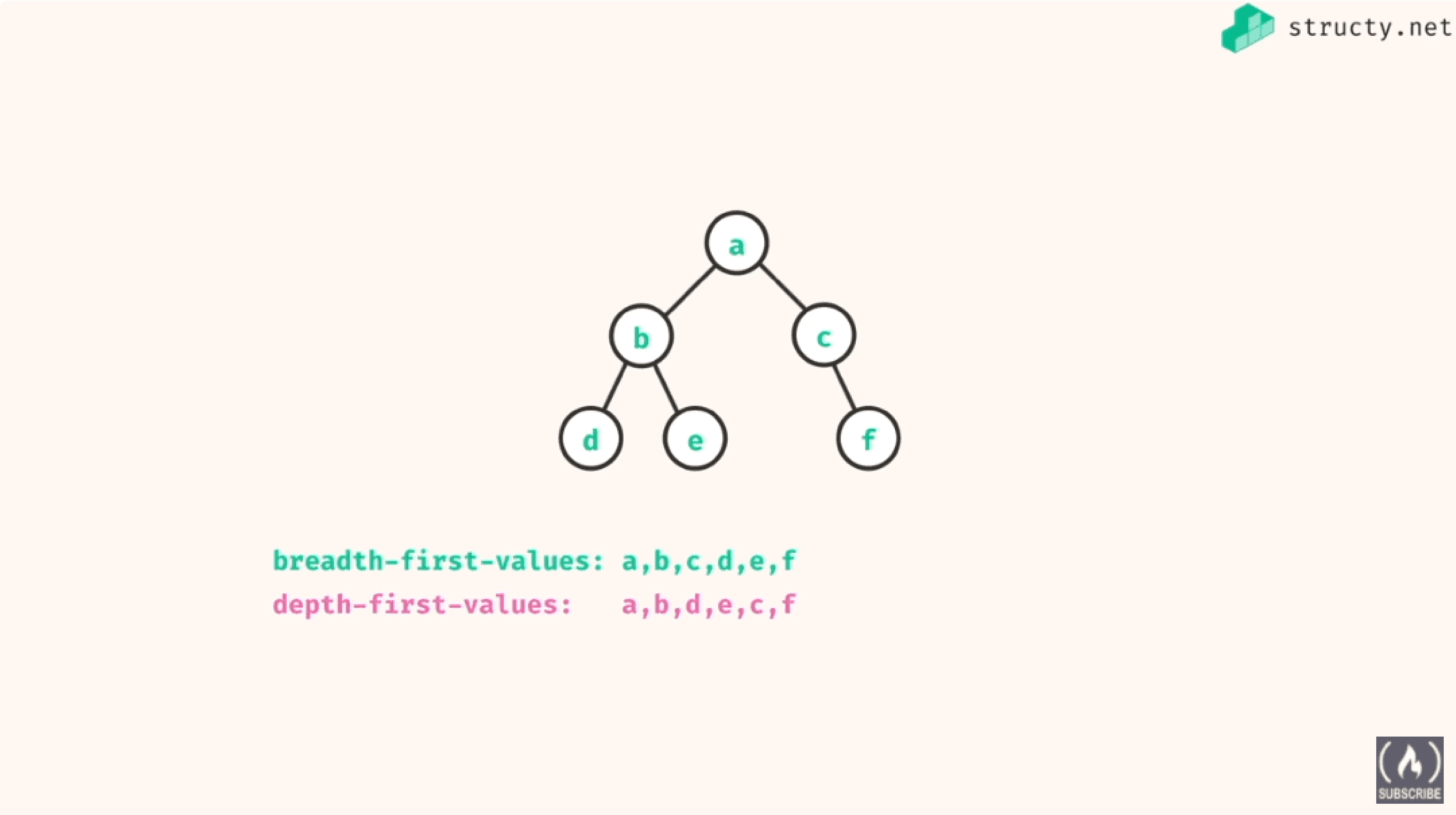
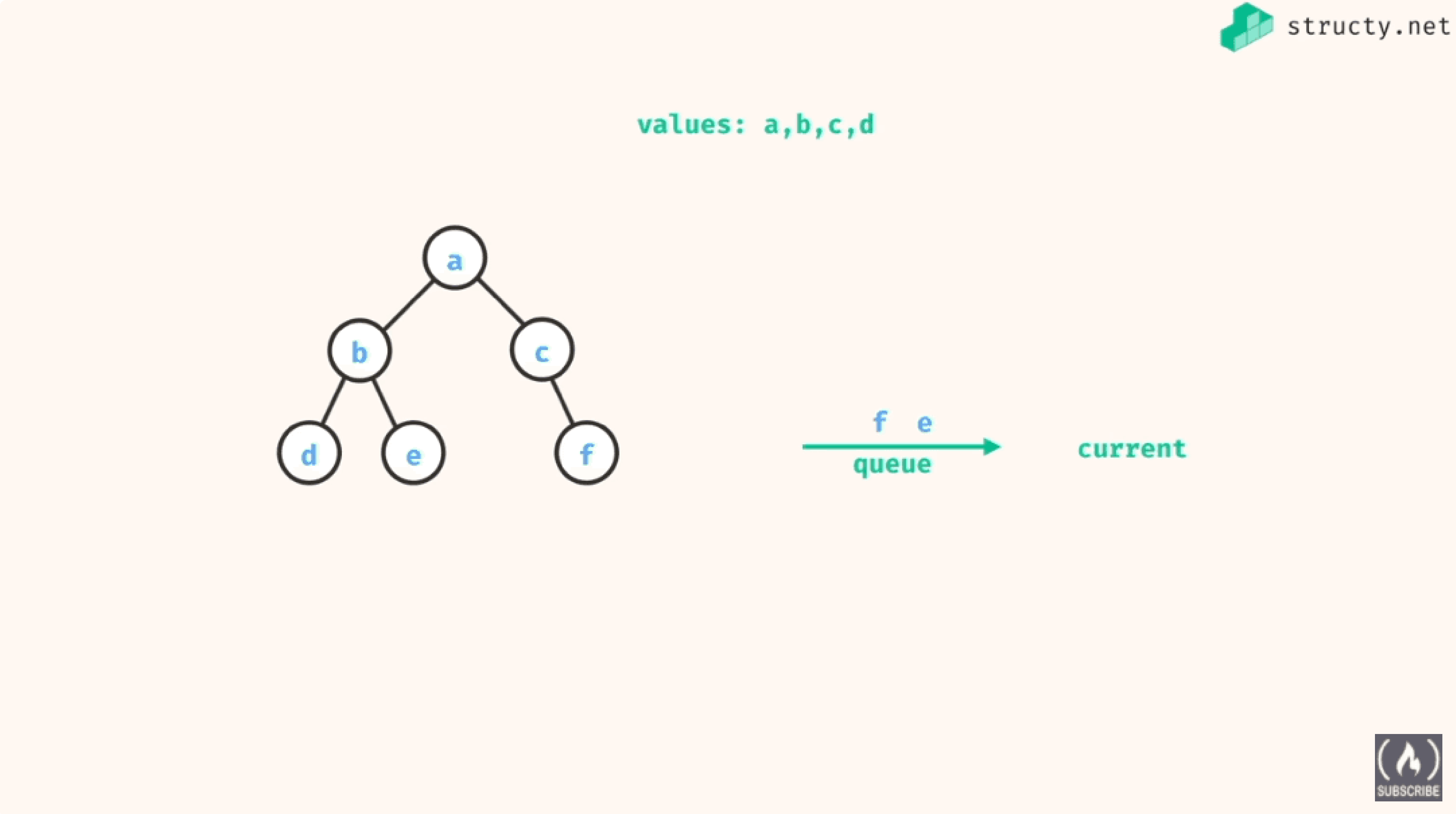
Breadth firsts values:

travellling across the tree



for bredth first we use a **Queue**



Time complexity and space complexity-> O(n) (assuming adding and removing item is constant time)

1. iterative code (in JS):

```javascript
const breadthFirstValues = (root) => {
  if (root === null) return [];

  const values = [];
  const queue = [ root ];

  while (queue.length > 0) {
    const current = queue.shift();
    values.push(current.val);

    if (current.left !== null) queue.push(current.left);
    if (current.right !== null) queue.push(current.right);
  }

  return values;
};
```

->no recursive solution because breadth first needs a queue and **recursion uses a stack**

Tree includes:

finding the target value in the given bianry tree

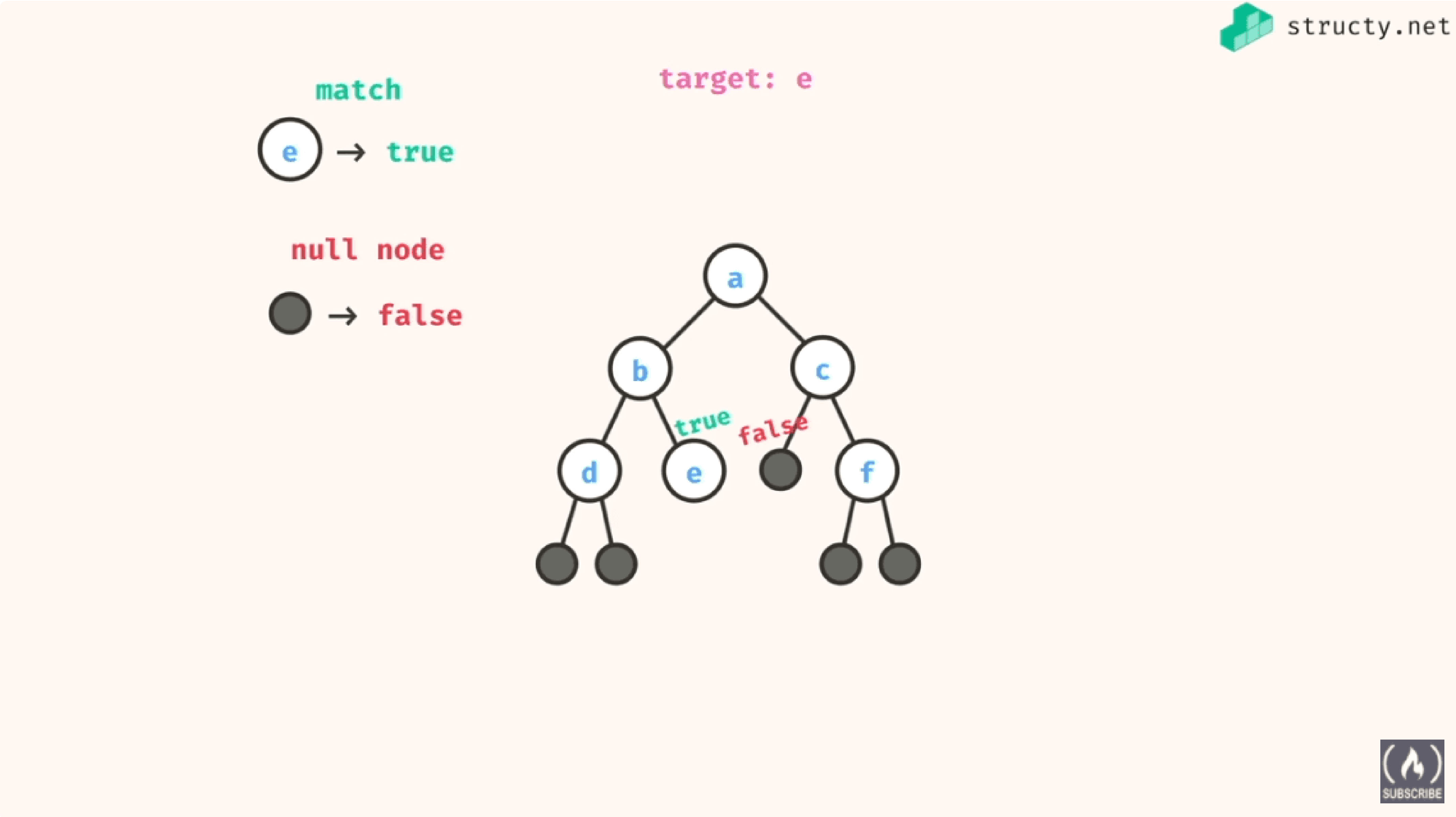
1. Breadth first approach=> breadth first search:

-> time complexity -> O(n)

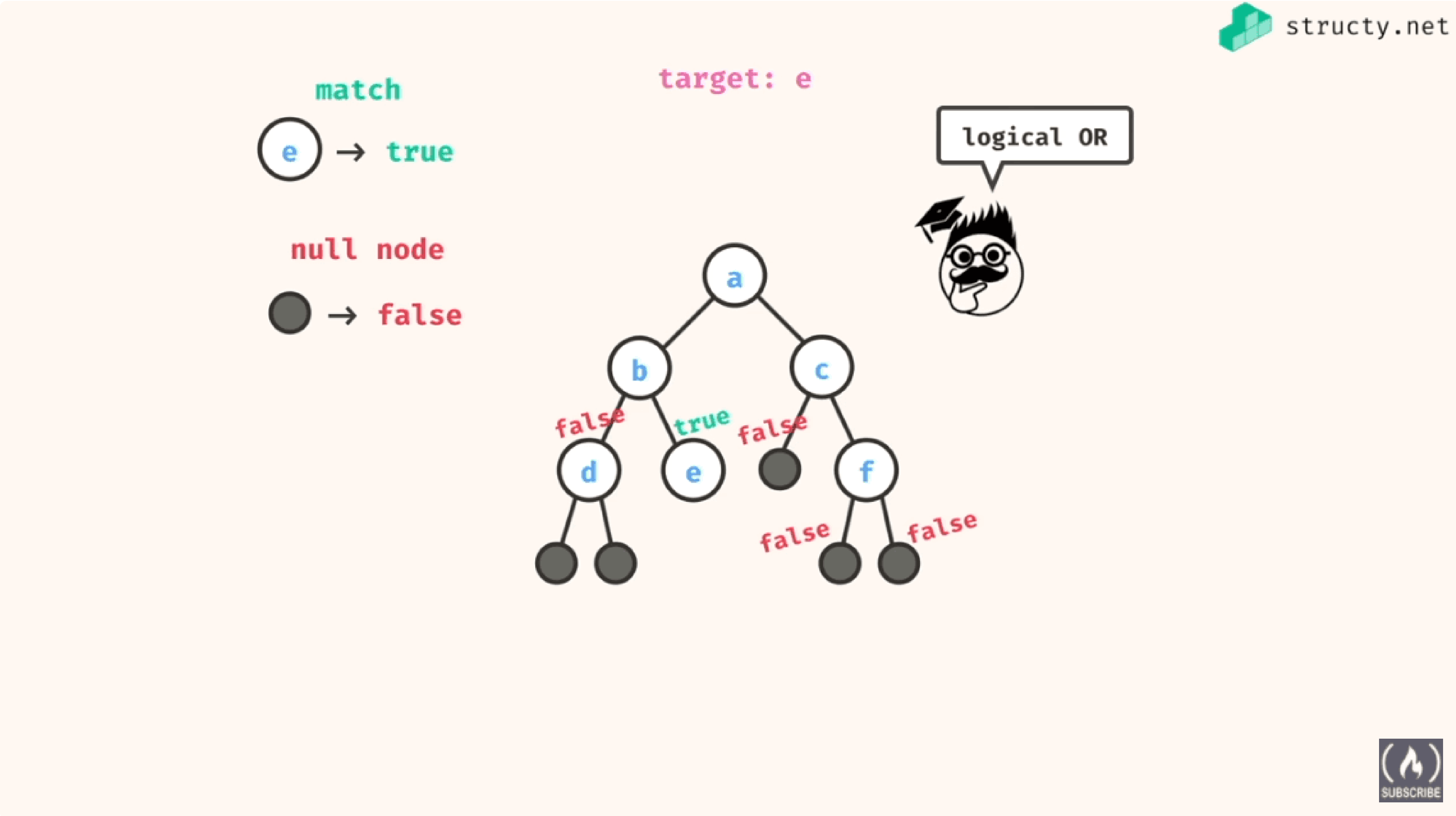
-> Space complexity -> O(n)

2. Depth first approach => depth first search:

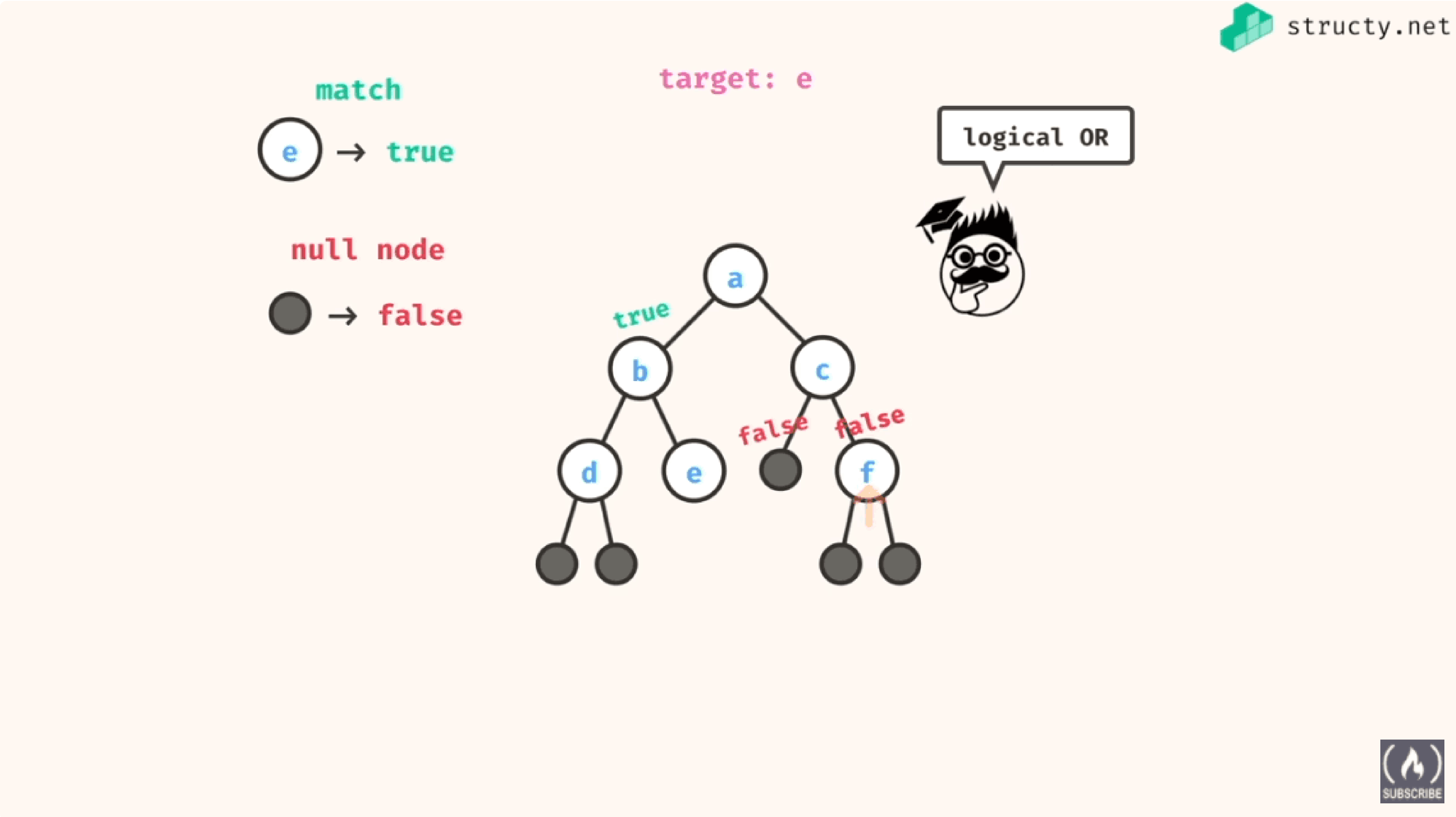
-> better use recursion



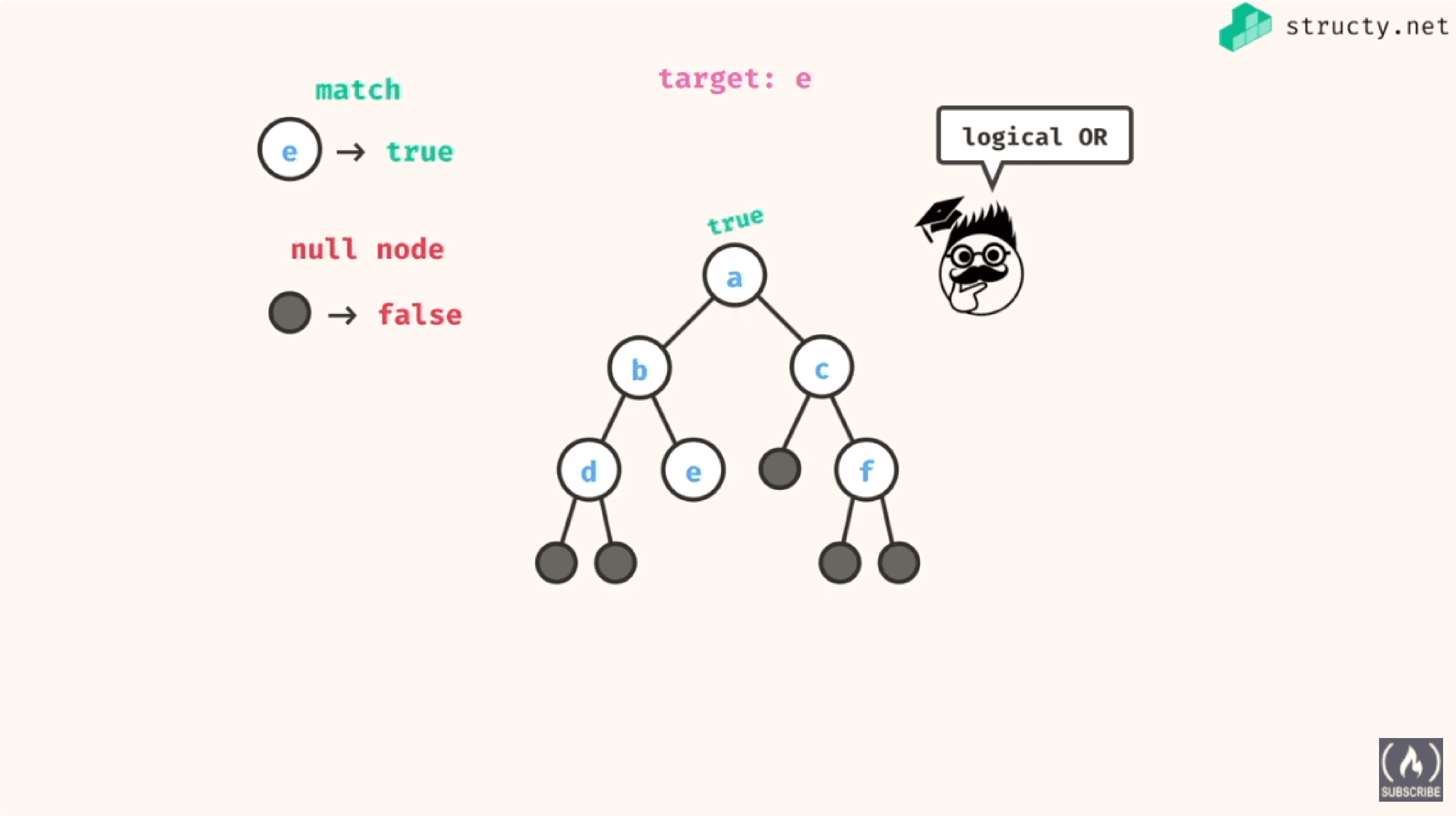
structy.net
target: e
match
e → true
null node
→ false
a
b
c
true
false
d
e
f
SUBSCRIBE



structy.net
target: e
match
e → true
null node
→ false
logical OR
a
b
c
d
e
f
false
true
false
false
false
SUBSCRIBE



structy.net
target: e
match
e → true
null node
→ false
logical OR
a
b
c
d
e
f
true
false
false
SUBSCRIBE


structy.net
match
e
→ true
null node
→ false
target: e
logical OR
true
a
b
c
d
e
f
SUBSCRIBE

Code:

1. Breadth first search

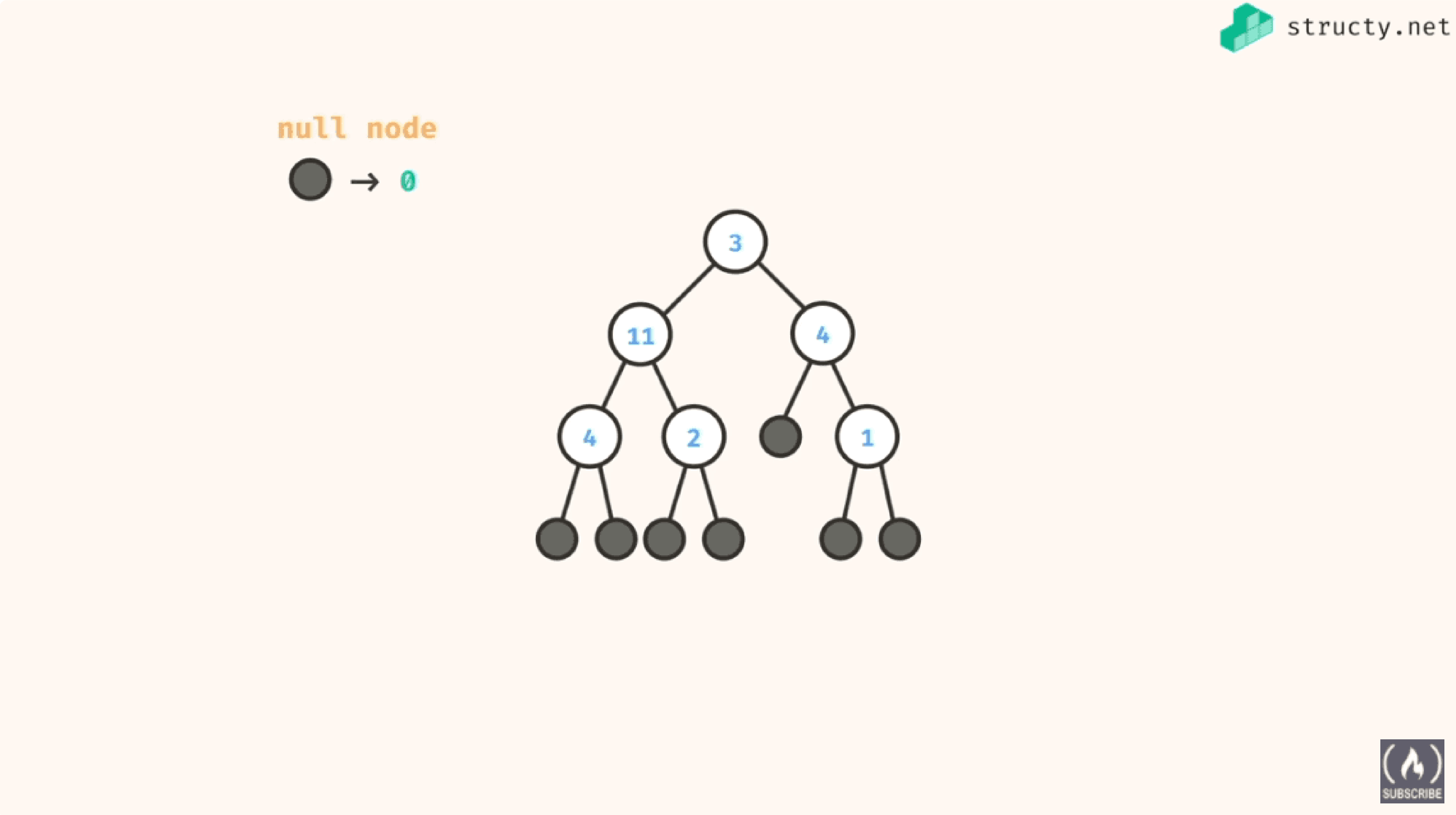
```
const treeIncludes = (root, target) => {
  if (root === null) return false;

  const queue = [ root ];
  while (queue.length > 0) {
    const current = queue.shift();
    if (current.val === target) {
      return true;
    }
    if (current.left) queue.push(current.left);
    if (current.right) queue.push(current.right);
  }
  return false;
};
```

2. Depth first search (recursive):

```
const treeIncludes = (root, target) => {
  if (root === null) return false;
  if (root.val === target) return true;
  return treeIncludes(root.left, target) || treeIncludes(root.right, target);
};
```

Tree sum:

1. Depth first / recursive:



-> time complexity -> O(n)

-> Space complexity -> O(n)

Code:

-> Depth first recursive

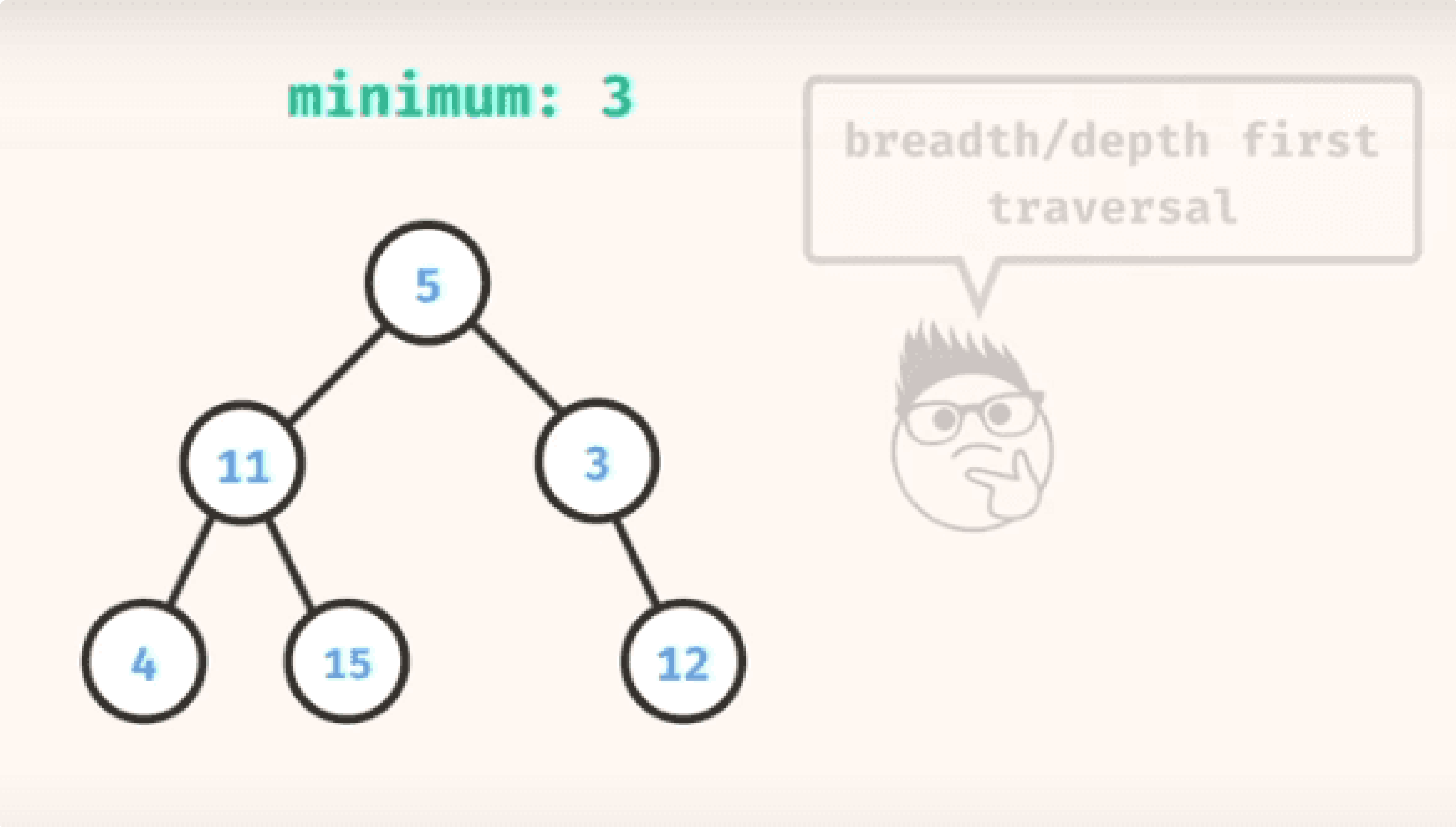
```
const treeSum = (root) => {
  if (root === null) return 0;
  return root.val + treeSum(root.left) + treeSum(root.right);
};
```

-> Breadth first iterative

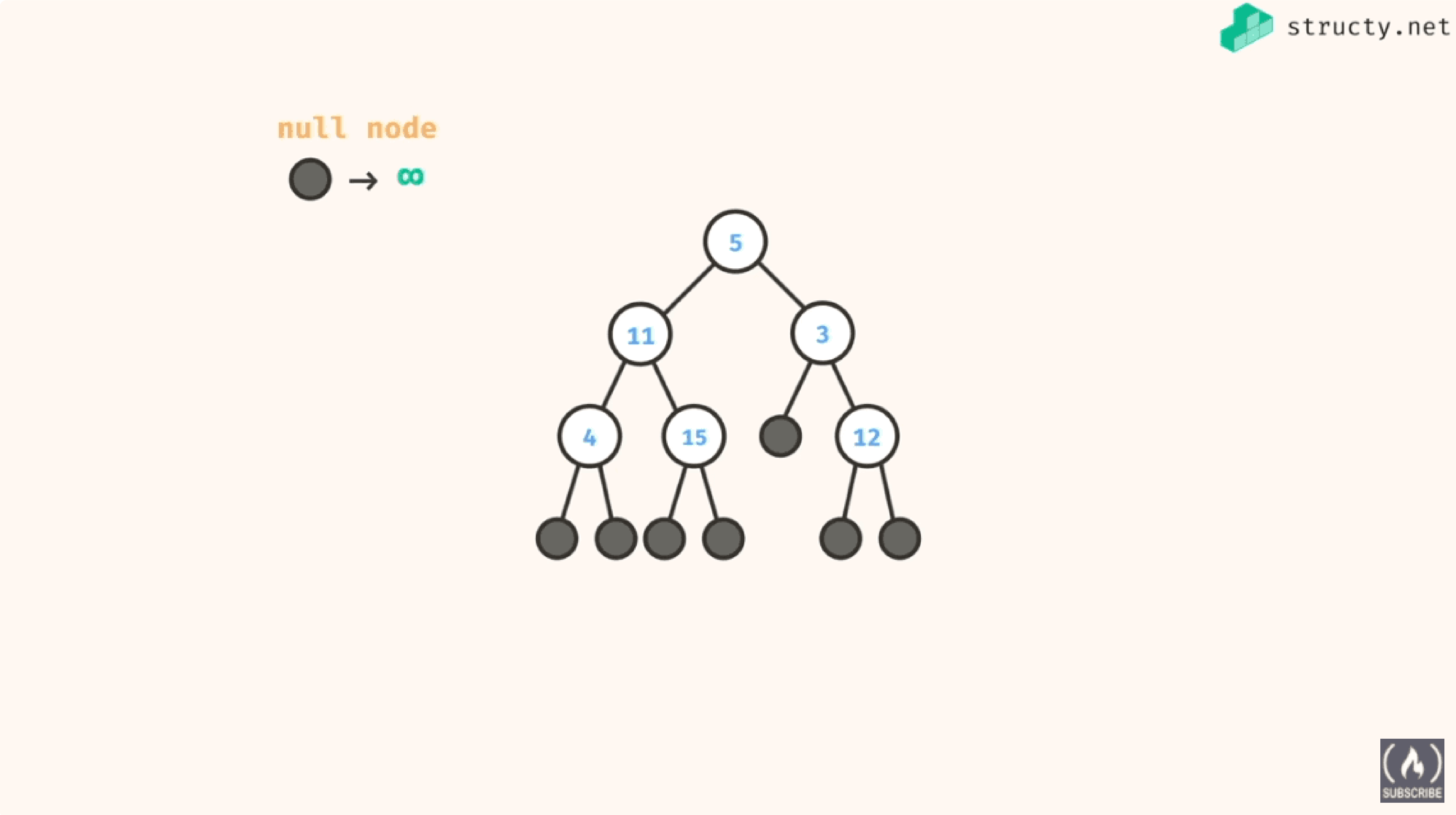
```
const treeSum = (root) => {
  if (root === null) return 0;
  let totalSum = 0;
  const queue = [ root ];

  while (queue.length > 0) {
    const current = queue.shift();
    totalSum += current.val;
    if (current.left !== null) queue.push(current.left);
    if (current.right !== null) queue.push(current.right);
  }

  return totalSum;
};
```

Tree min value:

finding the smallest number



1. Recursive approach:



-> time complexity -> O(n)

Code:

1. iterative depth first:

```
const treeMinValue = (root) => {
  let smallest = Infinity;
  const stack = [ root ];
  while (stack.length > 0) {
    const current = stack.pop();
    if (current.val < smallest) smallest = current.val;

    if (current.left !== null) stack.push(current.left);
    if (current.right !== null) stack.push(current.right);
  }

  return smallest;
};
```

2. iterative breadth first:

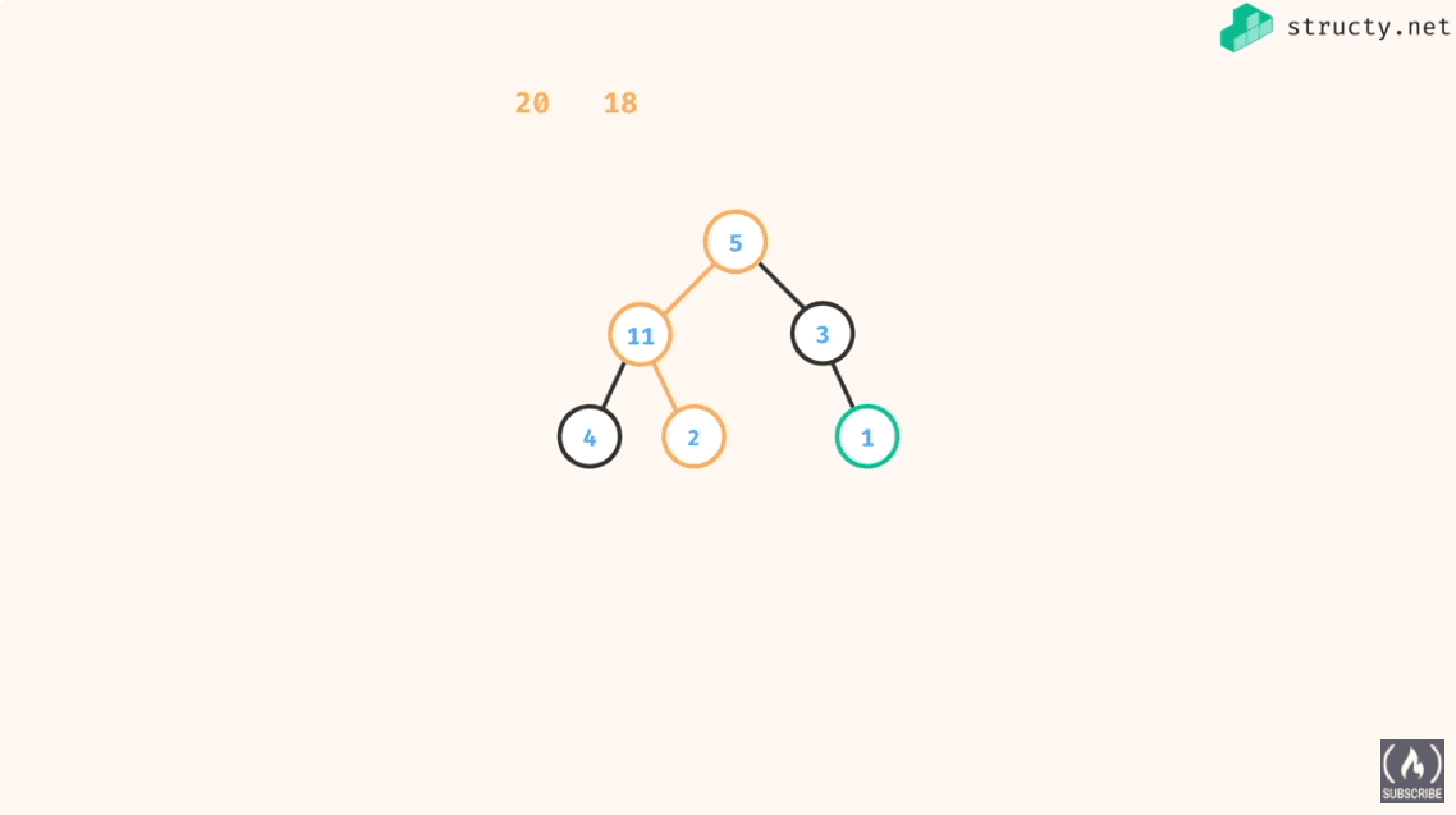
```js
const treeMinValue = (root) => {
  let smallest = Infinity;
  const queue = [ root ];
  while (queue.length > 0) {
    const current = queue.shift();
    if (current.val < smallest) smallest = current.val;

    if (current.left !== null) queue.push(current.left);
    if (current.right !== null) queue.push(current.right);
  }

  return smallest;
};
```

-> in js (as well as in python) we use lists as stack/queue, for better fast perf. use the colections queue/stack

3. recursive:

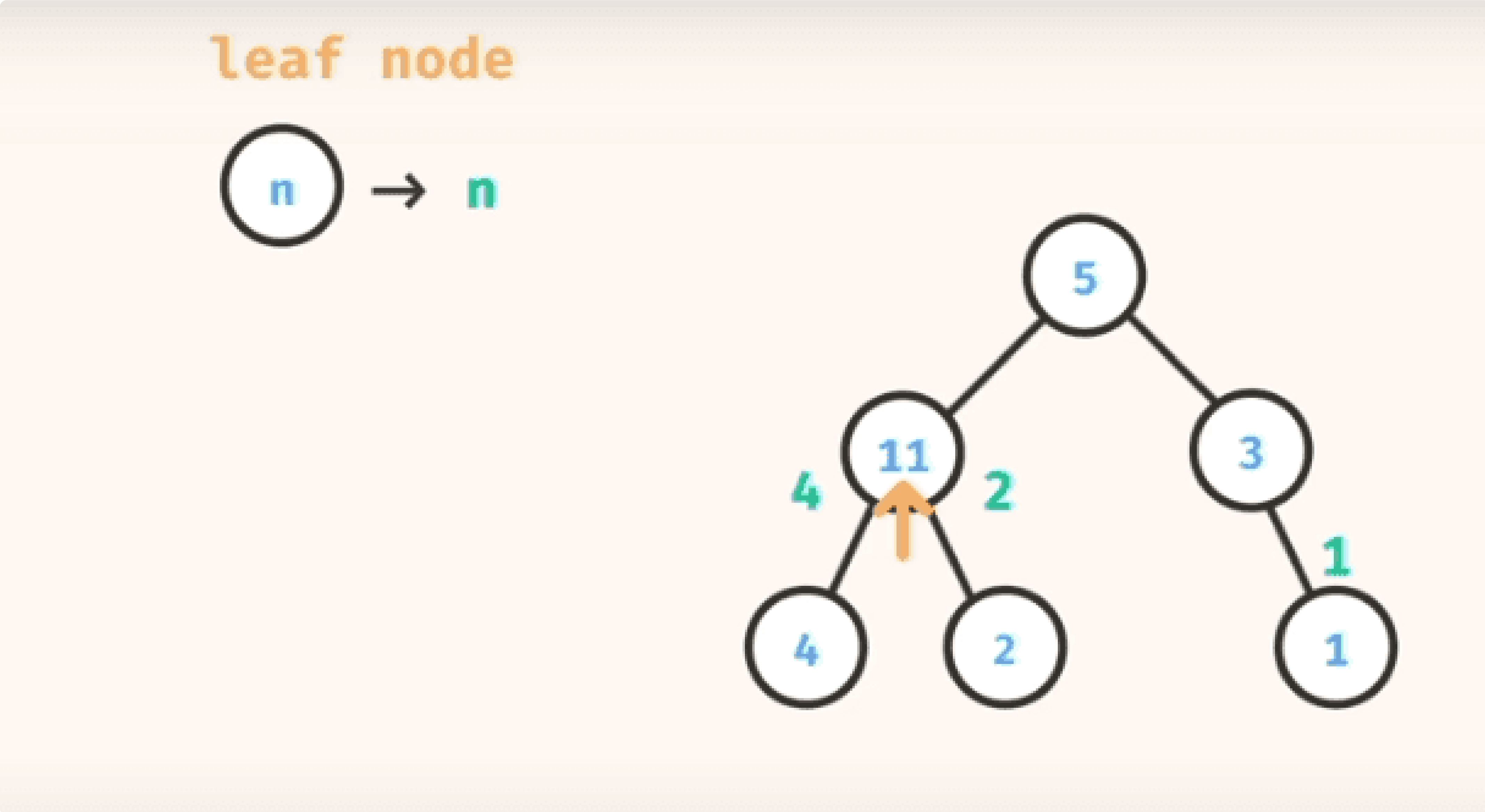
```
const treeMinValue = (root) => {
  if (root === null) return Infinity;
  const leftMin = treeMinValue(root.left);
  const rightMin = treeMinValue(root.right);
  return Math.min(root.val, leftMin, rightMin);
};
```

Max root to leaf path sum:

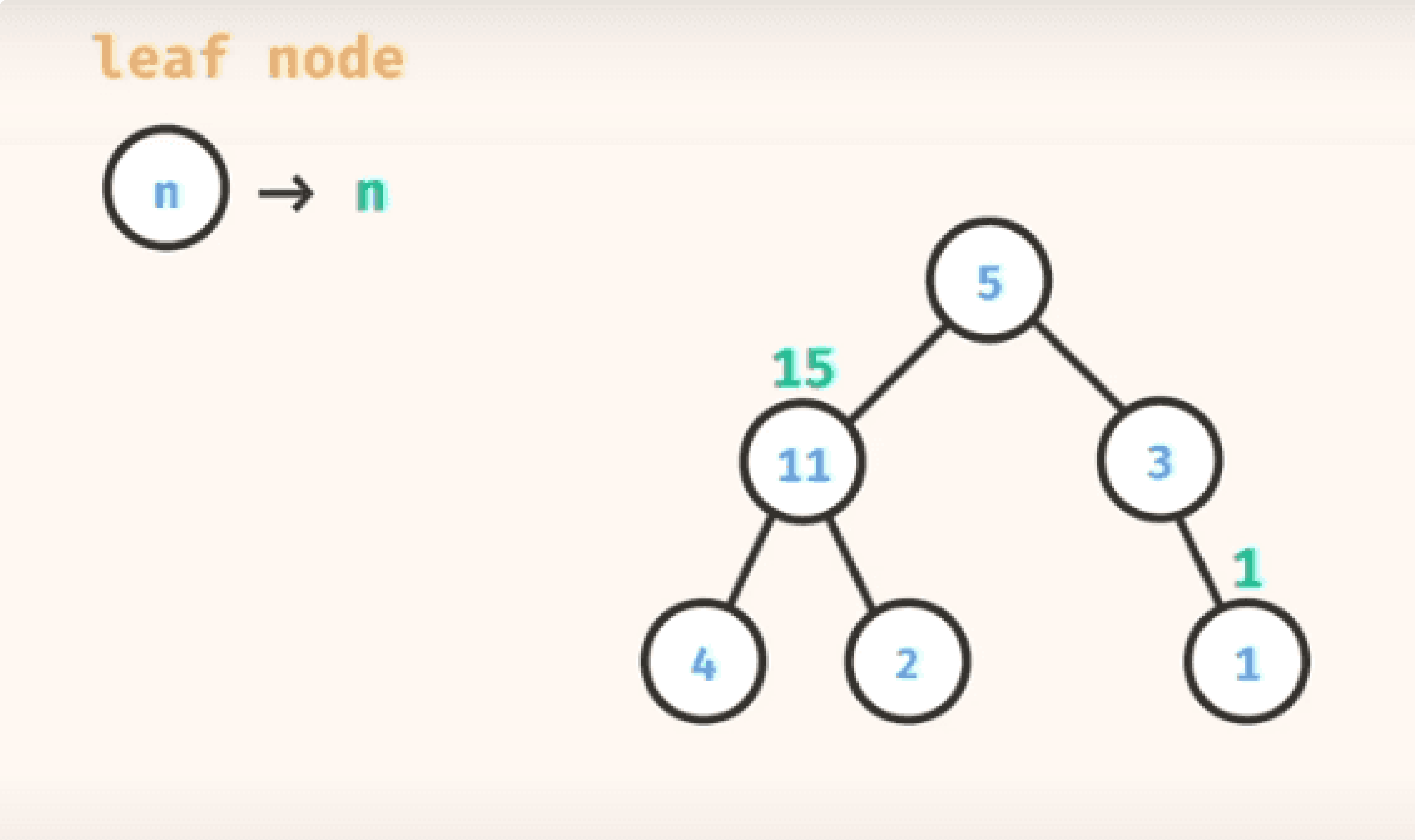


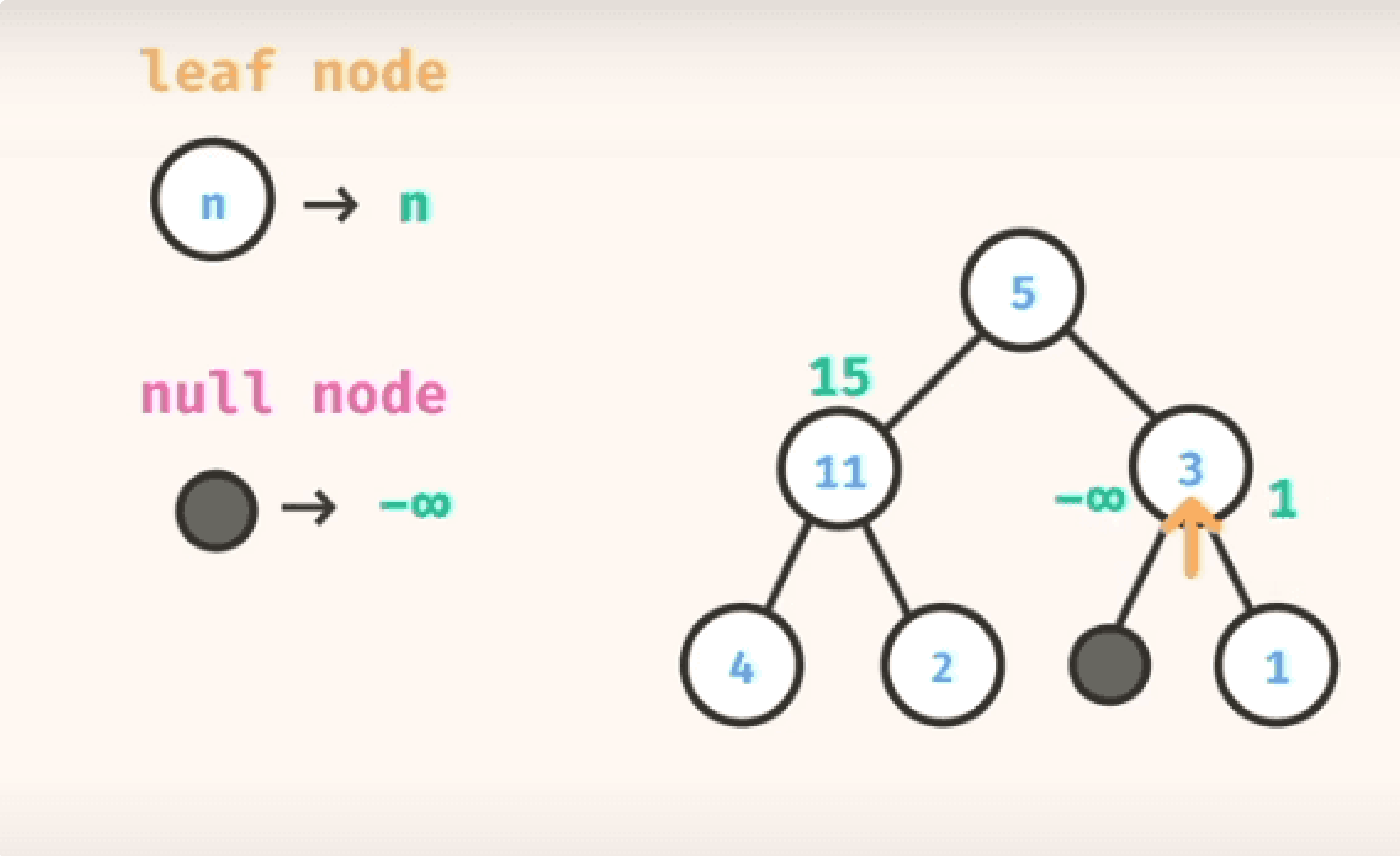
finding the max of the path-sums problem, path finding

-> for path finding recursive is the best solution



leaf node
n
n
5
11
3
4
2
1
4
2
1


leaf node
n
n
5
15
11
3
1
4
2
1



the game is to take the largest among children

-> time comp -> O(n)

-> Space comp -> O(n)

Code:

```
const maxPathSum = (root) => {
  if (root === null) return -Infinity;
  if (root.left === null && root.right === null) return root.val;
  const maxChildPathSum = Math.max(maxPathSum(root.left), maxPathSum(root.right));
  return root.val + maxChildPathSum;
};
```