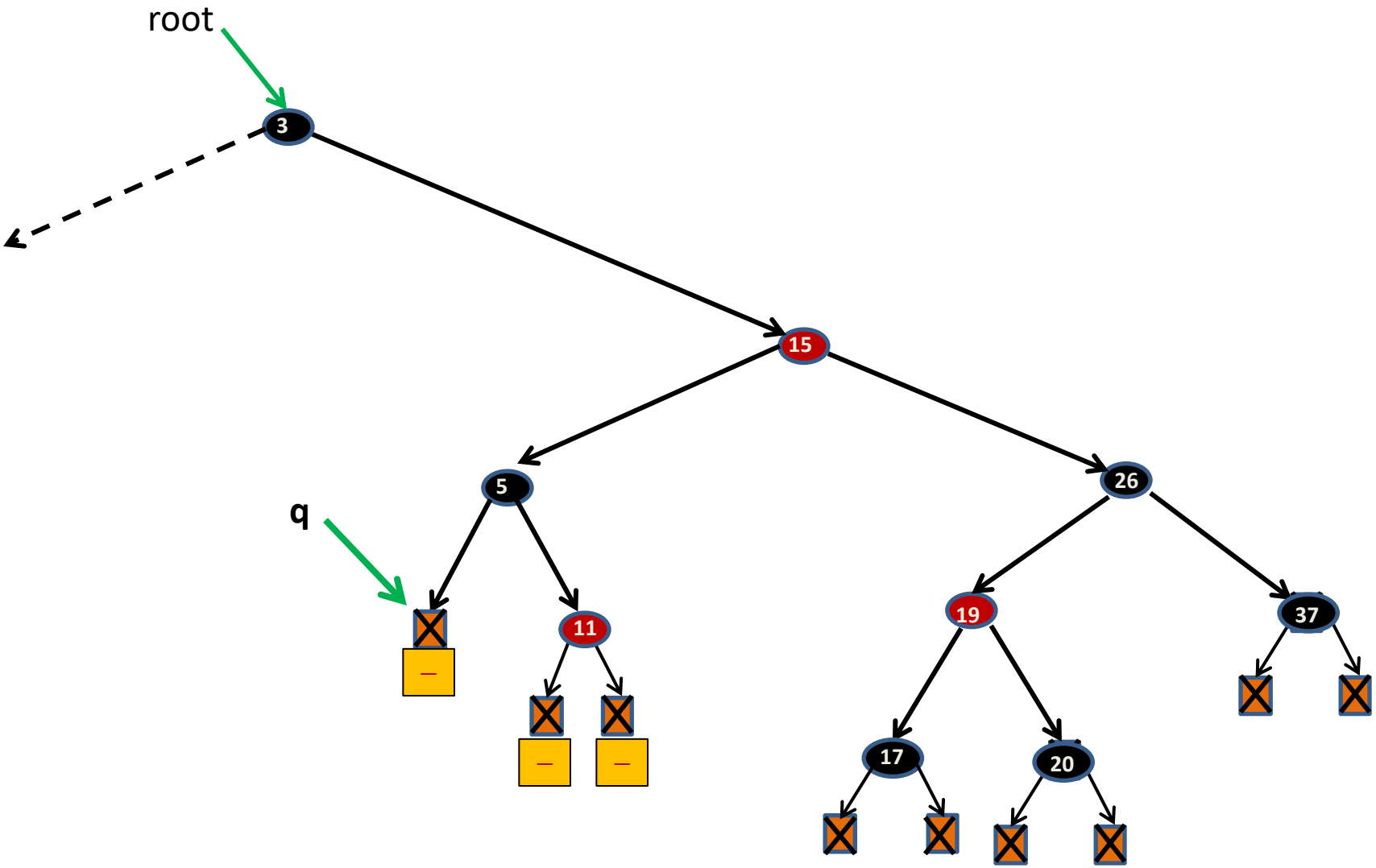
How to delete 9 ?



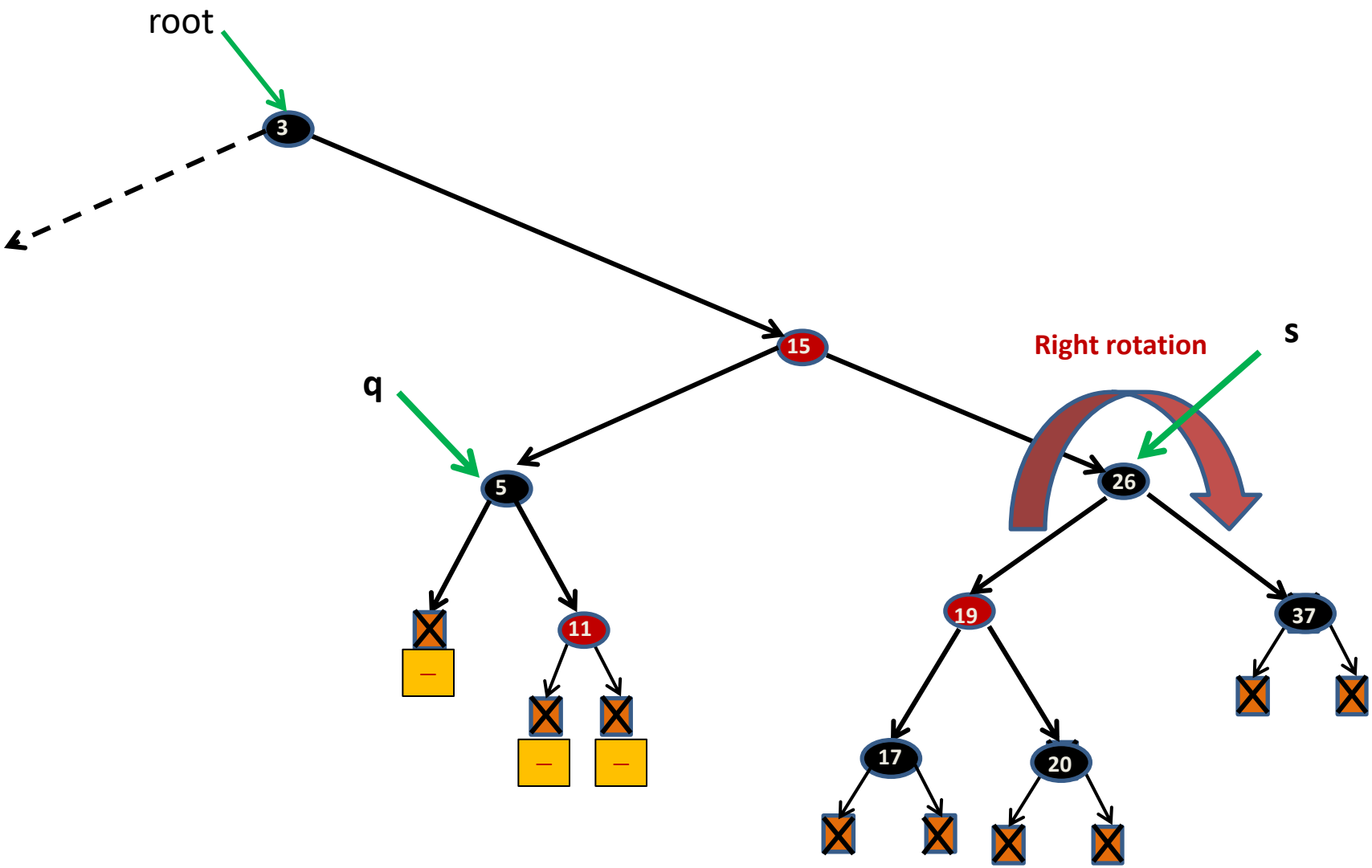
How to delete 9 ?



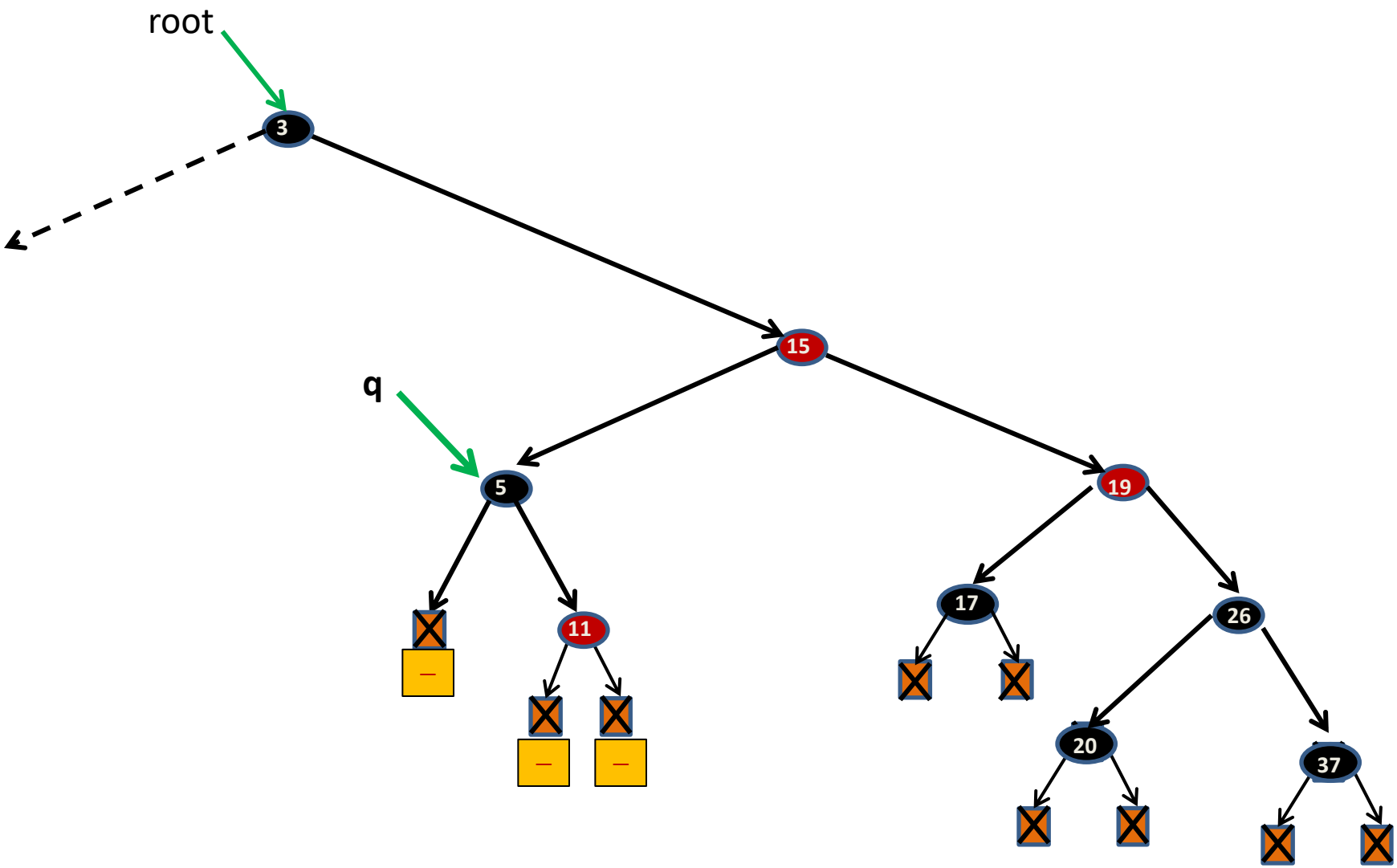
How to delete 9 ?



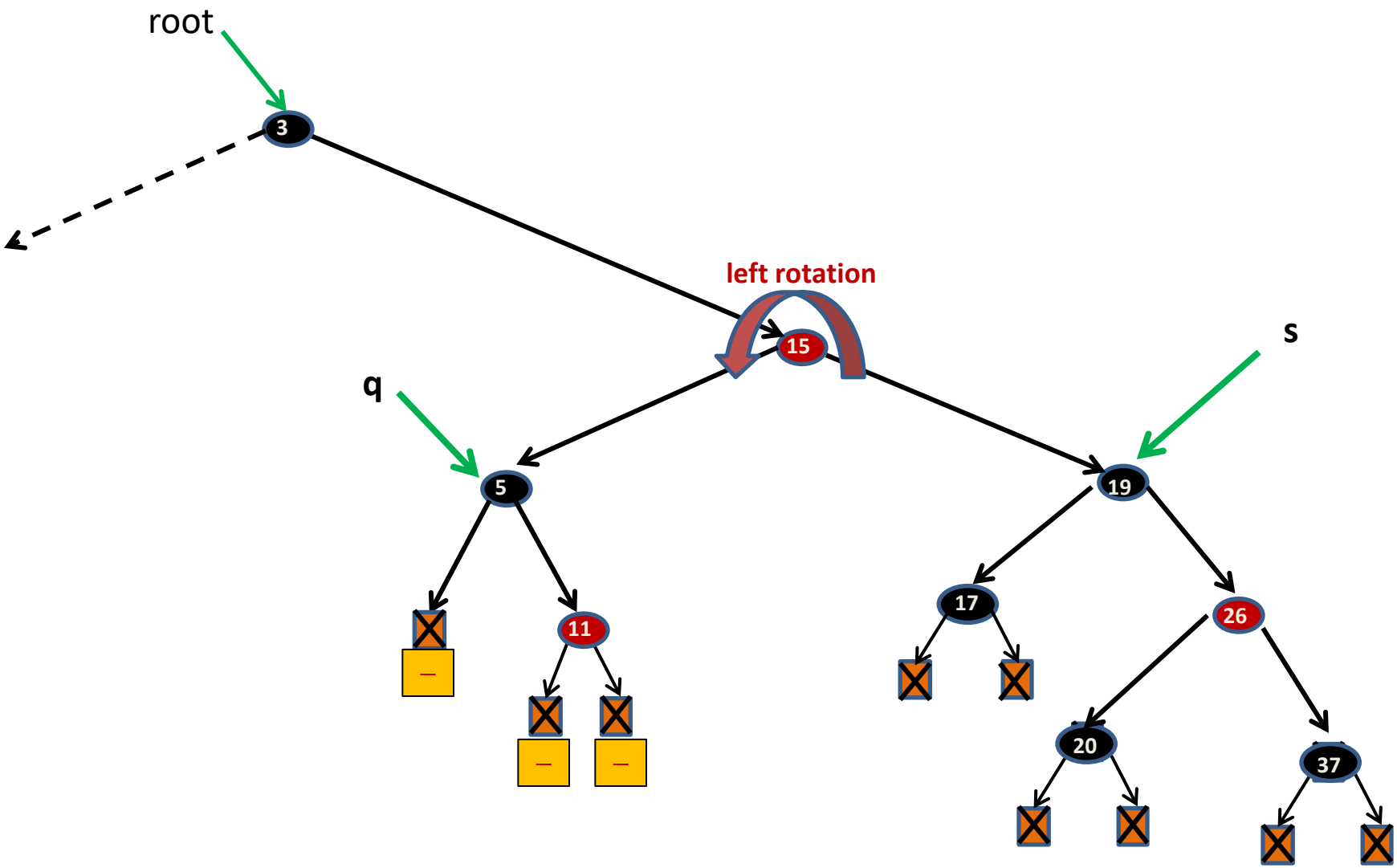
How to delete 9 ?



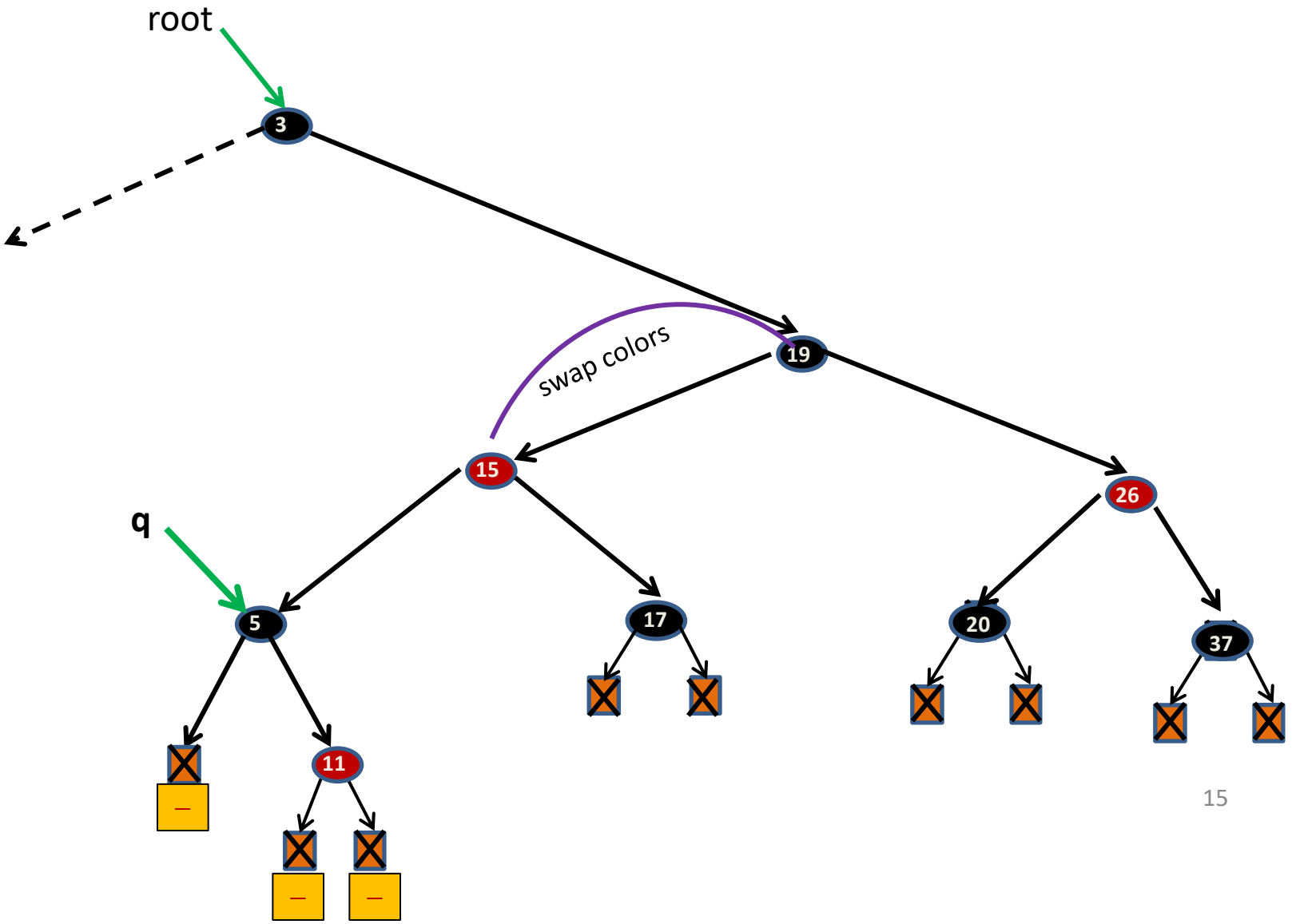
How to delete 9 ?



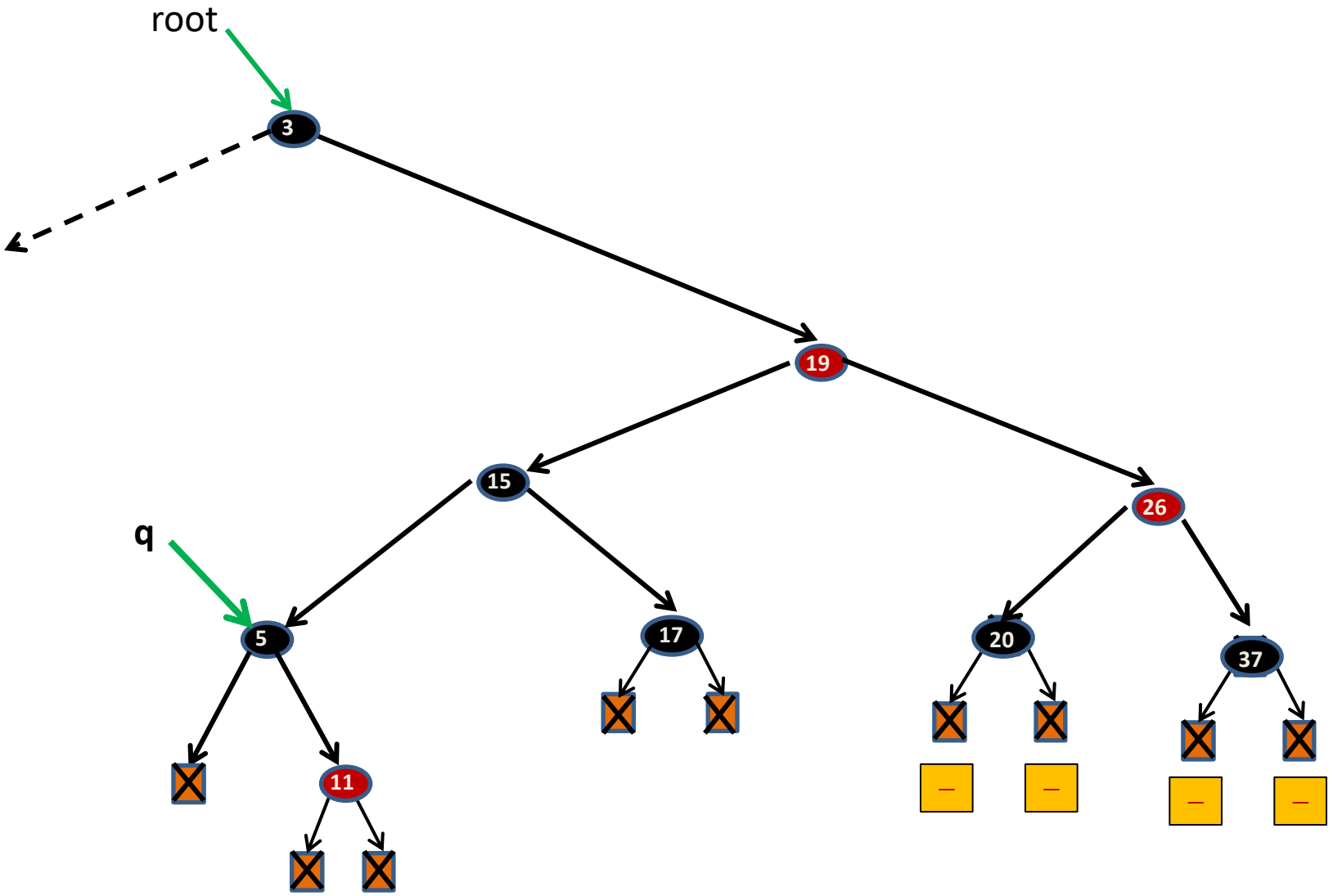
How to delete 9 ?



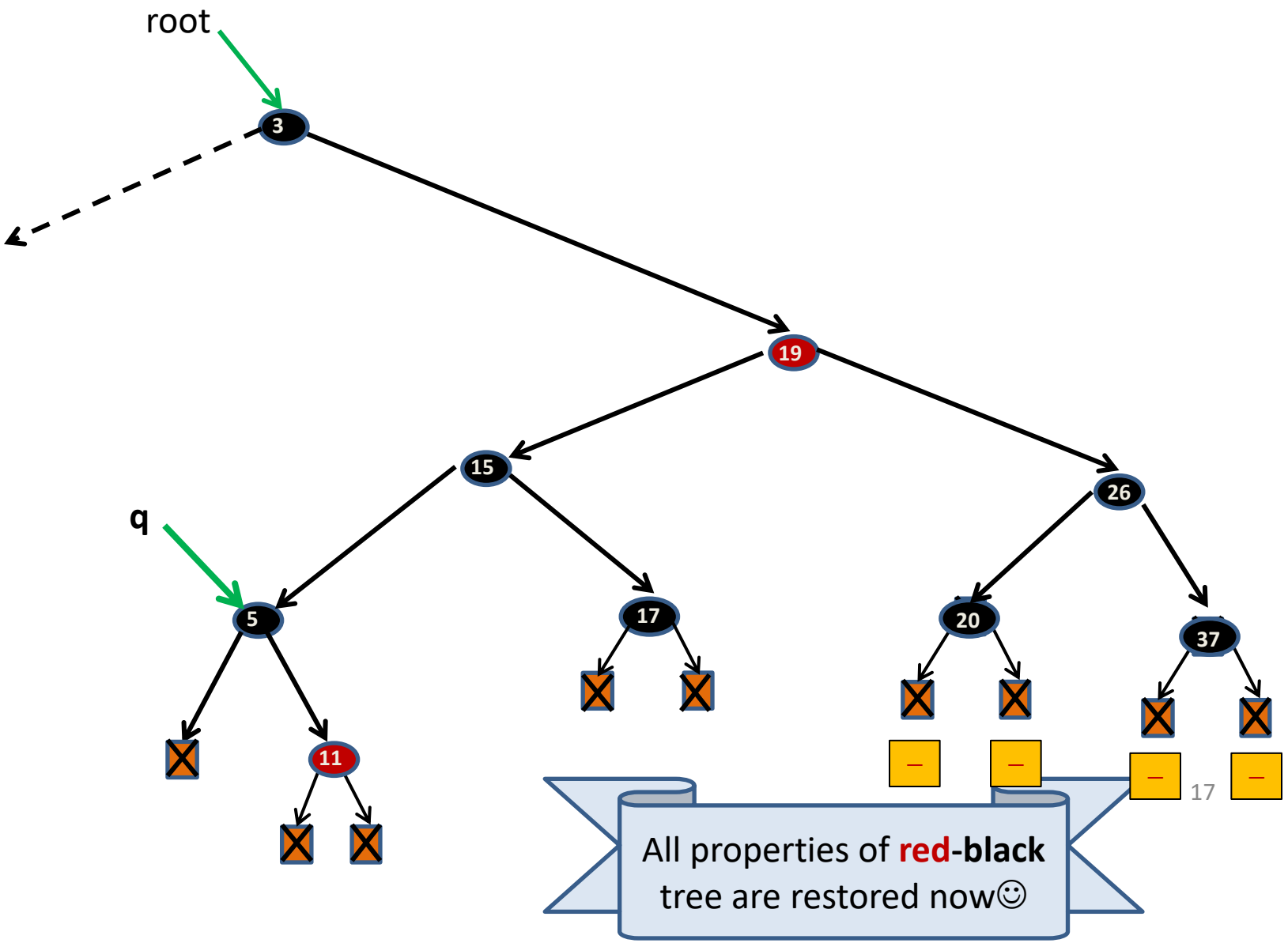
How to delete 9 ?



How to delete 9 ?



How to delete 9 ?



Practice problems

Sheet 2

Stack with maxima

You need to maintain a **stack** whose elements will be positive numbers.

There will be an additional operation called **Report-Maximum**.

This operation is supposed to return the maximum element among all those elements present in the stack.

You need to provide an implementation that will achieve **O**(1) time for each operation (including **Report-Maxima**) on the stack.

At every moment of time, your data structure must occupy **O**(n) space,

where n is the number of elements present in the stack at that time.

Stack with maxima

Solution sketch:

- Keep **two** stacks
  - One stack for usual pushing and popping of elements
  - Another stack for storing Max of all elements in the stack

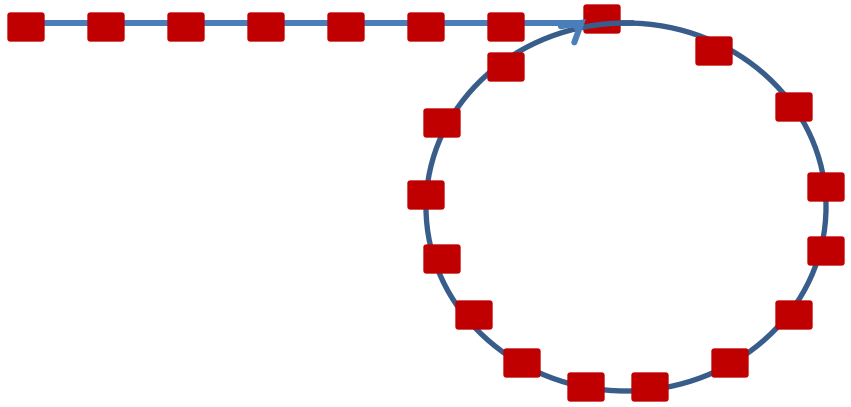
(Give details of **push** and **pop** operations now)

Question : Is it possible to achieve this goal using just one stack ?

Answer: Yes.

Ponder over this question with free mind without any worry (because it will not be asked in the exams of this course.)

Detecting self-loop in linked list



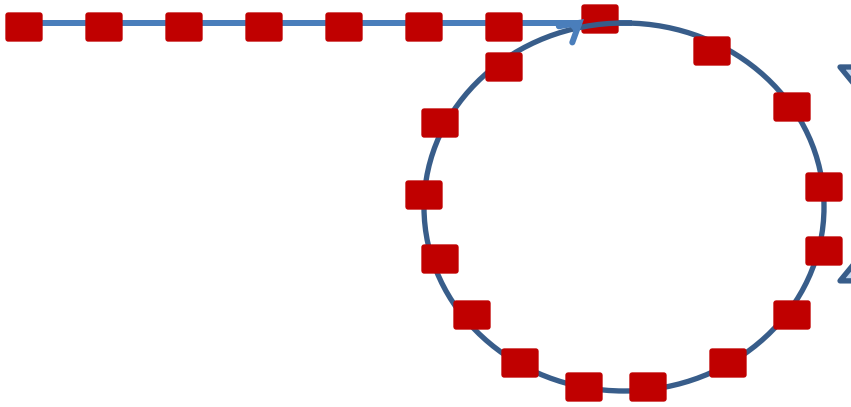
Aim:

To detect whether there is any **self-loop** in the linked list in **O**(k) time, where k is the **no. of nodes** in the list.

Let us first try to design an algorithm for this problem.

Then we shall try to improve its time complexity.

Detecting self-loop in linked list



Translate this observation into an algorithm.

Intuition:

If there is a self loop, then after a complete traversal of the list,

we shall start visiting nodes again and again.

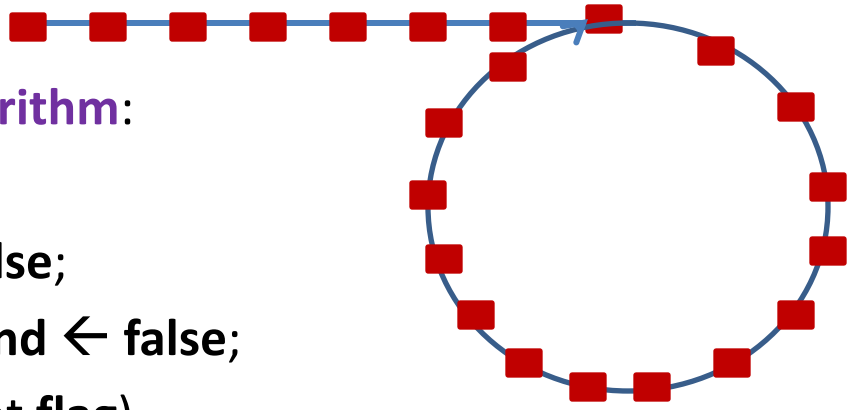
Observation: Suppose after traversing j steps, we are at node p.

If we happen to visit p again in the next j steps

then There is a self loop

Else There is **no** self loop in the prefix of length j of the list

Detecting self-loop in linked list



First Algorithm:

$j \leftarrow 2$;

flag ← false;

Loop-found ← false;

While (not **flag**)

{ Traverse loop from the **start** for j steps;

Let **p** be the node at jth place;

In the next j steps if we reach **NULL** then **flag← true**;

else if **p** is visited again then {**Loop-found ← true**; **flag← true** }

else $j \leftarrow j + 1$;

}

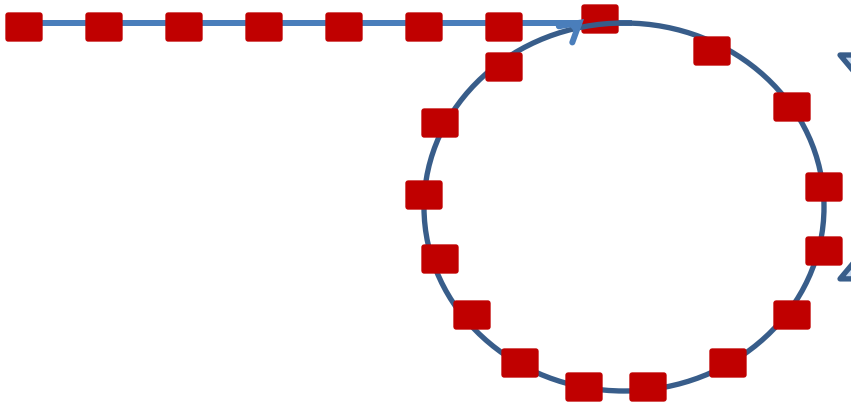
If (**Loop-found**) then print "there is self-loop" else print "there is no self-loop"

The correctness of the algorithm follows from the previous slide.

Time complexity :

$$\sum_{1}^{k} O(j) = O(k^2)$$

Detecting self-loop in linked list



Use this hint to **speed-up** the First algorithm.

Question: Can you improve the time complexity ?

Hint:

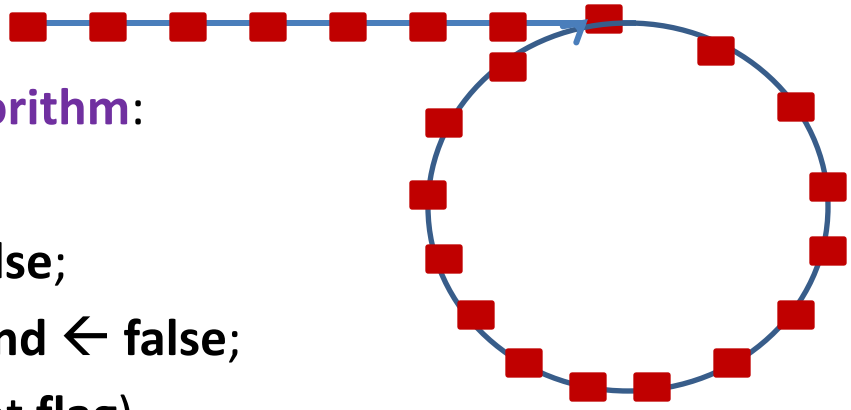
Let integer k^* be the power of 2 such that $k \le k^* < 2k$. Then observe that

$$\sum_{1}^{k} O(j) = O(k^2)$$

but

$$1 + 2 + 4 + 8 + \cdots + k^* = O(k)$$

Detecting self-loop in linked list



Time complexity of **Final algorithm** is $O(k)$.
Can you simplify the algorithm further ?

Final Algorithm:

$j \leftarrow 2$;

flag ← false;

Loop-found ← false;

While (not **flag**)

{ Traverse loop from the **start** for j steps;

Let **p** be the node at jth place;

In the next j steps if we reach **NULL** then **flag← true**;

else if **p** is visited again then {**Loop-found ← true**; **flag← true** }

else $j \leftarrow 2 * j$;

}

If (**Loop-found**) then print “there is self-loop” else print “there is no self-loop”

Practice problems

Sheet 3

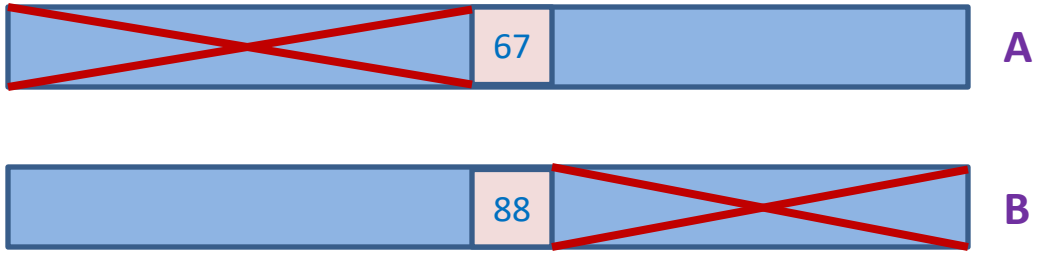
Median of 2 arrays

There are 2 sorted arrays **A** and **B**, each storing n distinct numbers.

Design an algorithm to find the median of **A**∪**B**.

The time complexity of your algorithm should be **O**(log n).

Solution sketch:



Compare the medians of **A** and **B**.

Based on the outcome, discard a **fraction** of both **A** and **B**.

Proceed recursively.

Note: Be careful in choosing the parameters of recursive calls.

Rest of the course

- Graph algorithms
  - (we shall cover it just after the mid-semester exam.)
- Greedy algorithms
- **Incredible** data structures