Red Black Trees

Sai-Keung Wong (黃世強)

National Yang Ming Chiao Tung University

Properties

Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

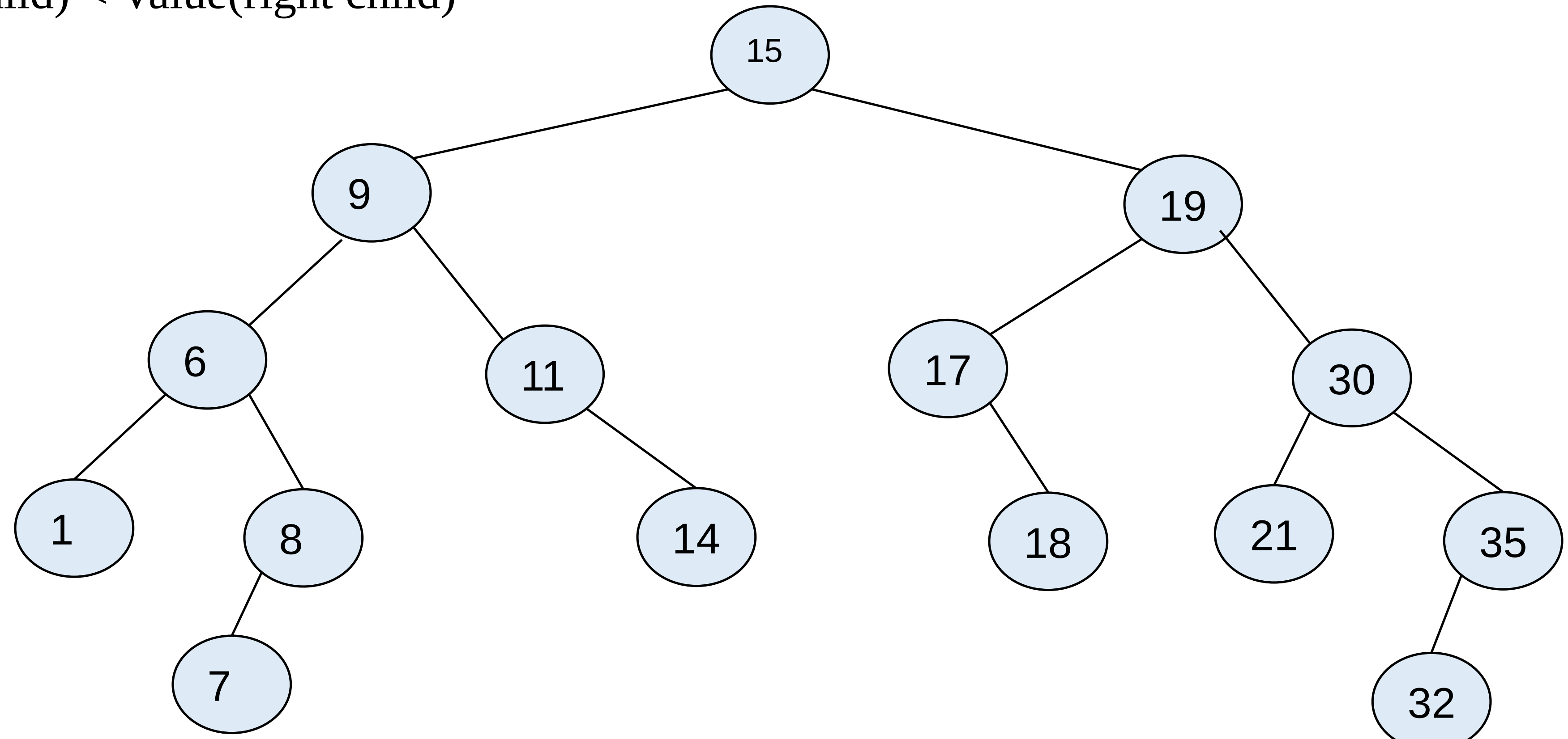
Properties

Colored Nodes Definition

- Binary search tree.
- Each node is colored red or black.
- **Root** and **all external nodes** are **black**.
- Two consecutive red nodes are not allowed along a path from the root to an external node.
- All root-to-external-node paths have the same number of black nodes

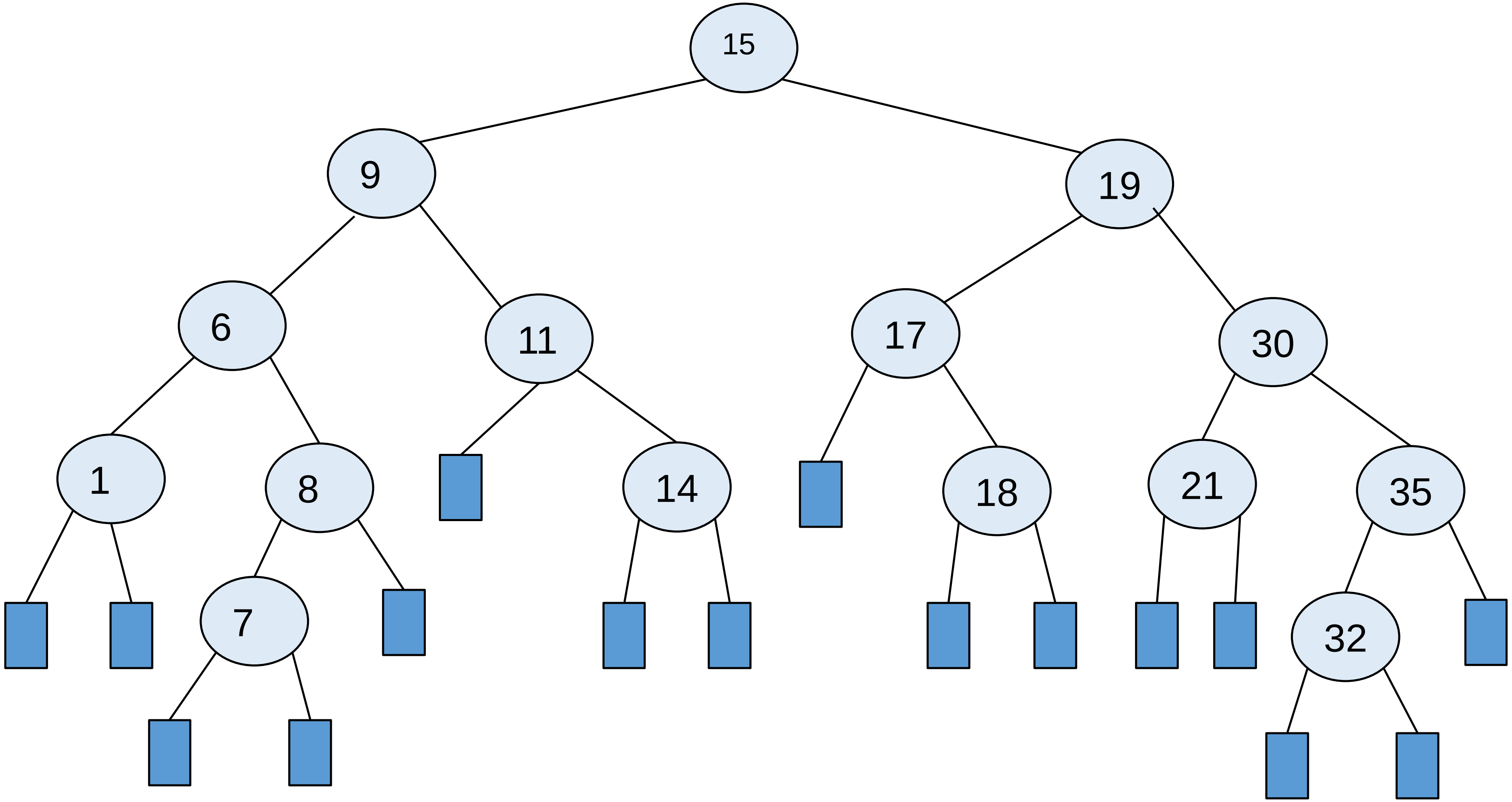
Search Trees

- Each node stores a key (or value)
- Value(left child) <= root < Value(right child)
- Value(left child) < Value(right child)



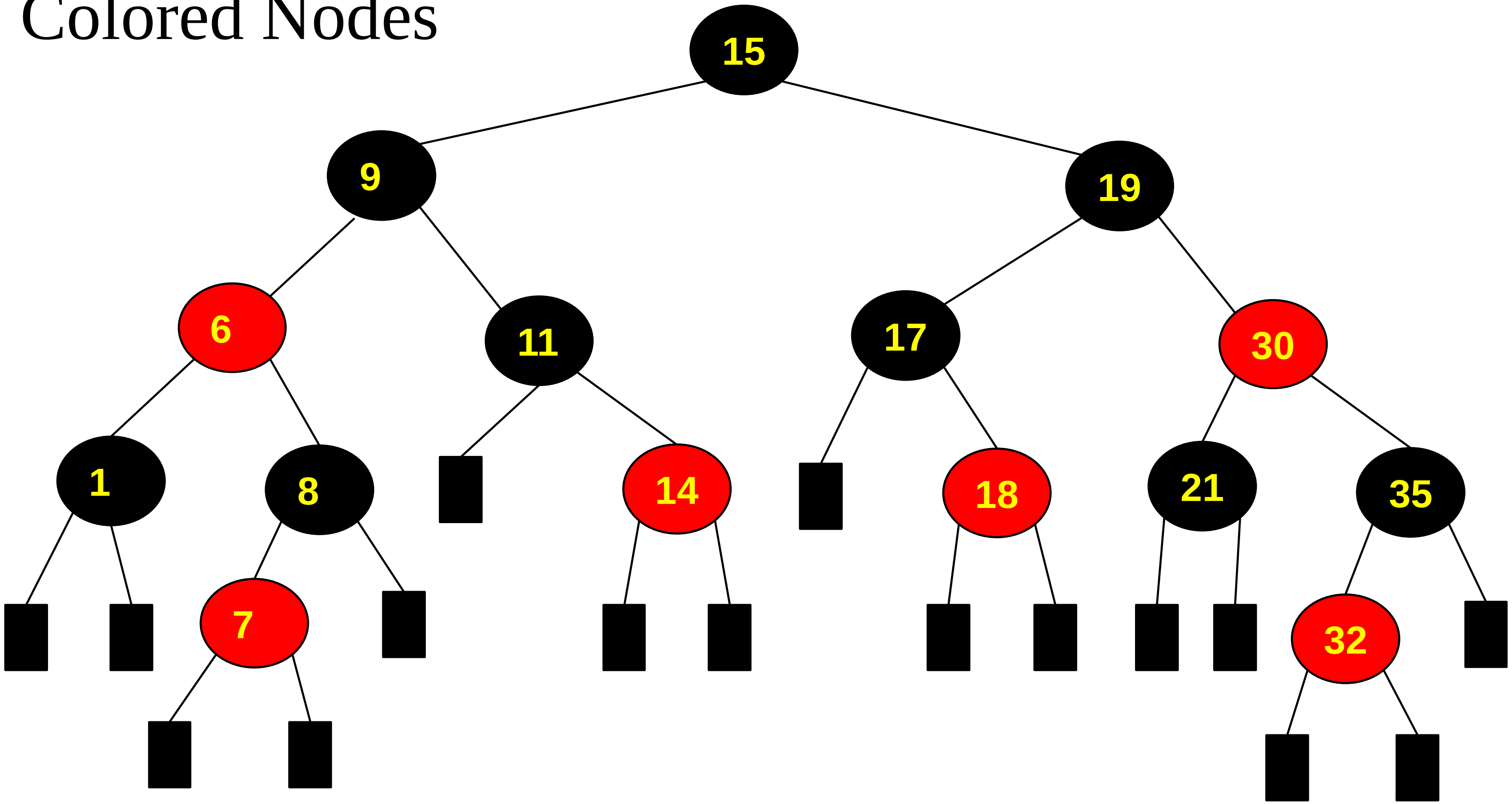
Extended Search Trees

- Add the left and right children to each leaf.
- These two extra nodes are called extended nodes.



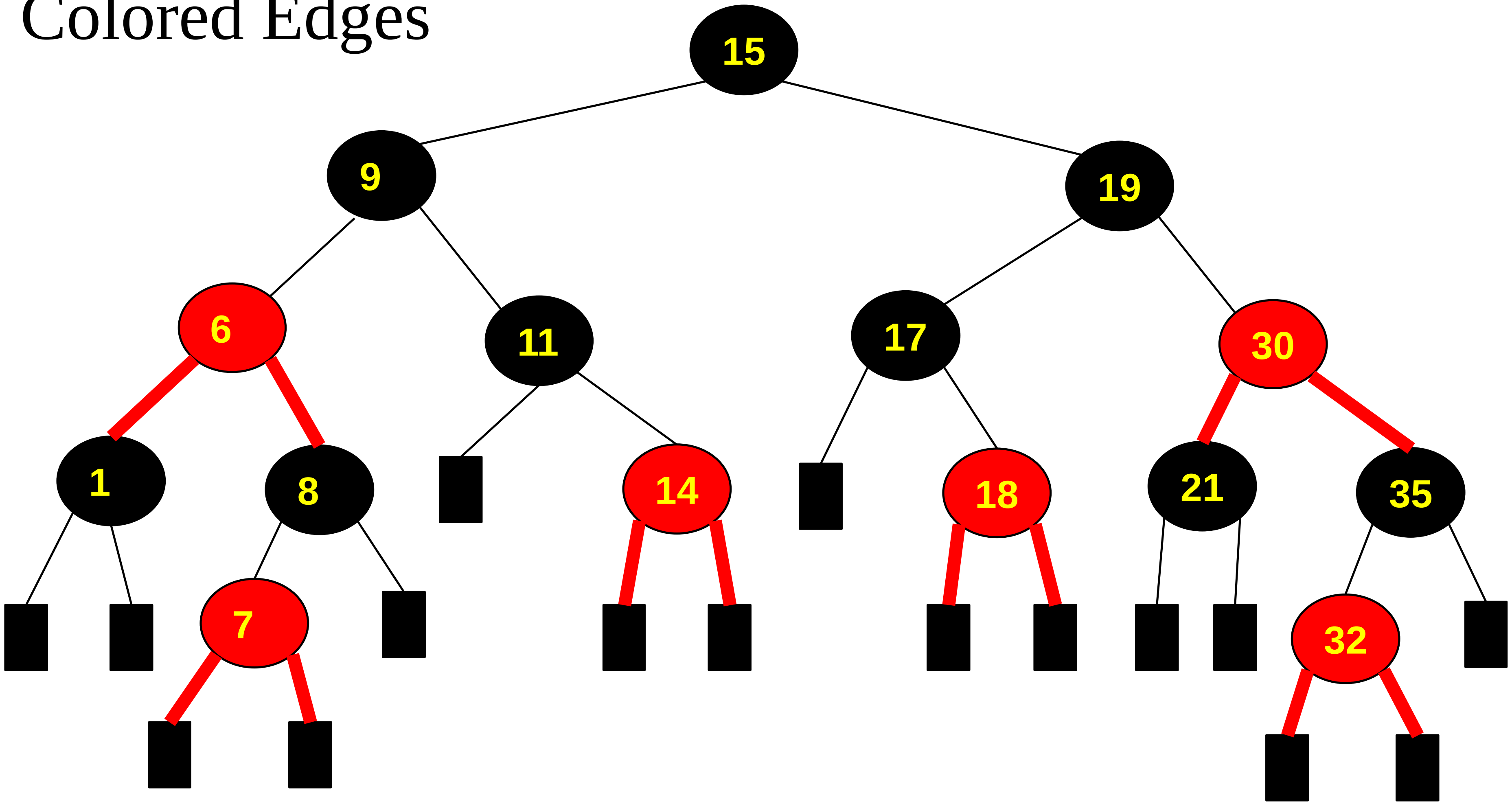
Red Black Trees
Search Trees

Colored Nodes



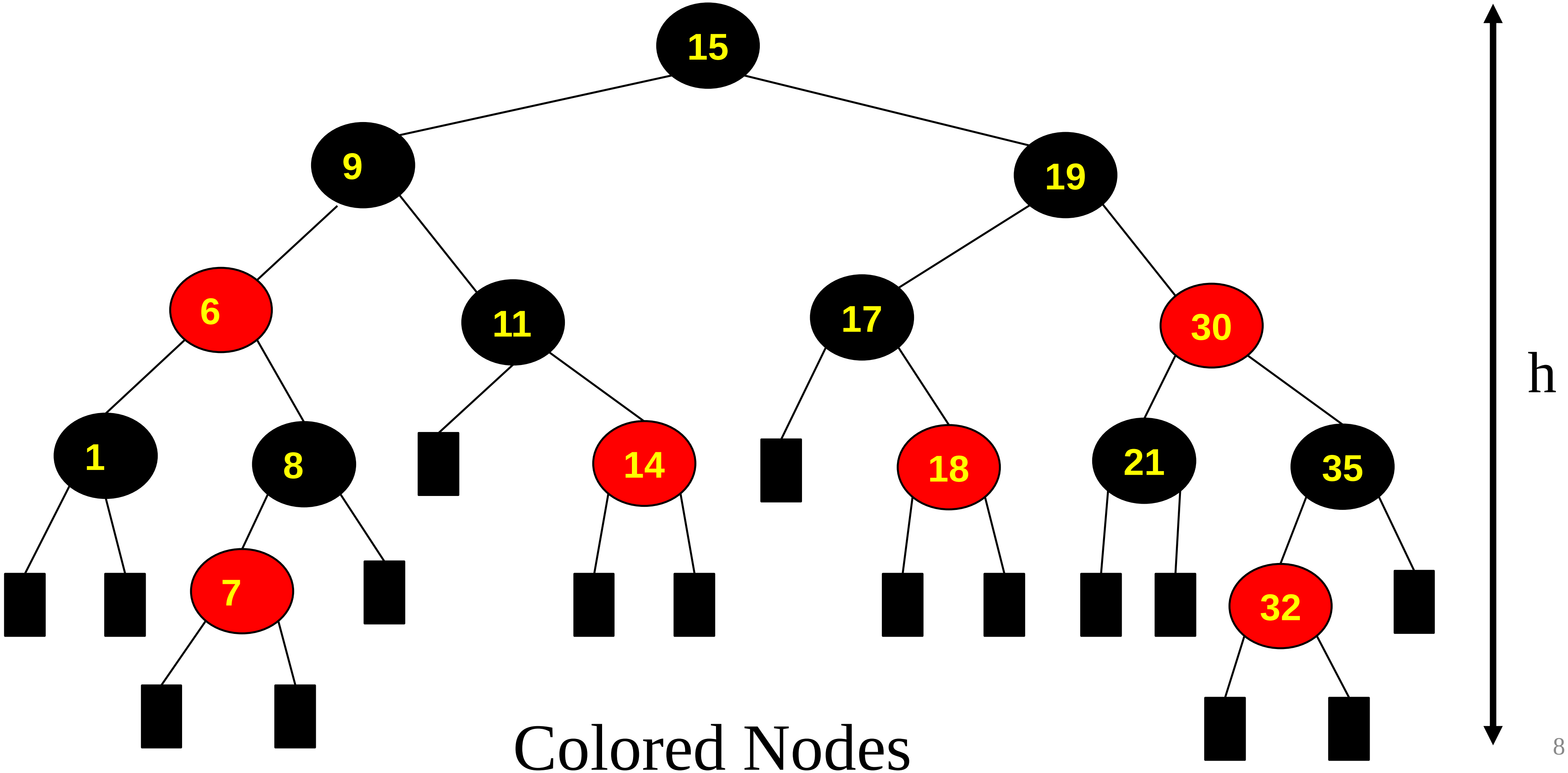
Red Black Trees
Search Trees

Colored Edges



Properties

- Let h be the height of a red black tree that has n internal nodes.
- $\log_2(n+1) \le h \le 2\log_2(n+1)$.



Colored Nodes

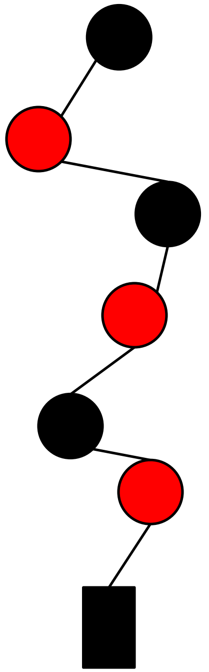
Properties: Collapsed Trees

- Start with a red black tree whose height is h.
- Collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4.
- The height of the collapsed tree is h’ >= h/2.

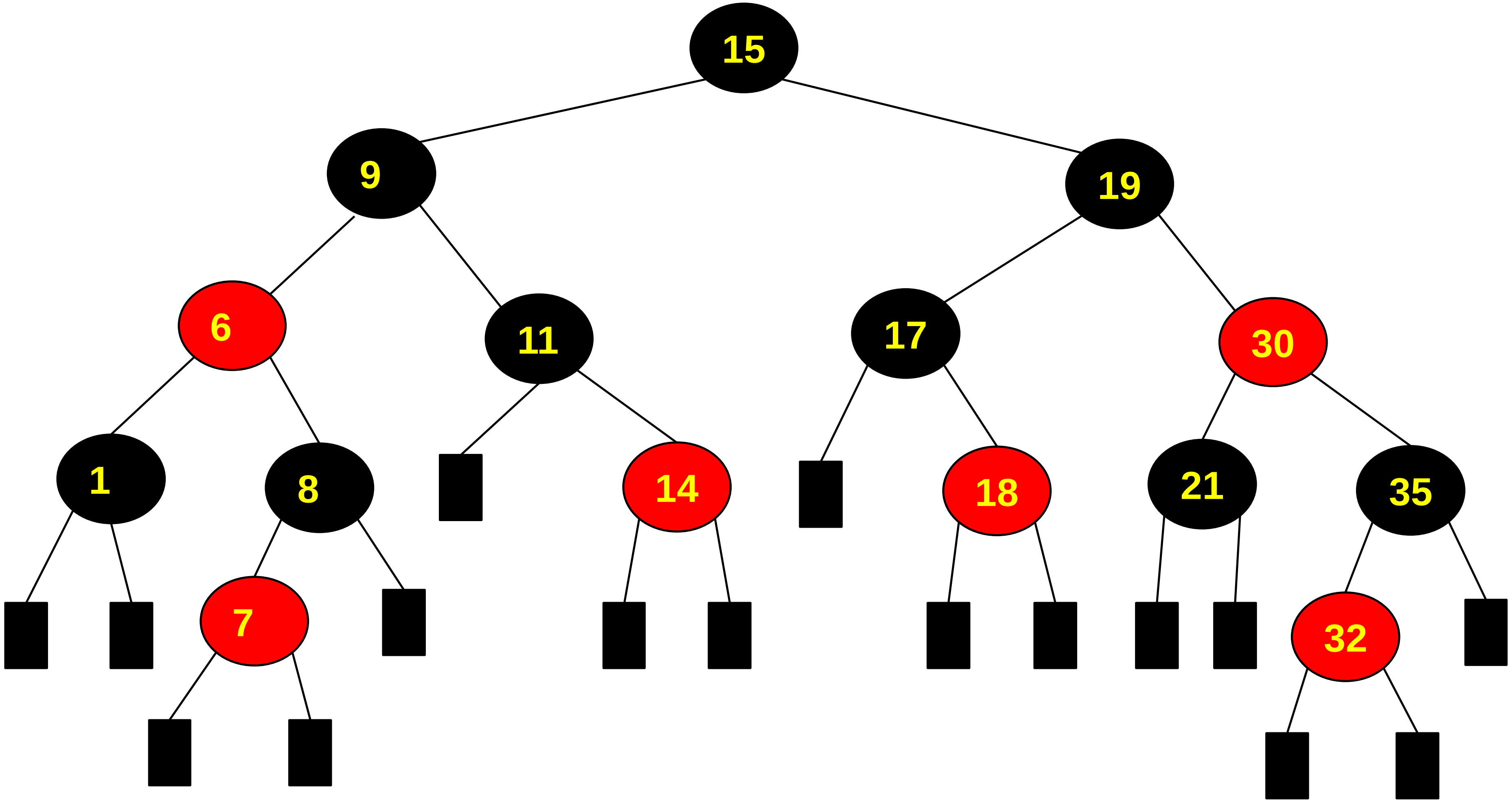
  There are two extreme cases:

  1) all nodes are black

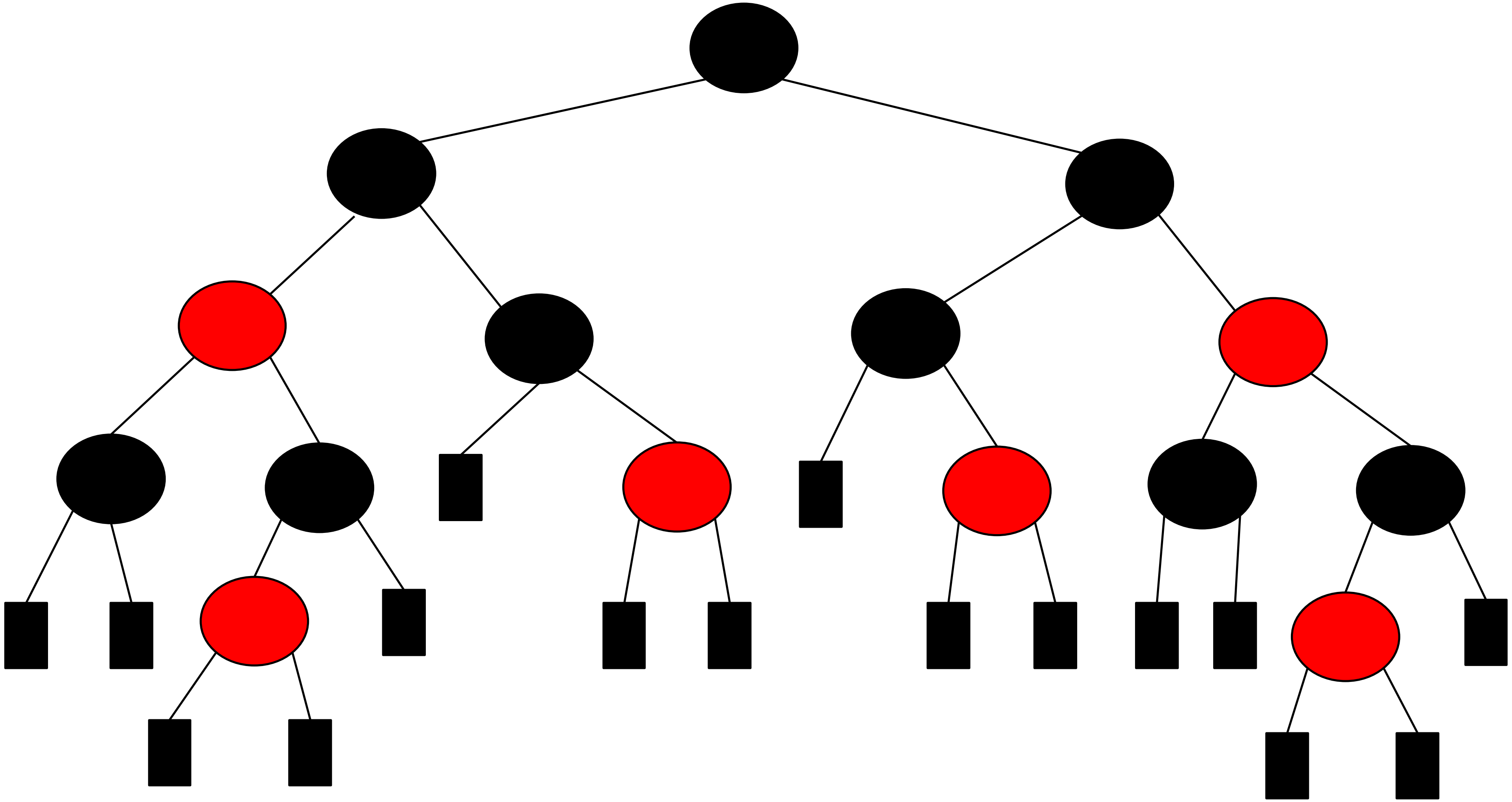
  2) Half of the nodes along all root-to-external-node paths are red. (not include the external nodes)
- All external nodes are at the same level.



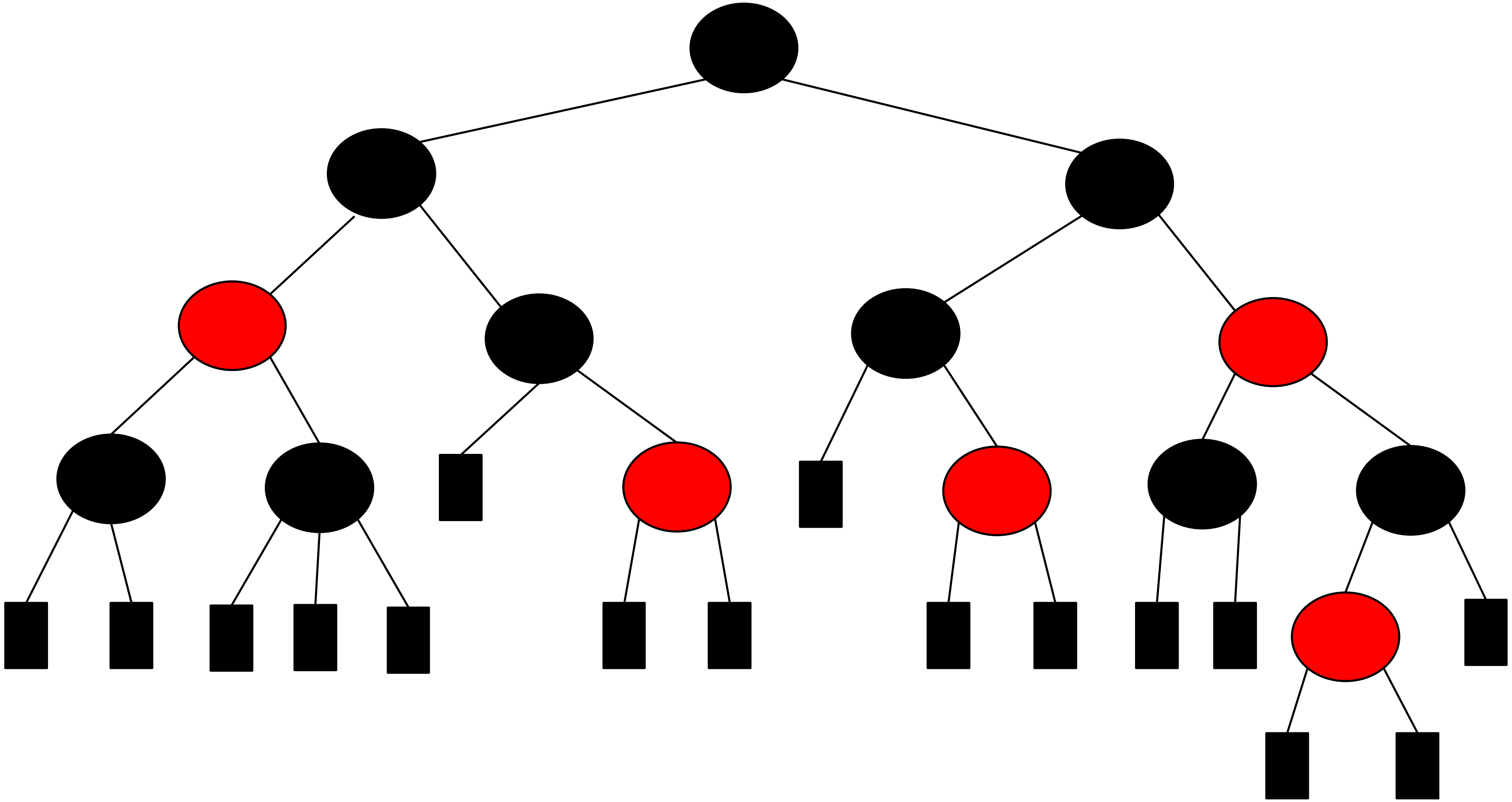
Properties
Collapsed Trees



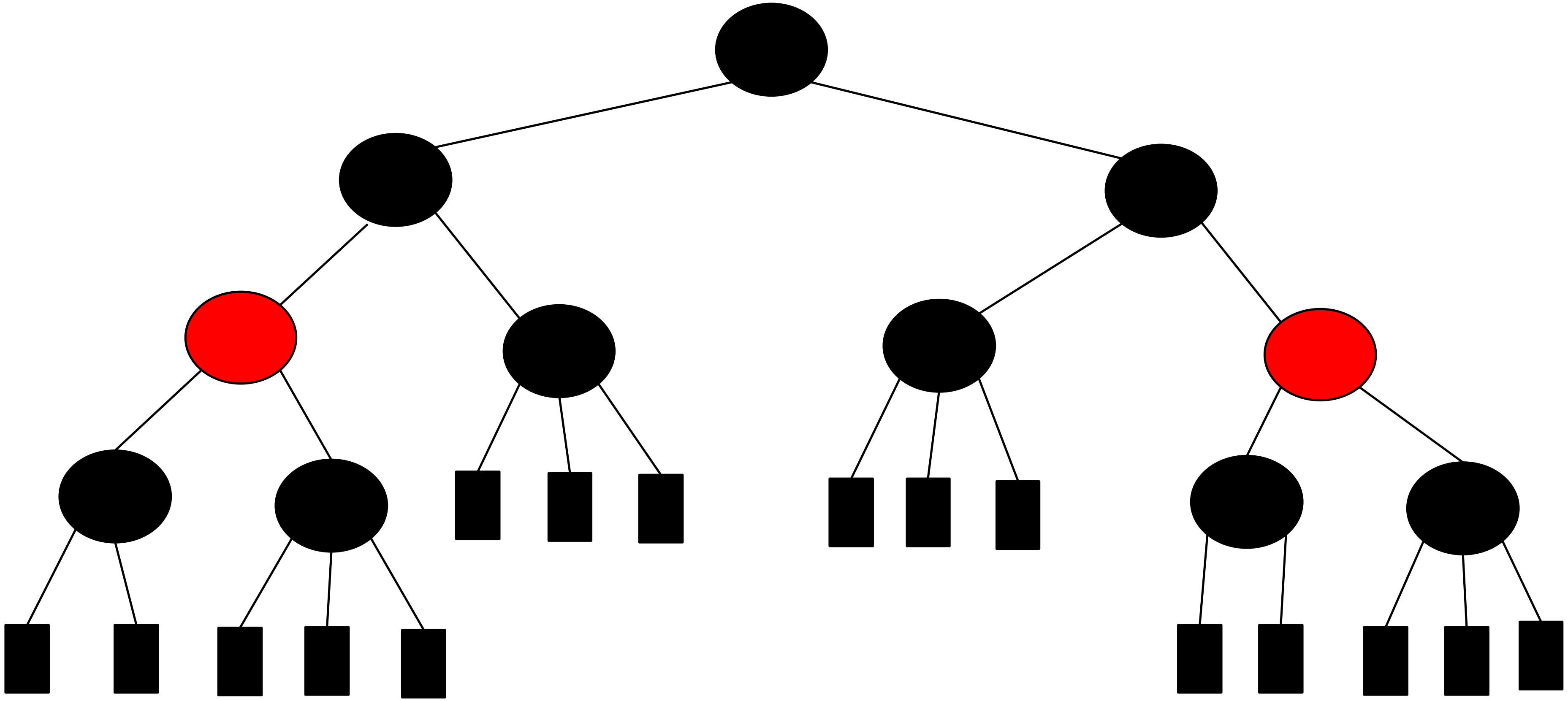
Properties
Collapsed Trees



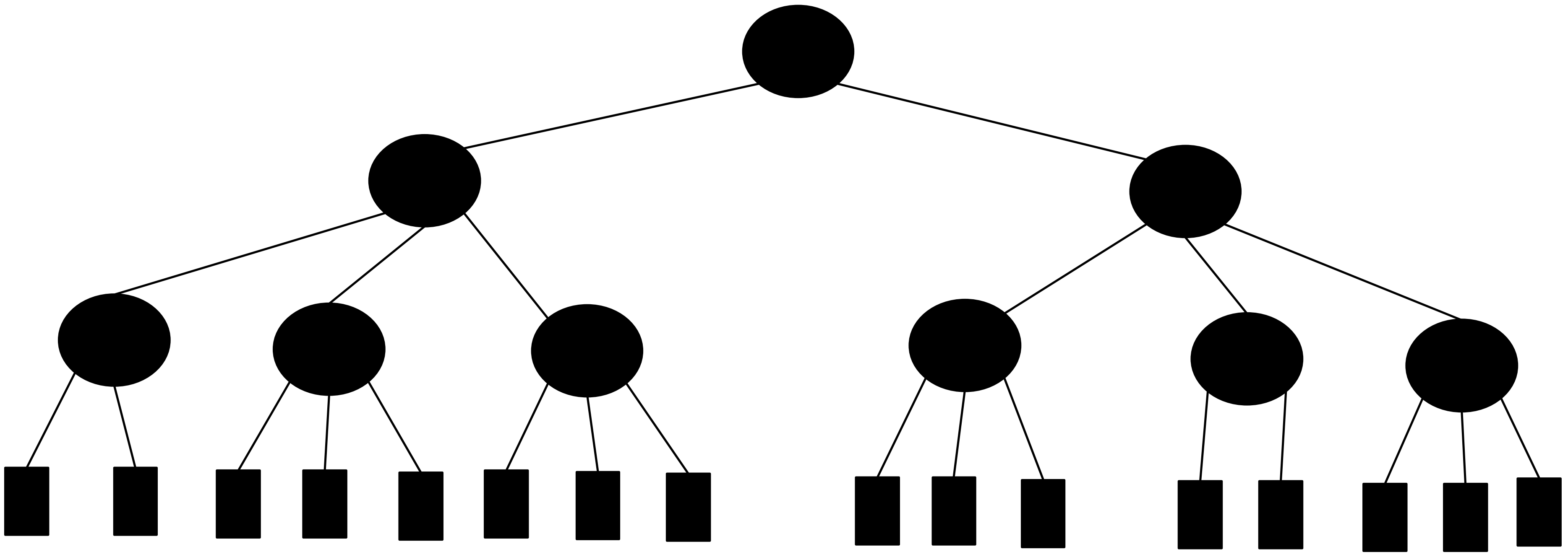
Properties
Collapsed Trees



Properties
Collapsed Trees

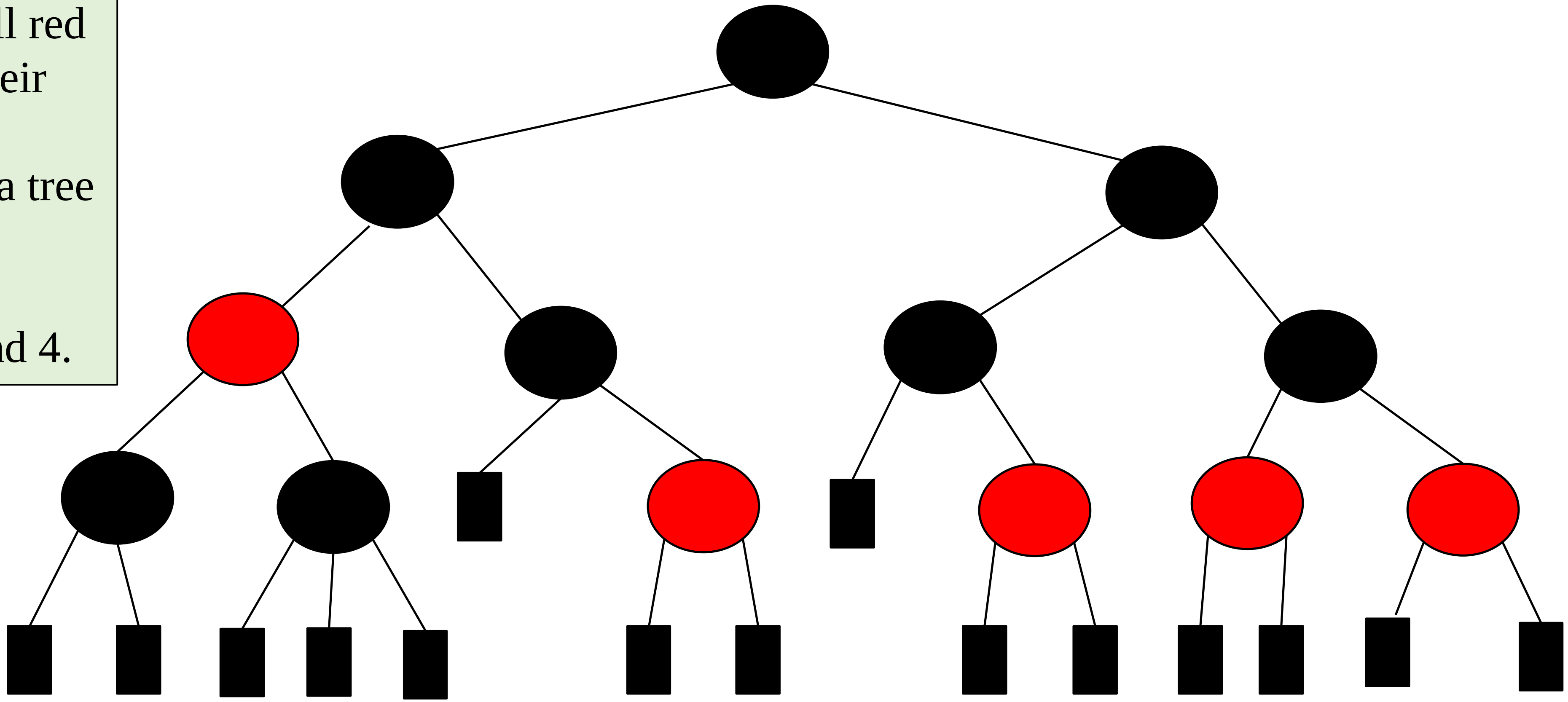


Properties
Collapsed Trees



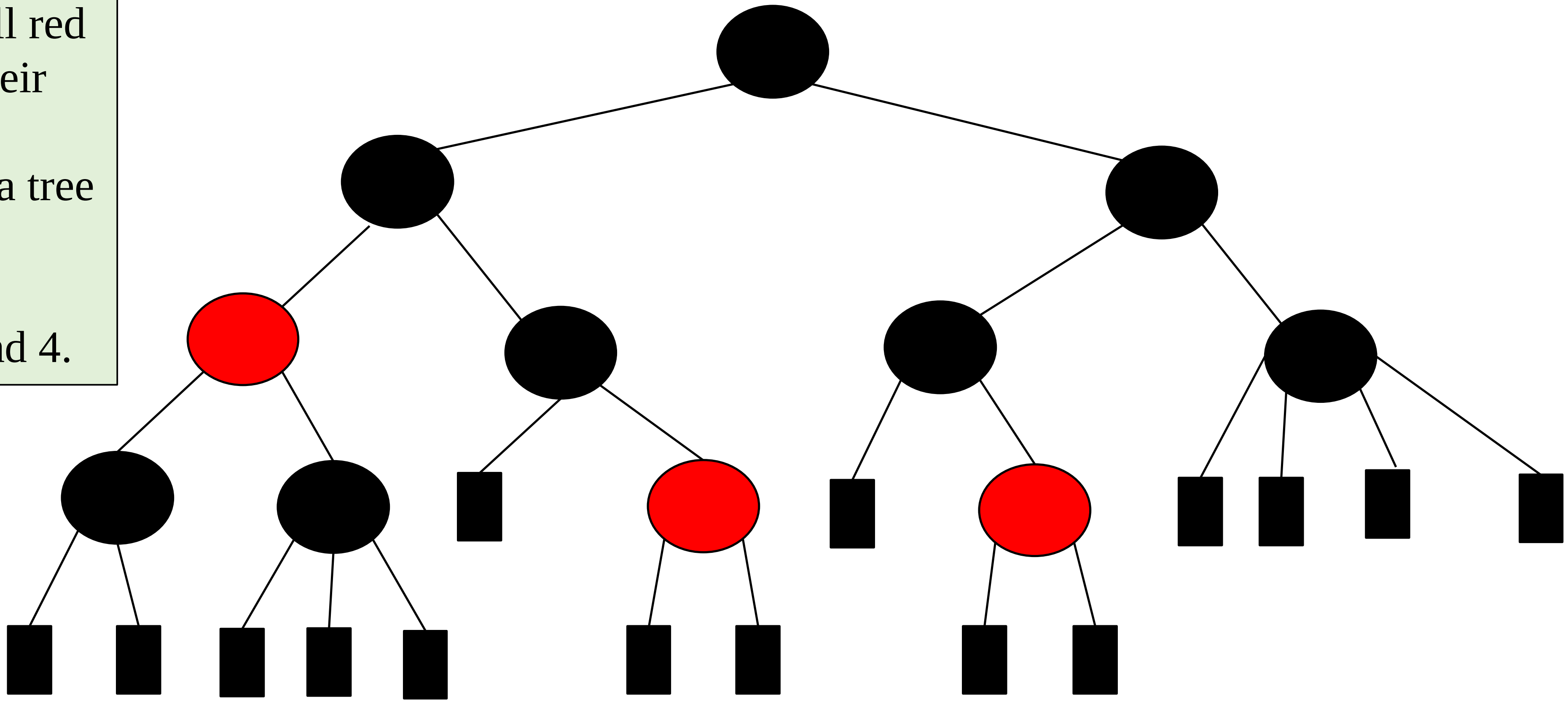
Properties
Collapsed Trees

- Collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4.



Properties
Collapsed Trees

- Collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4.



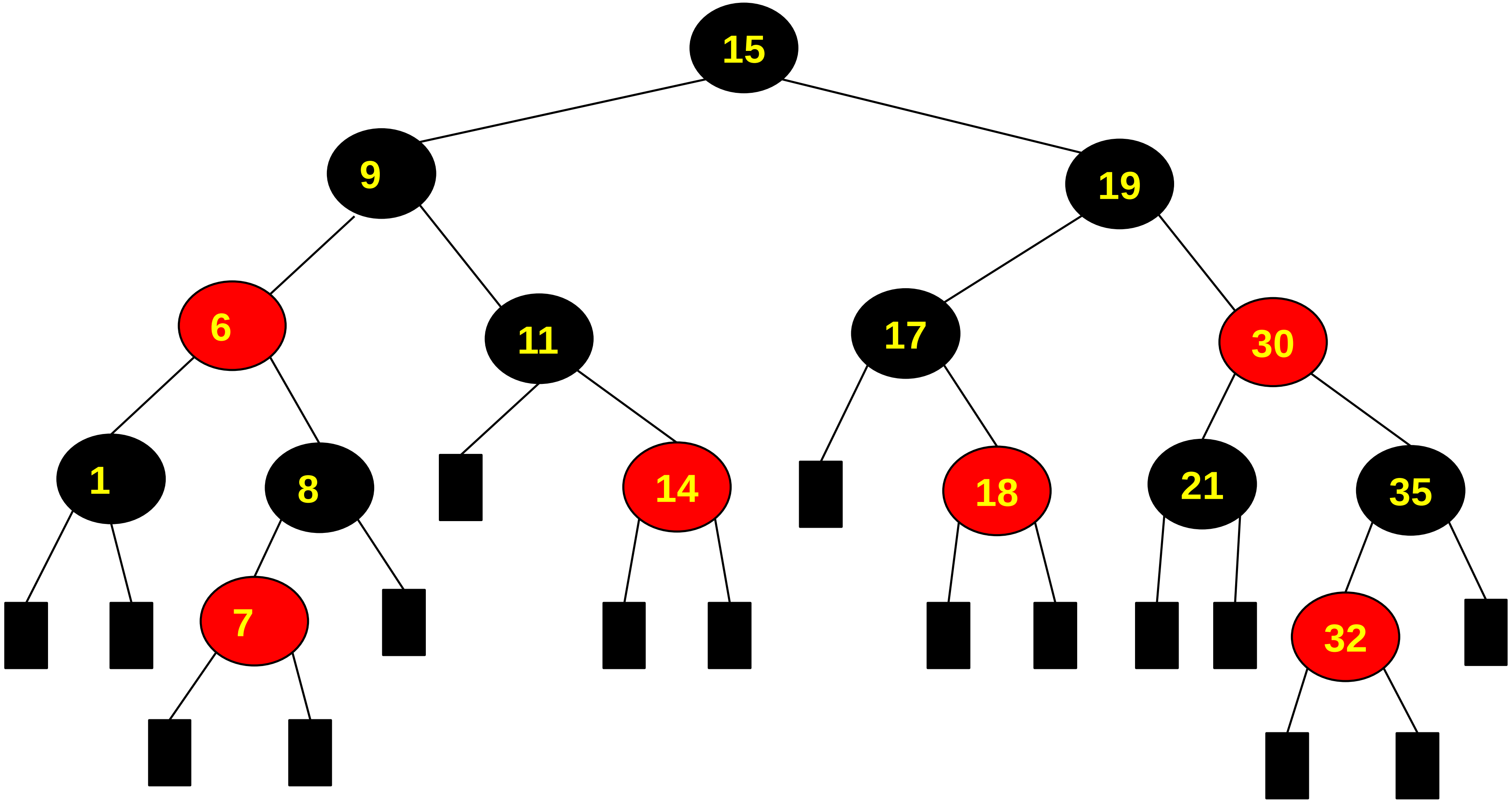
Properties

- Let h’ (>= h/2) be the height of the collapsed tree.
- In worst-case, all internal nodes of collapsed tree have degree 2.
- Number of internal nodes in collapsed tree >= $2^{h'}$-1.
- So, n >= $2^{h'}$-1
- So, h <= 2 $\log_2$ (n + 1)

Operations

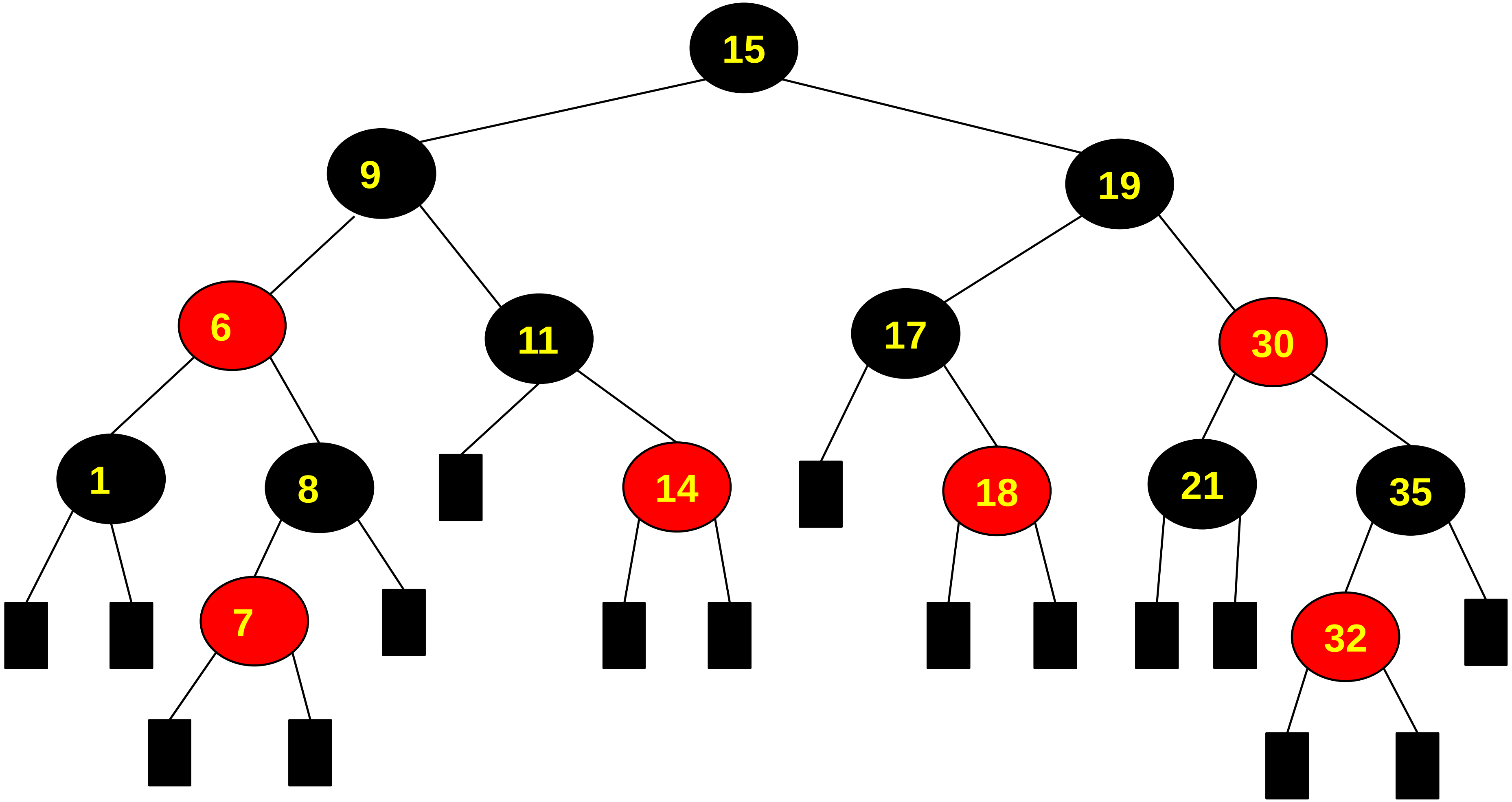
- Node insertion
- Node deletion
- Need to maintain the properties of red-black trees.

In-order traversal result



In-order traversal result

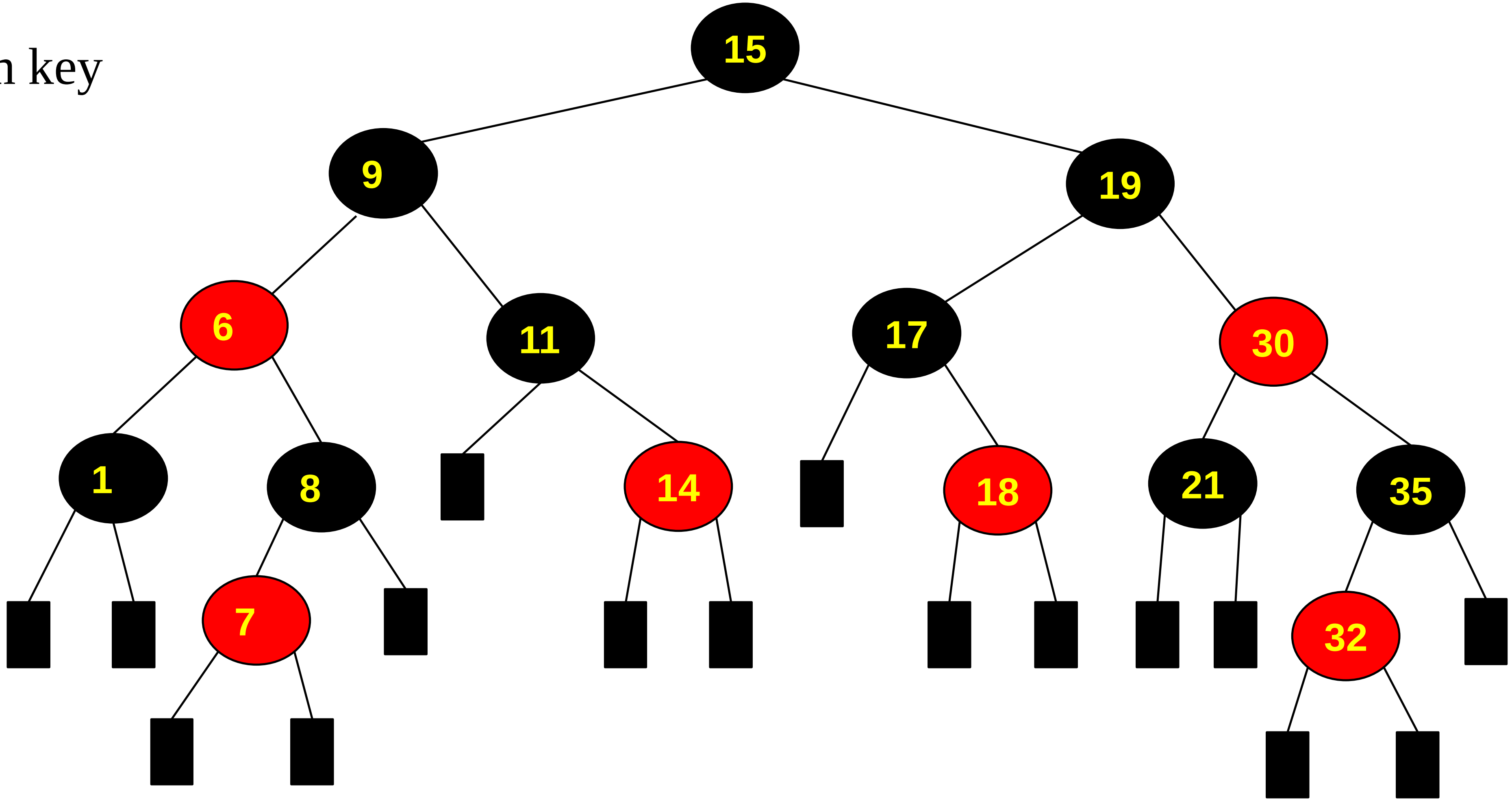
1, 6, 7, 8, 9, 11, 14, 15, 17, 18, 19, 21, 30, 32, 35



In-order traversal result

1, 6, 7, 8, 9, 11, 14, 15, 17, 18, 19, 21, 30, 32, 35

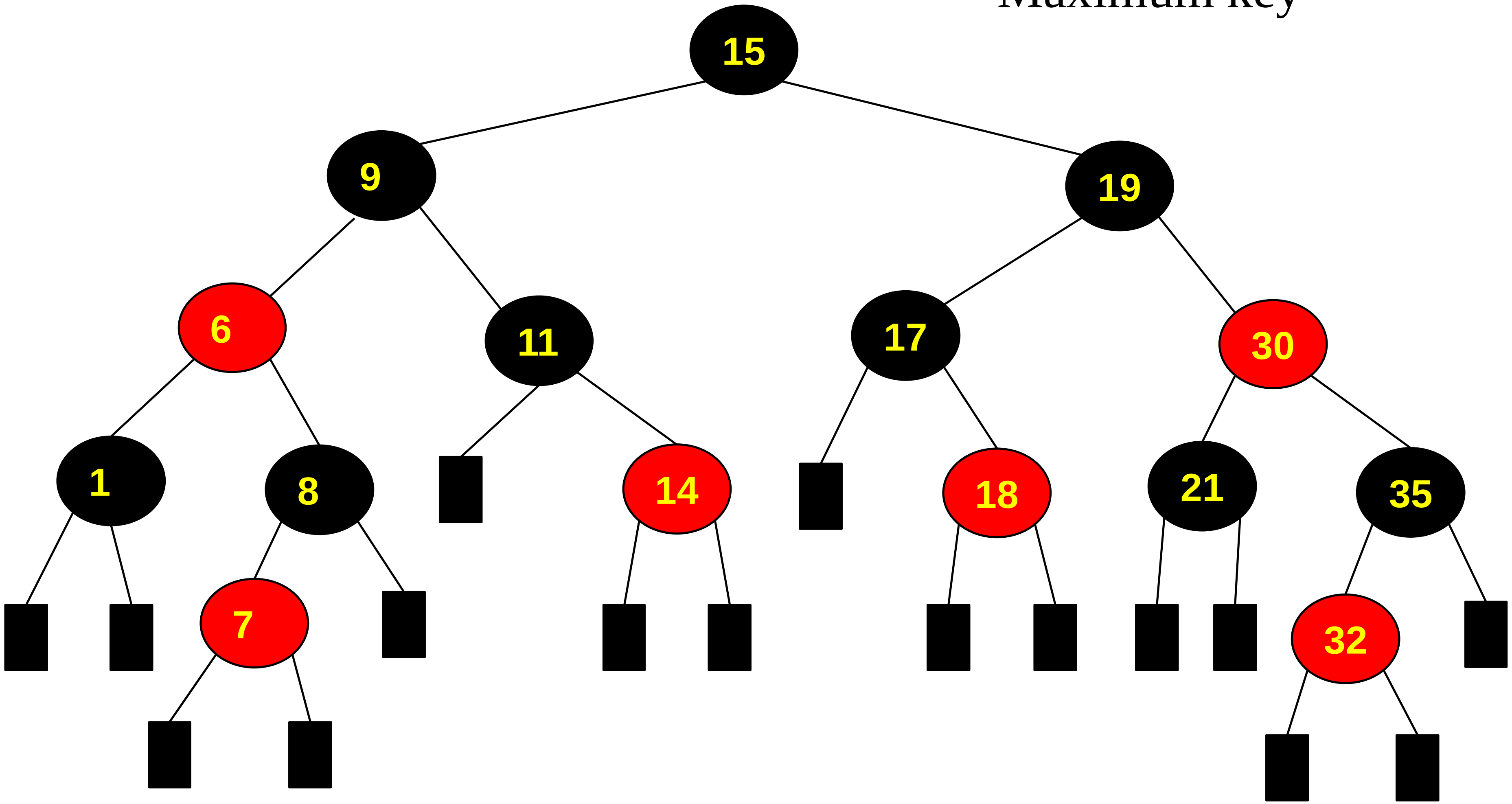
Minimum key



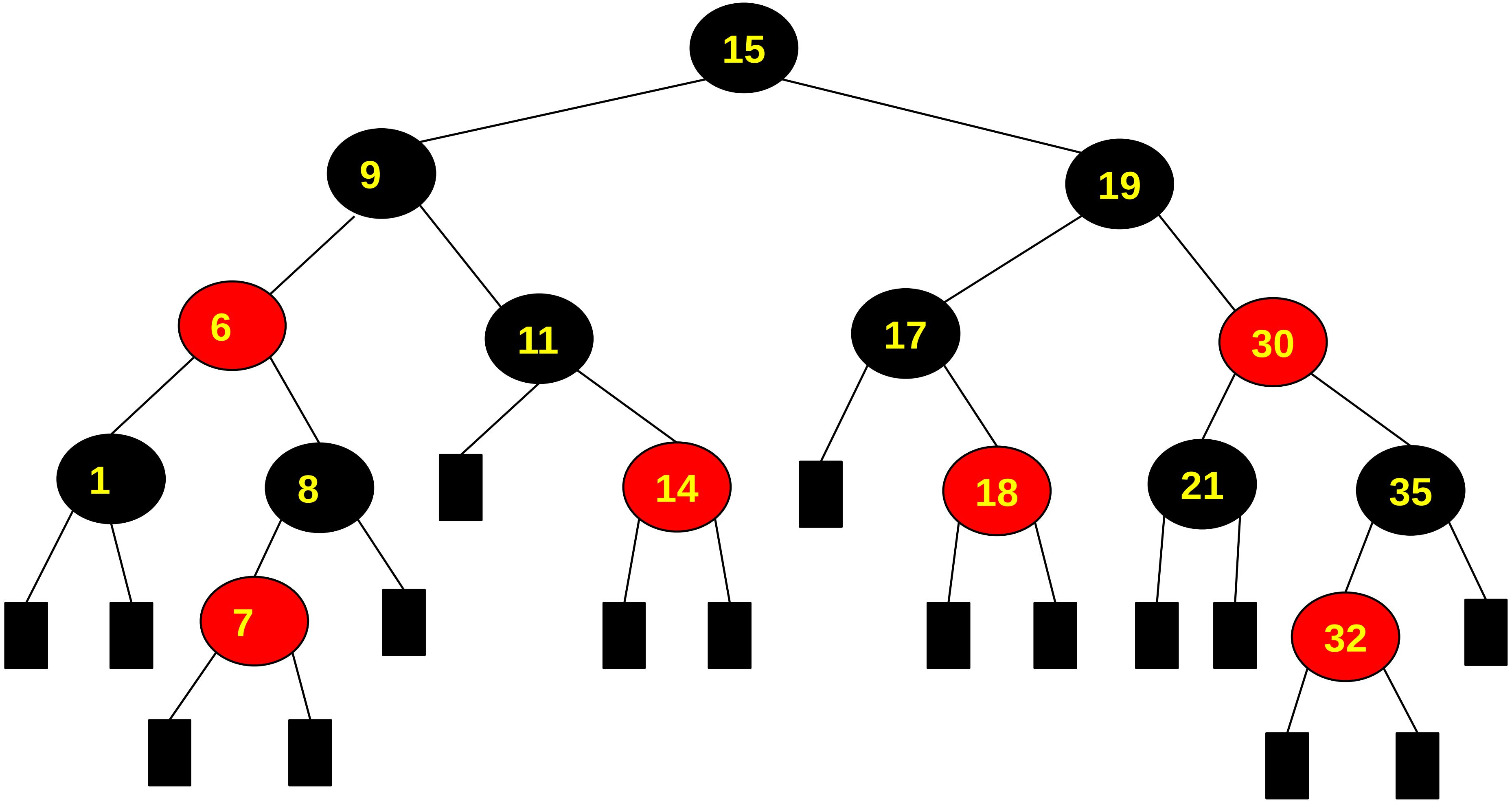
In-order traversal result

1, 6, 7, 8, 9, 11, 14, 15, 17, 18, 19, 21, 30, 32, 35

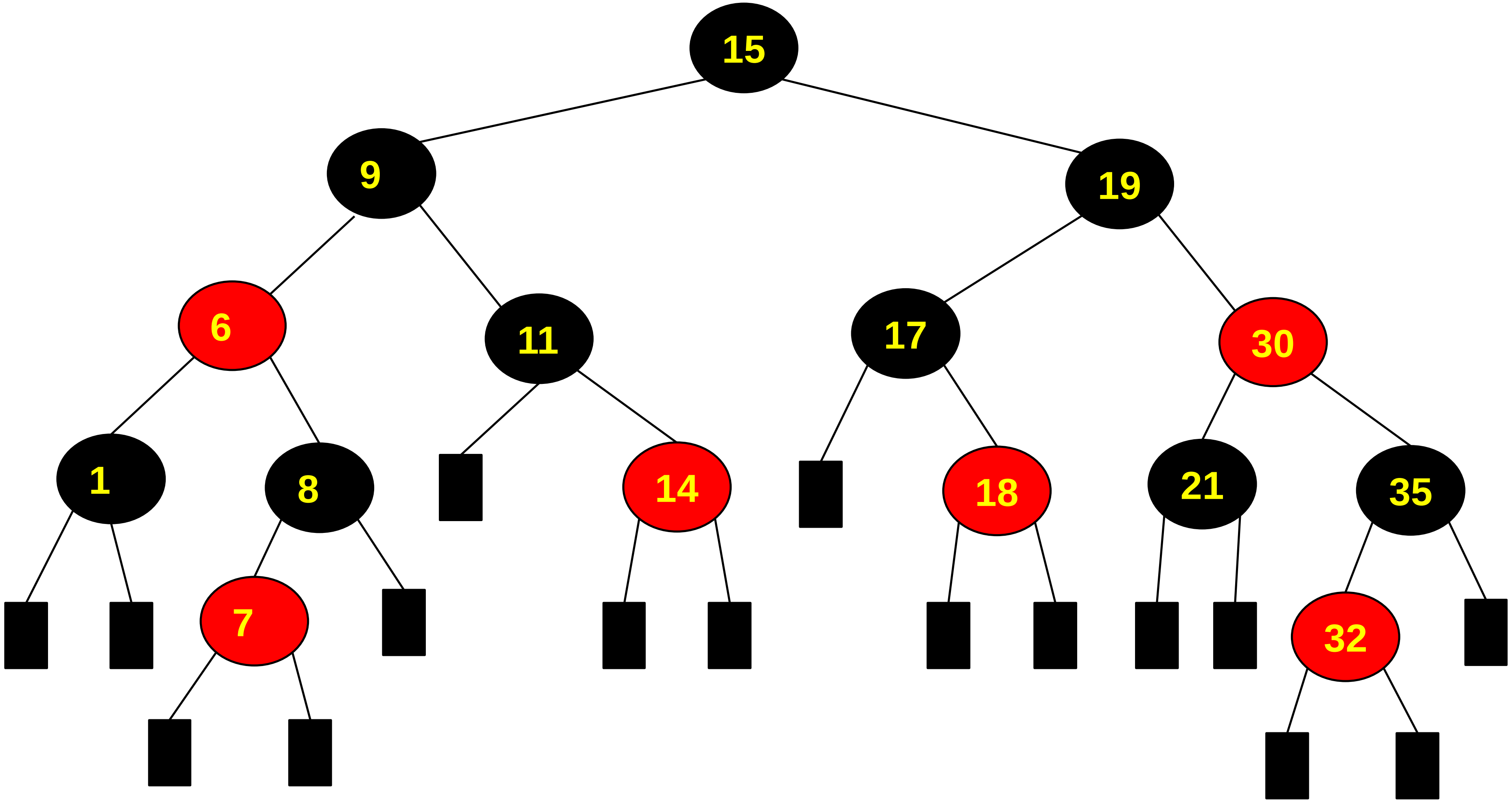
Maximum key



Delete a node?

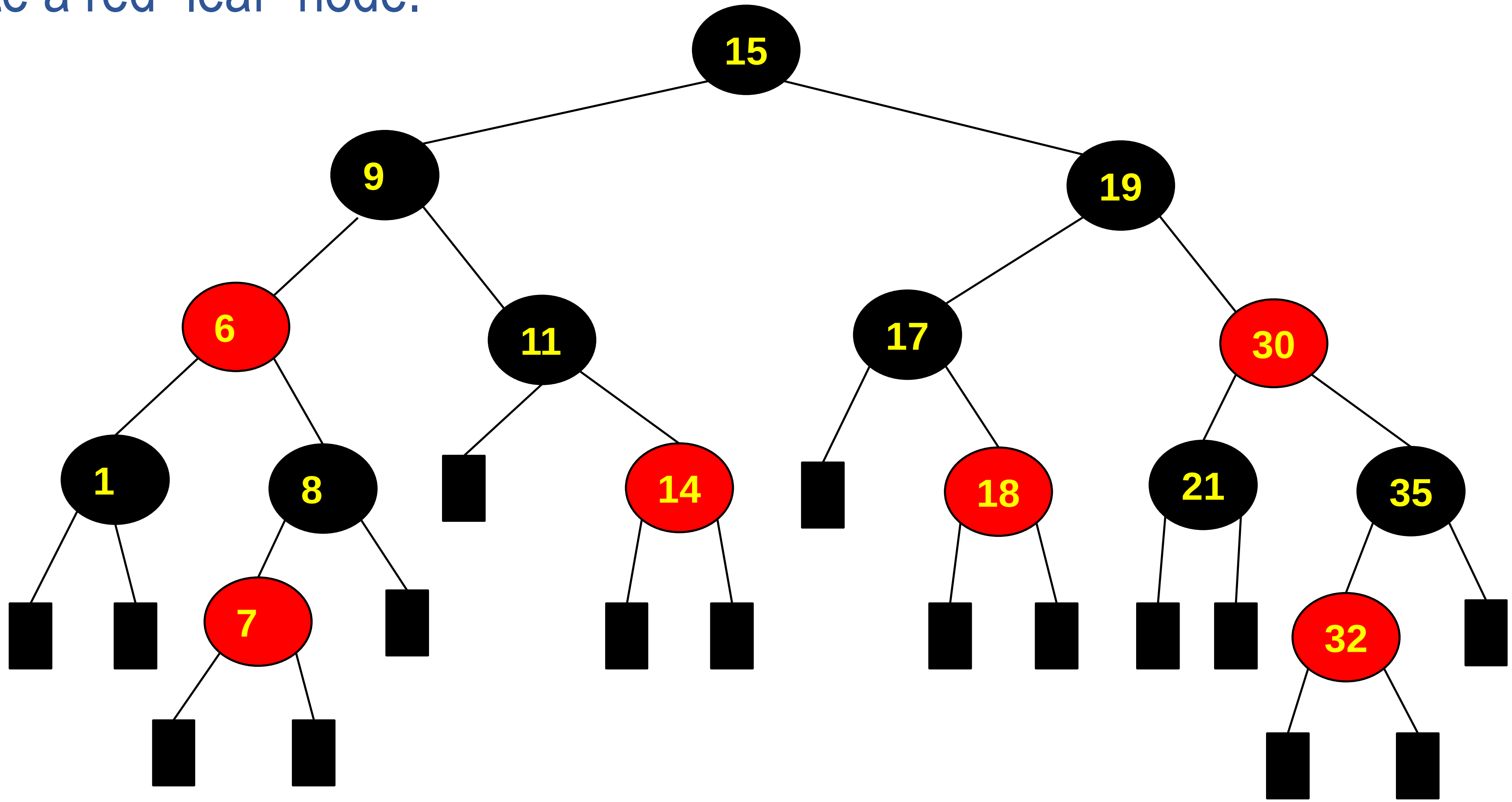


Delete a node? (1)
Delete 14



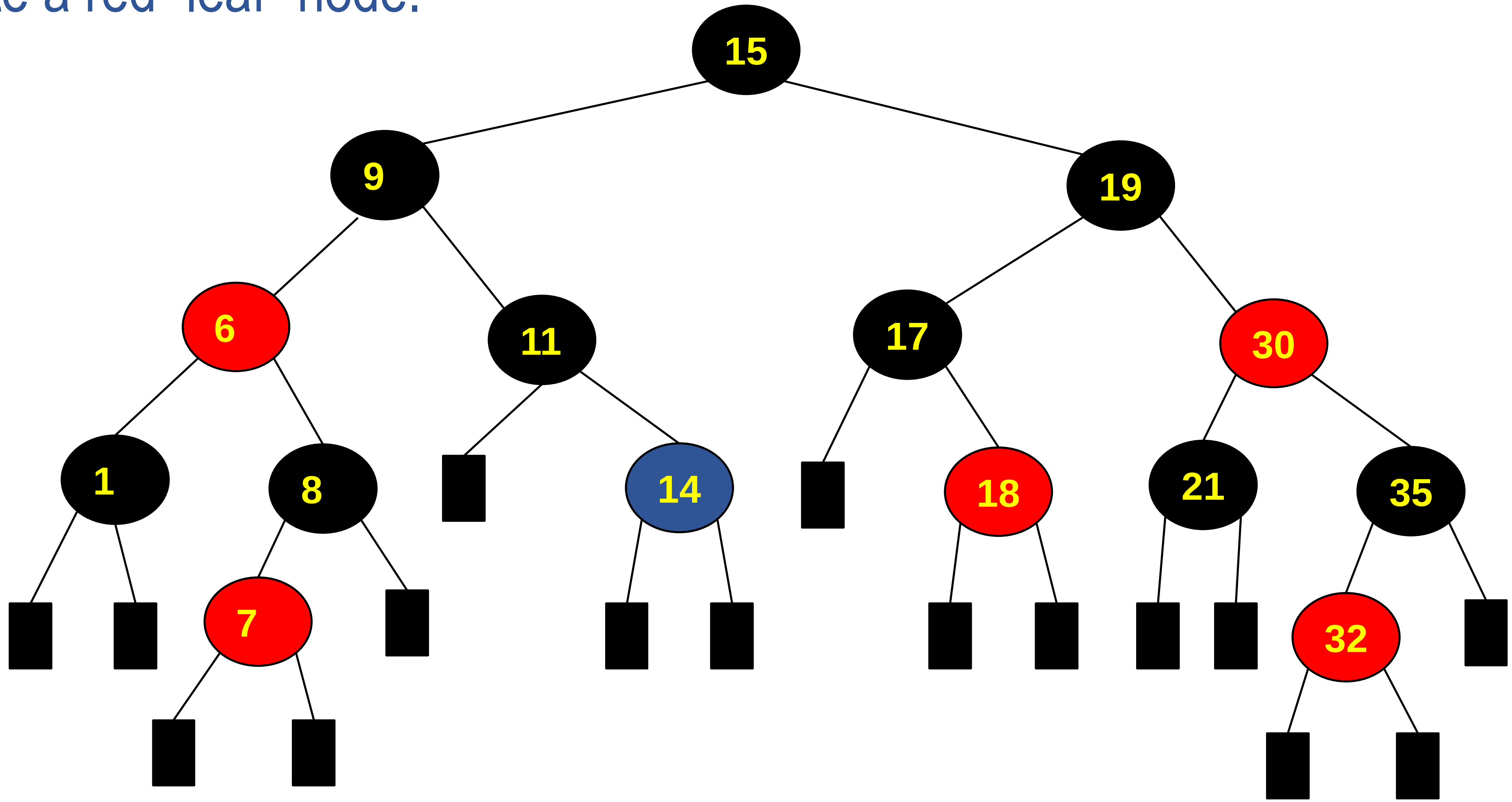
Delete a node? (1)
Delete 14

Delete a red “leaf” node.



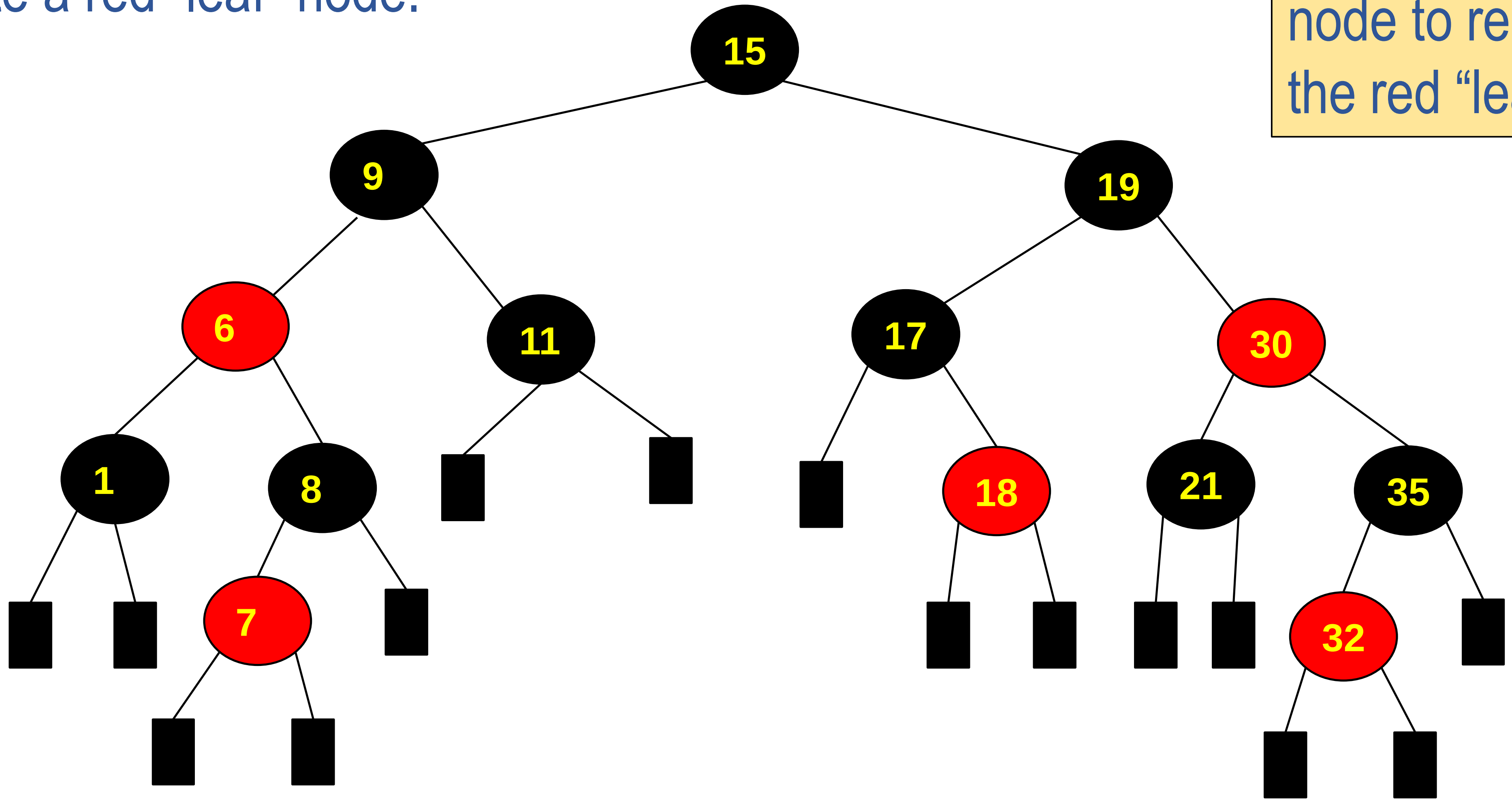
Delete a node? (2)
Delete 14

Delete a red "leaf" node.



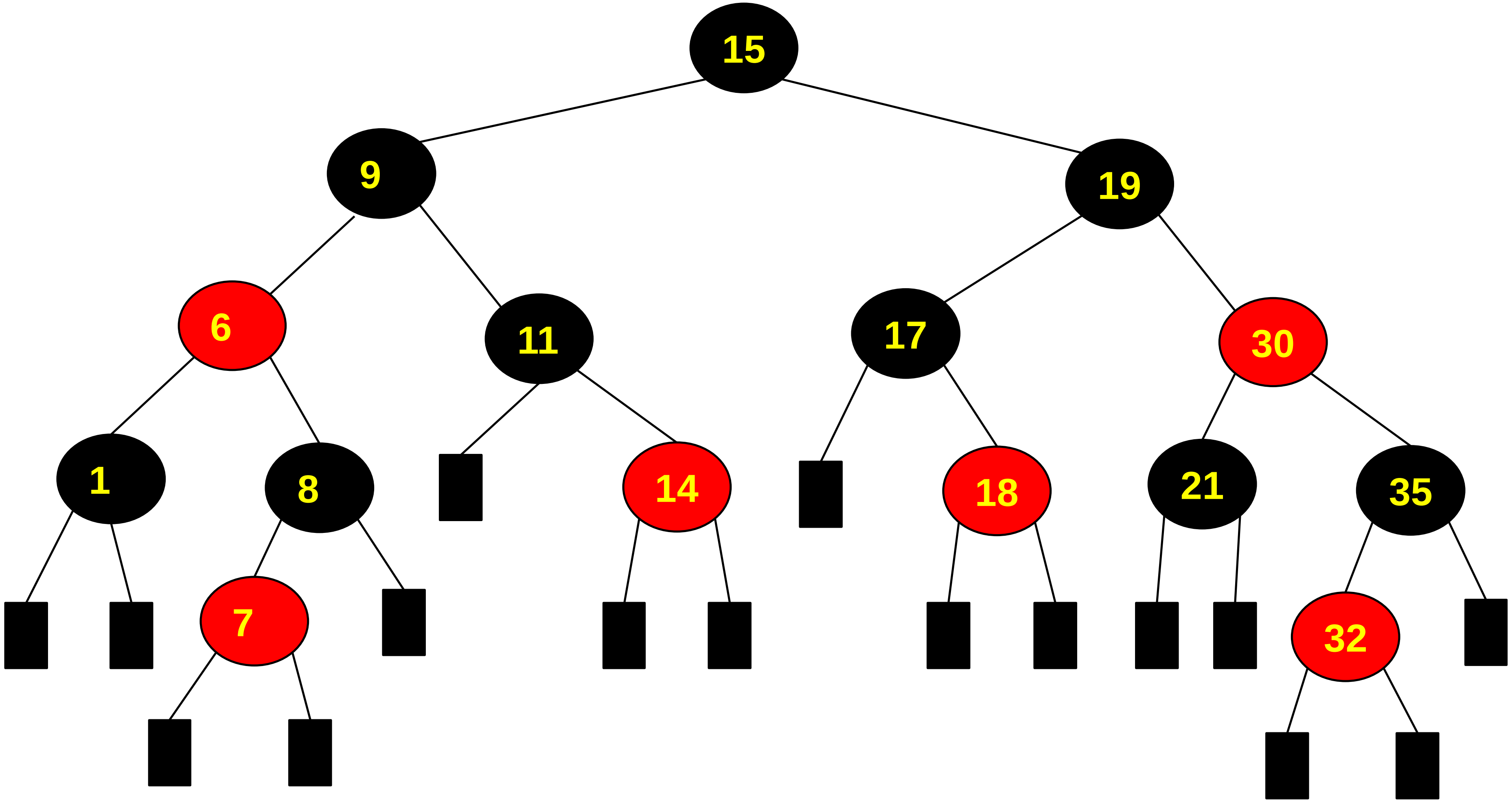
Delete a node? (3)
Delete 14

Delete a red "leaf" node.



Create an external node to replace with the red "leaf" node.

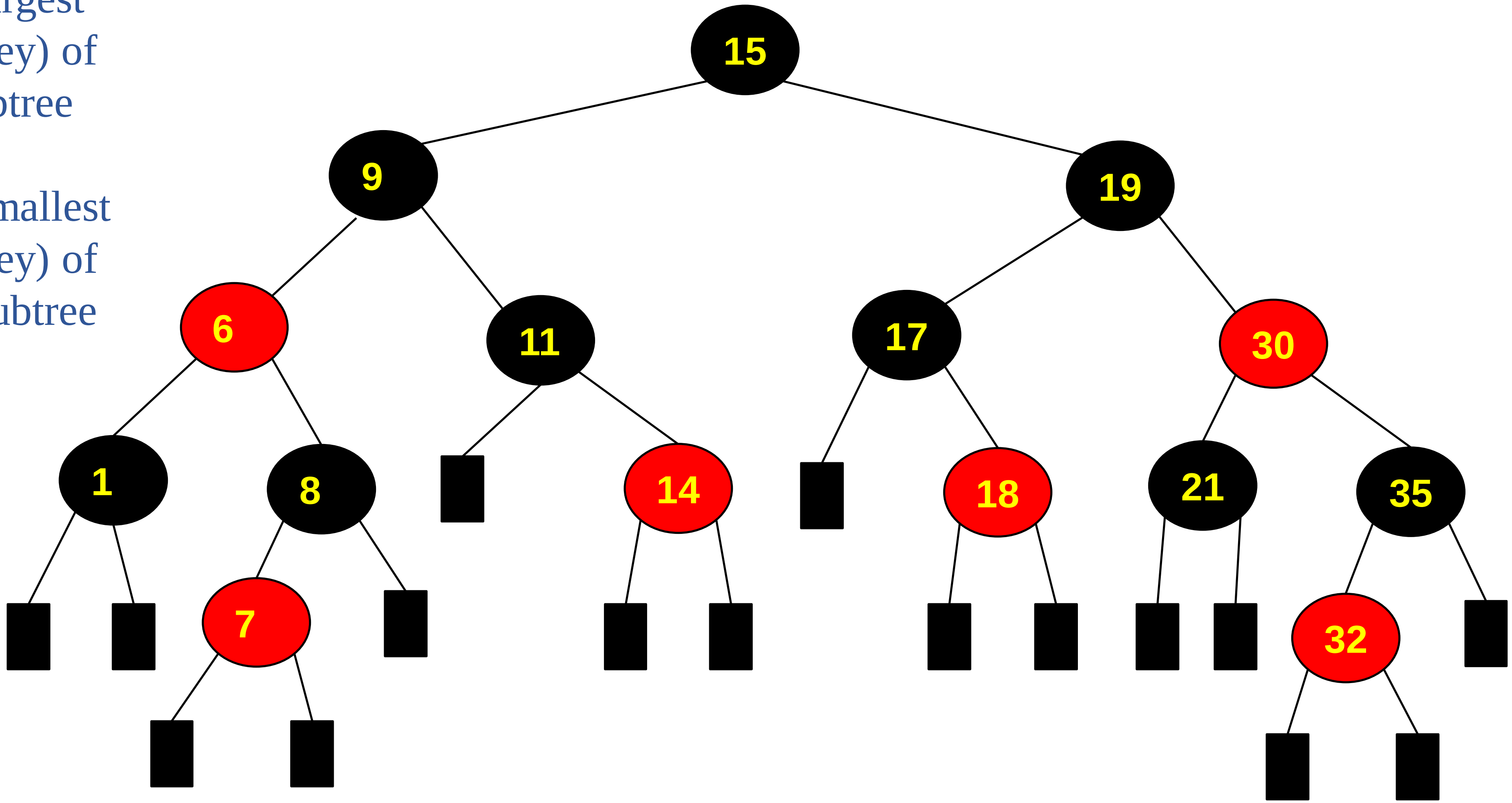
Delete a node? (1)
Delete 6



Delete a node? (1)
Delete 6

- Find the largest number (key) of the left subtree

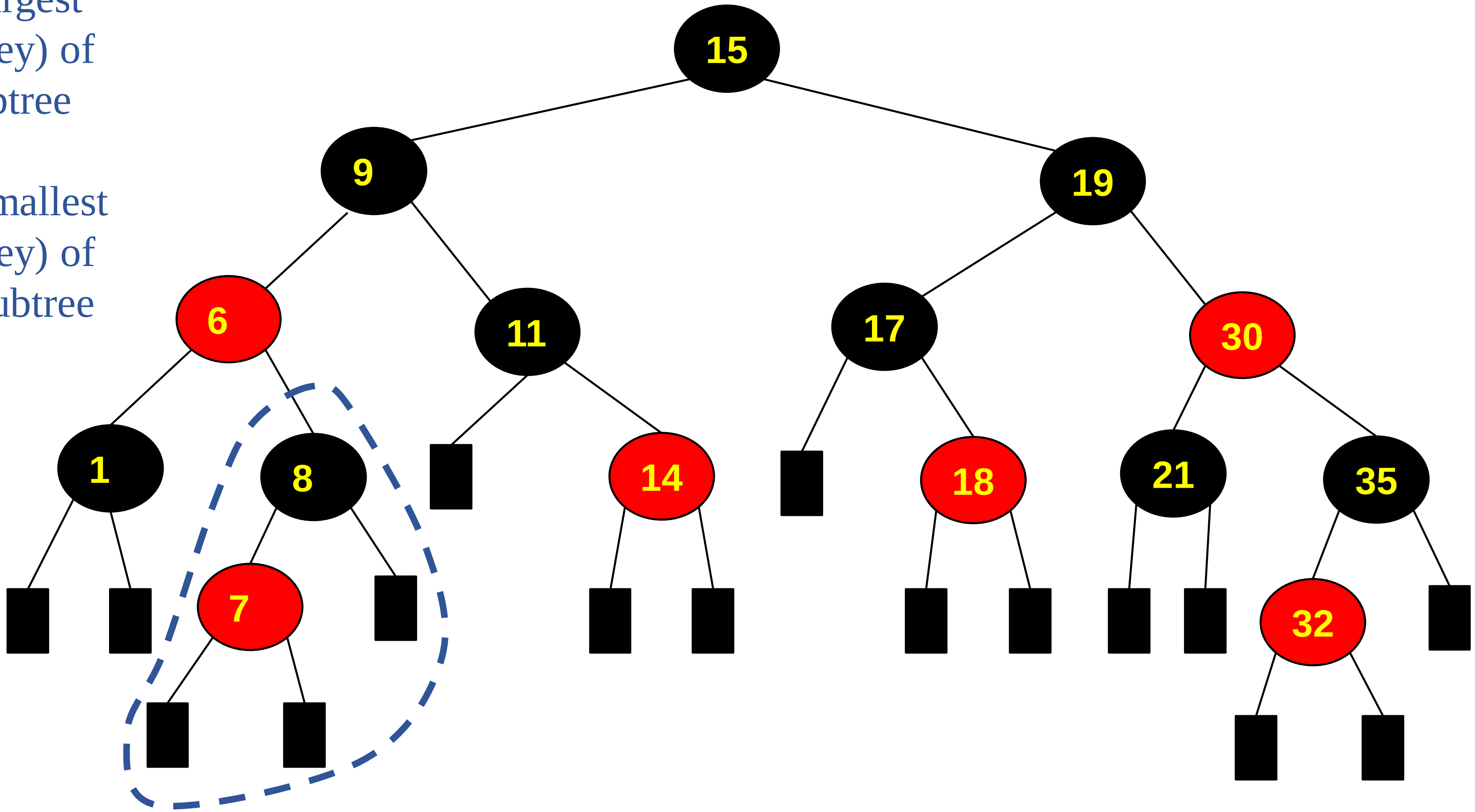
- Find the smallest number (key) of the right subtree



Delete a node? (1)
Delete 6

- Find the largest number (key) of the left subtree

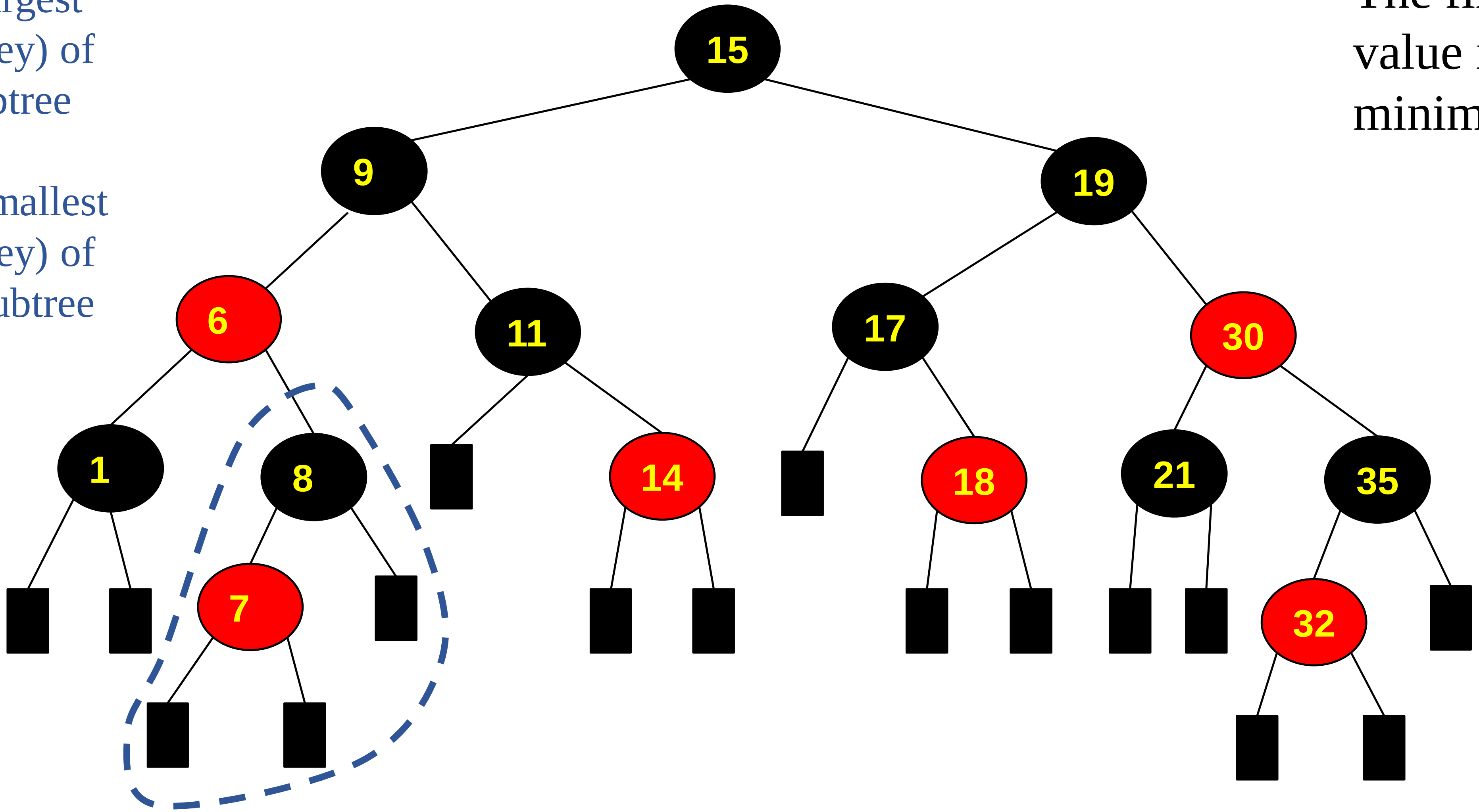
- Find the smallest number (key) of the right subtree



Delete a node? (1)
Delete 6

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree

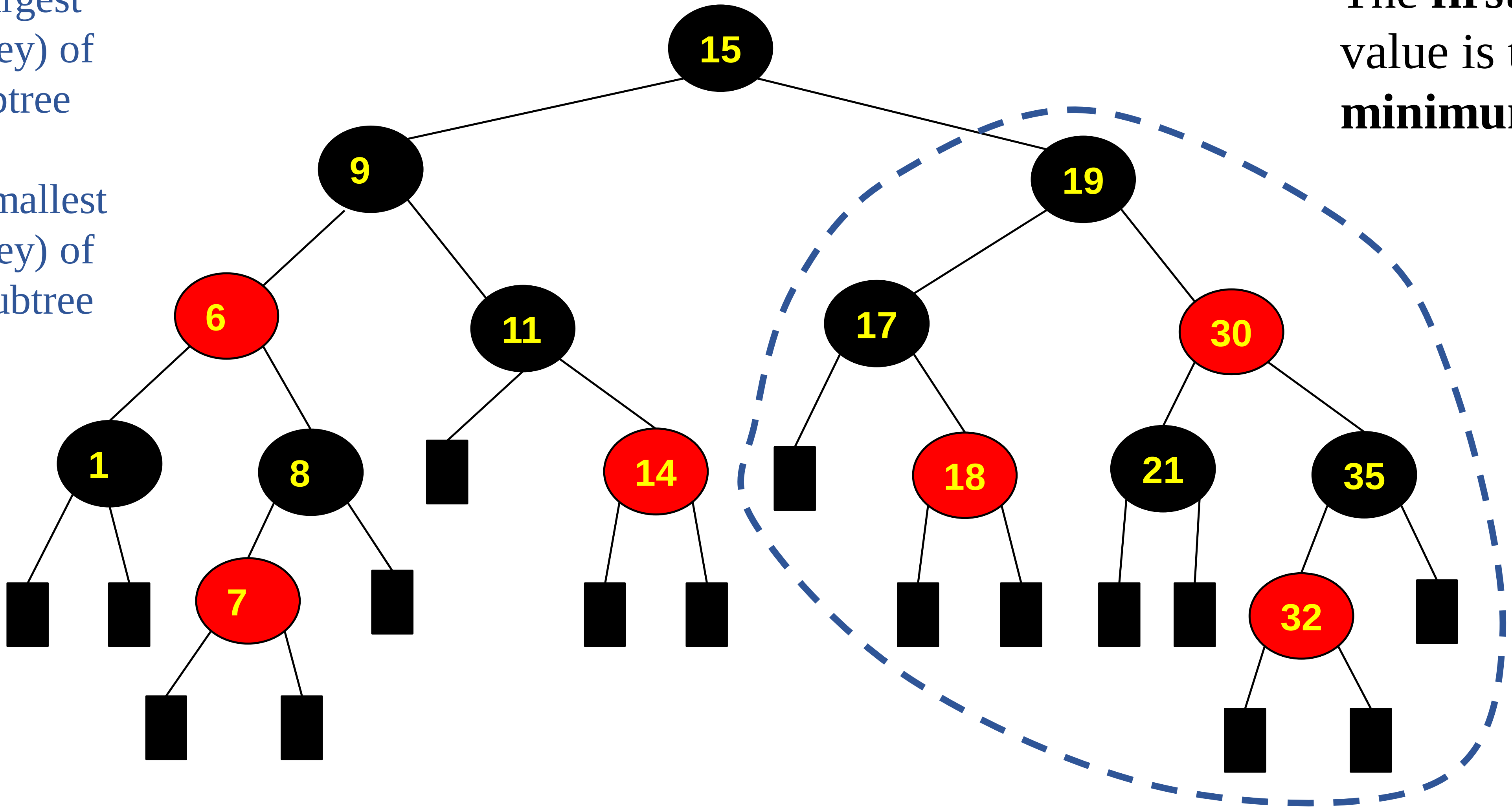


In-order traversal. The first output value is the minimum key.

Delete a node? (1)
Delete 6

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree

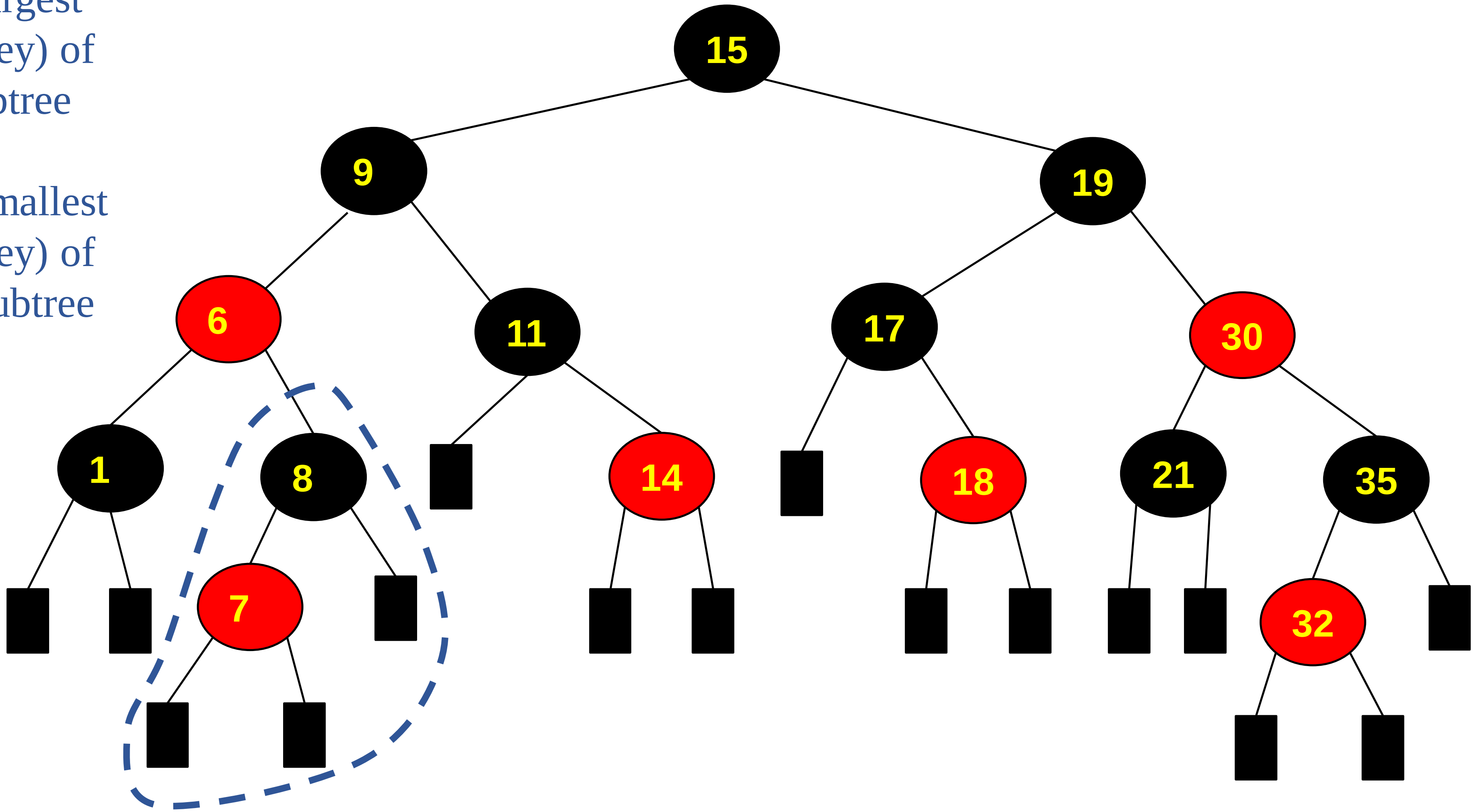


In-order traversal. The **first** output value is the **minimum** key.

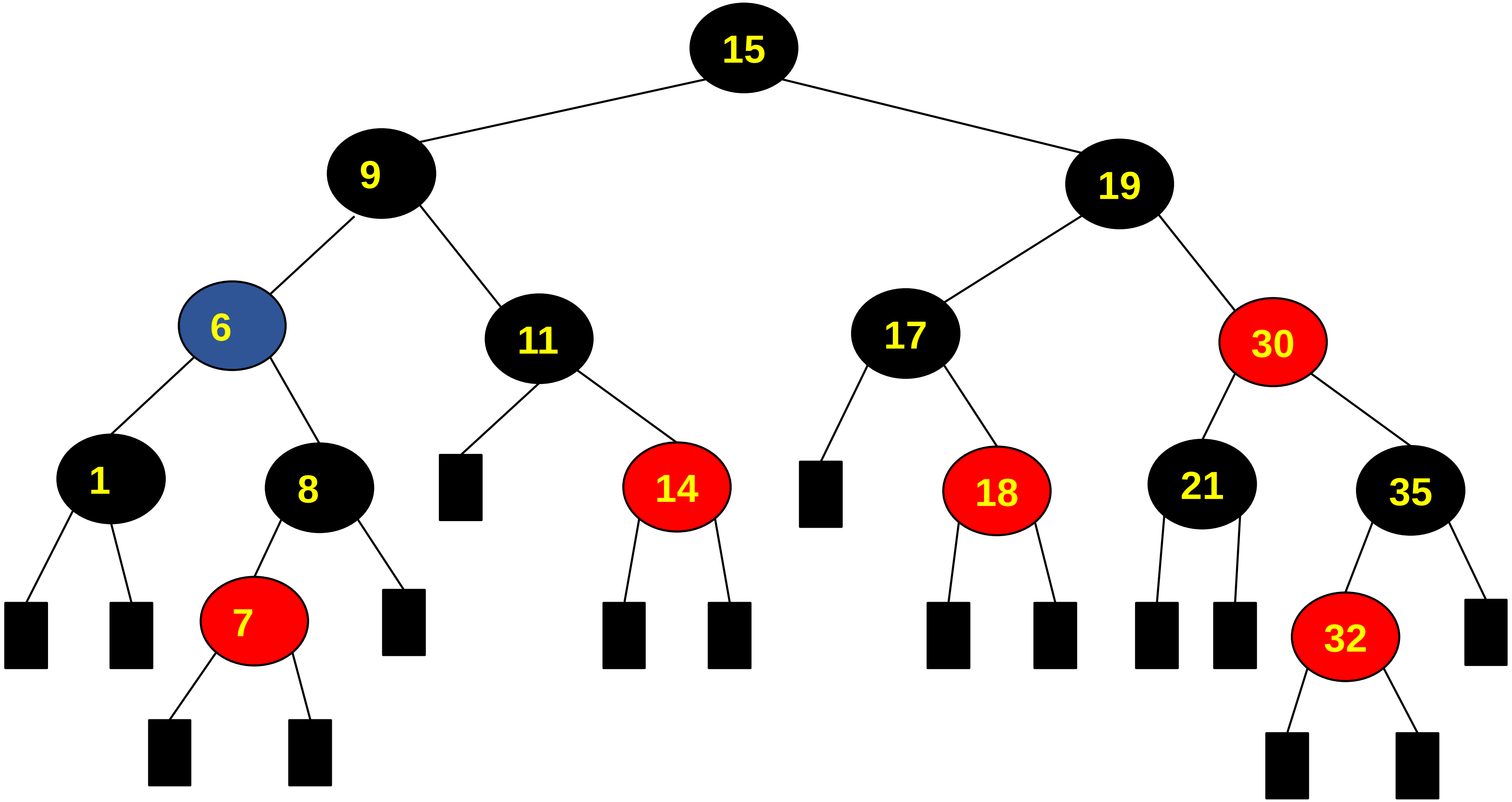
Delete a node? (1)
Delete 6

- Find the largest number (key) of the left subtree

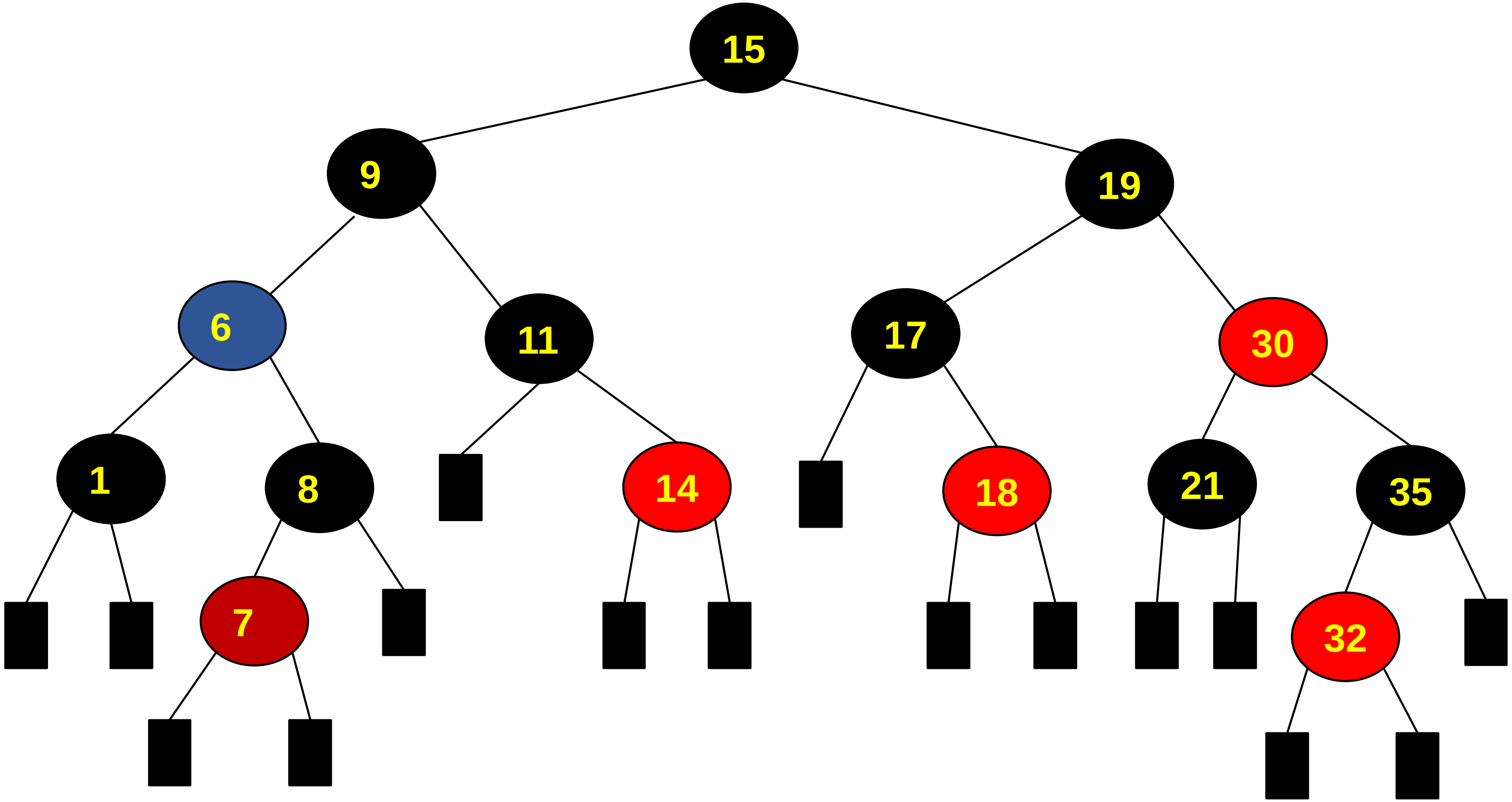
- Find the smallest number (key) of the right subtree



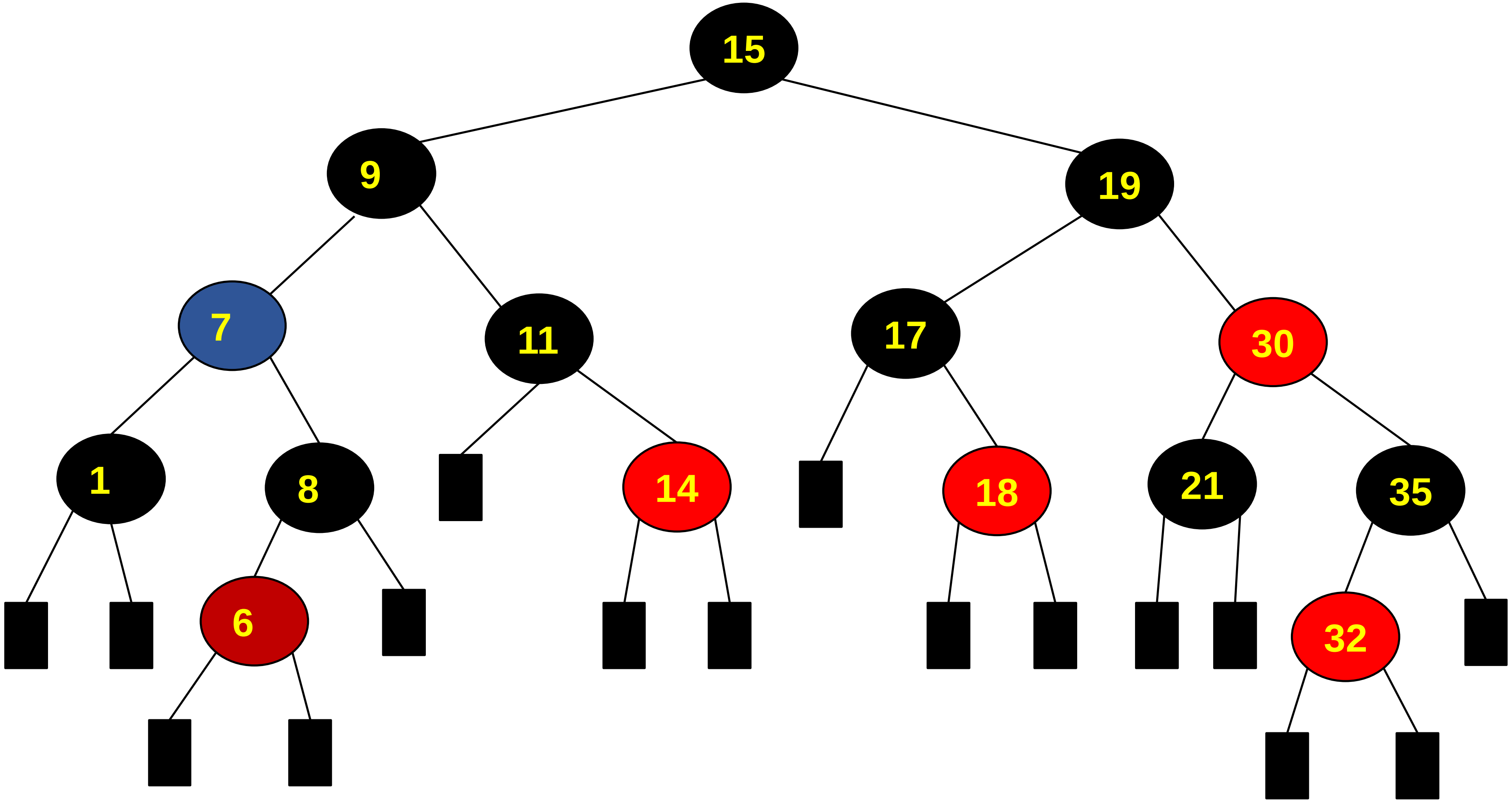
Delete a node? (2)
Delete 6



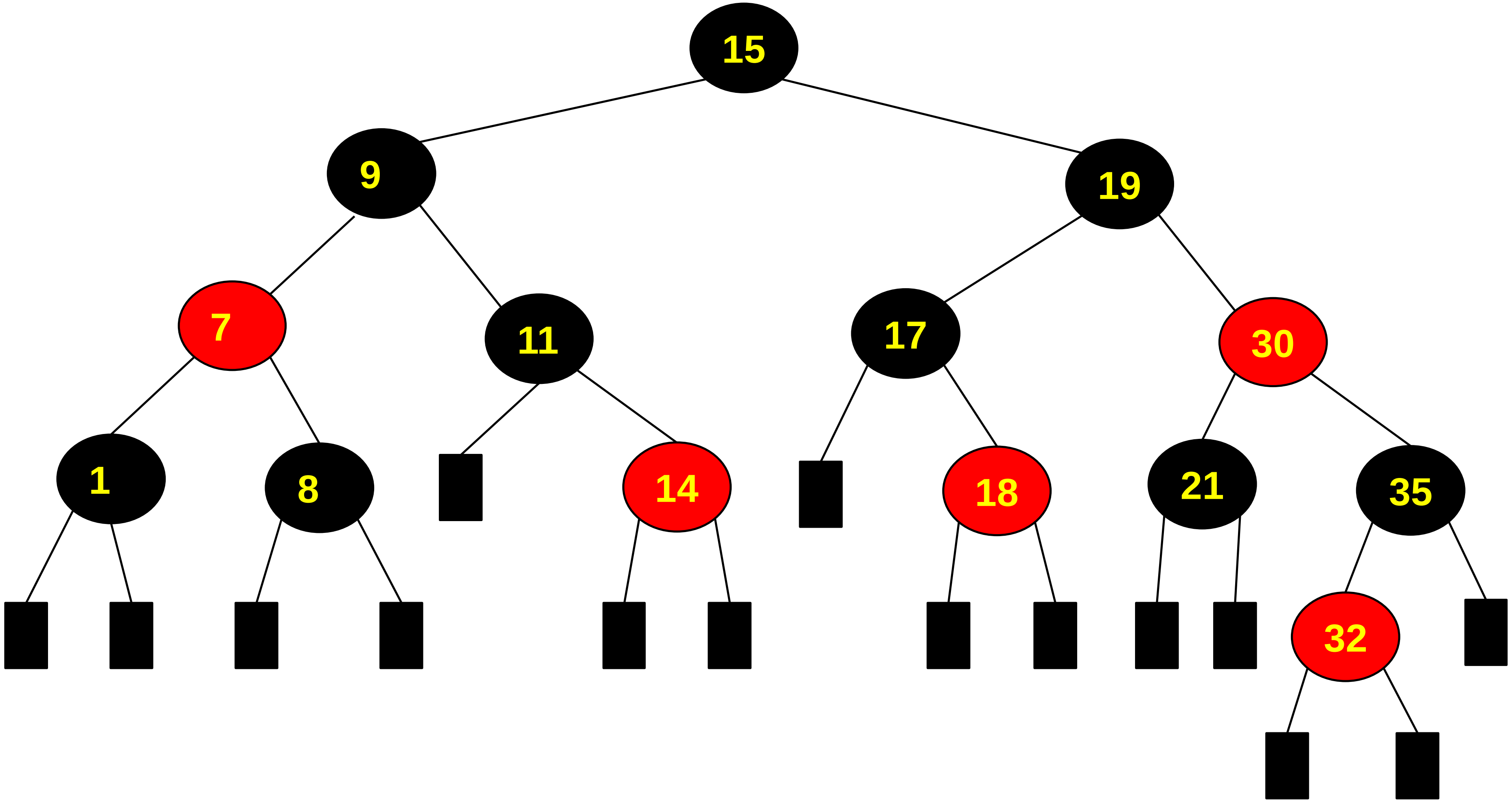
Delete a node? (3)
Delete 6



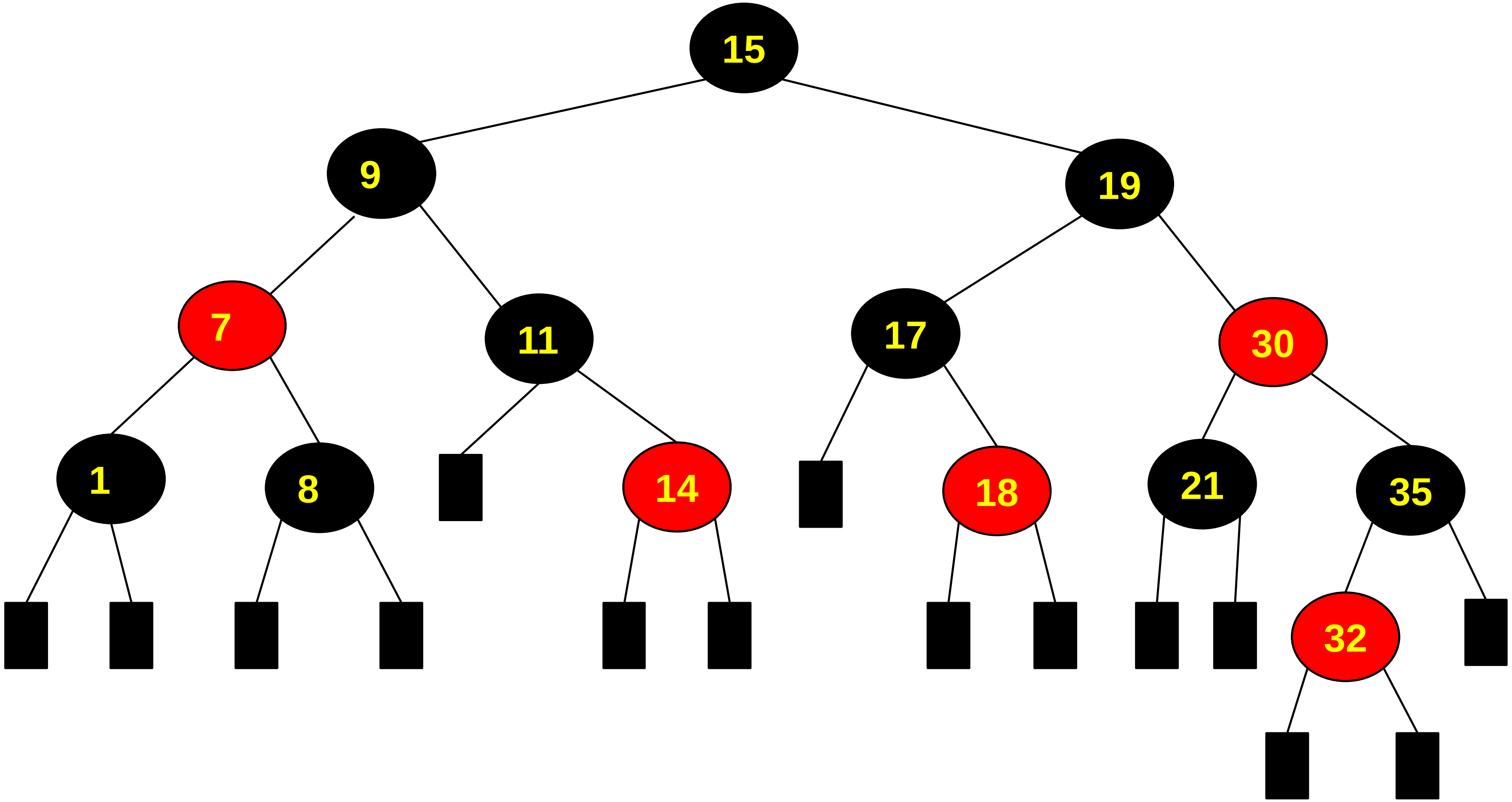
Delete a node? (4)
Delete 6



Delete a node? (5)
Delete 6



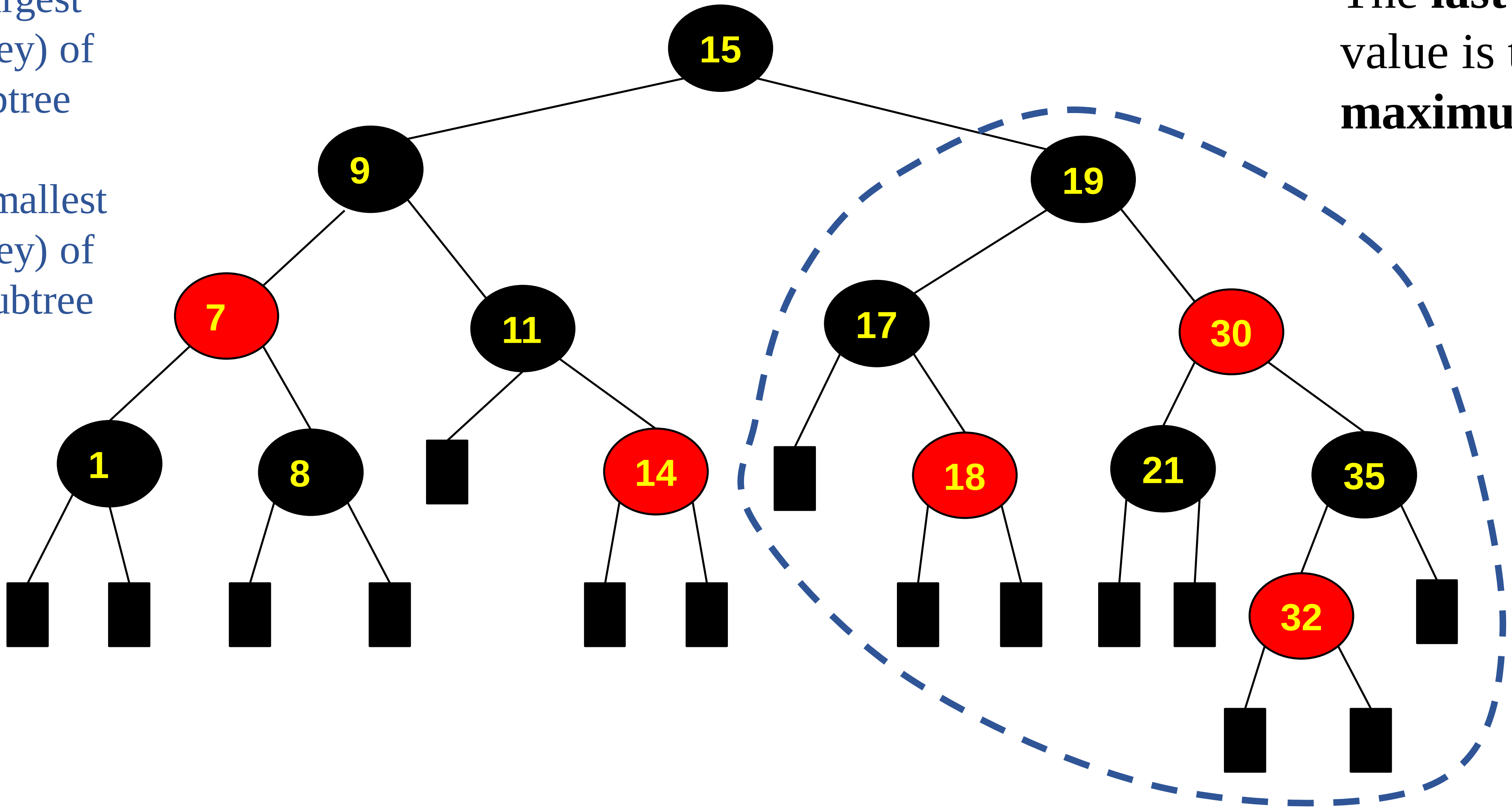
Delete a node? (1)
Delete 19



Delete a node? (2)
Delete 19

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree

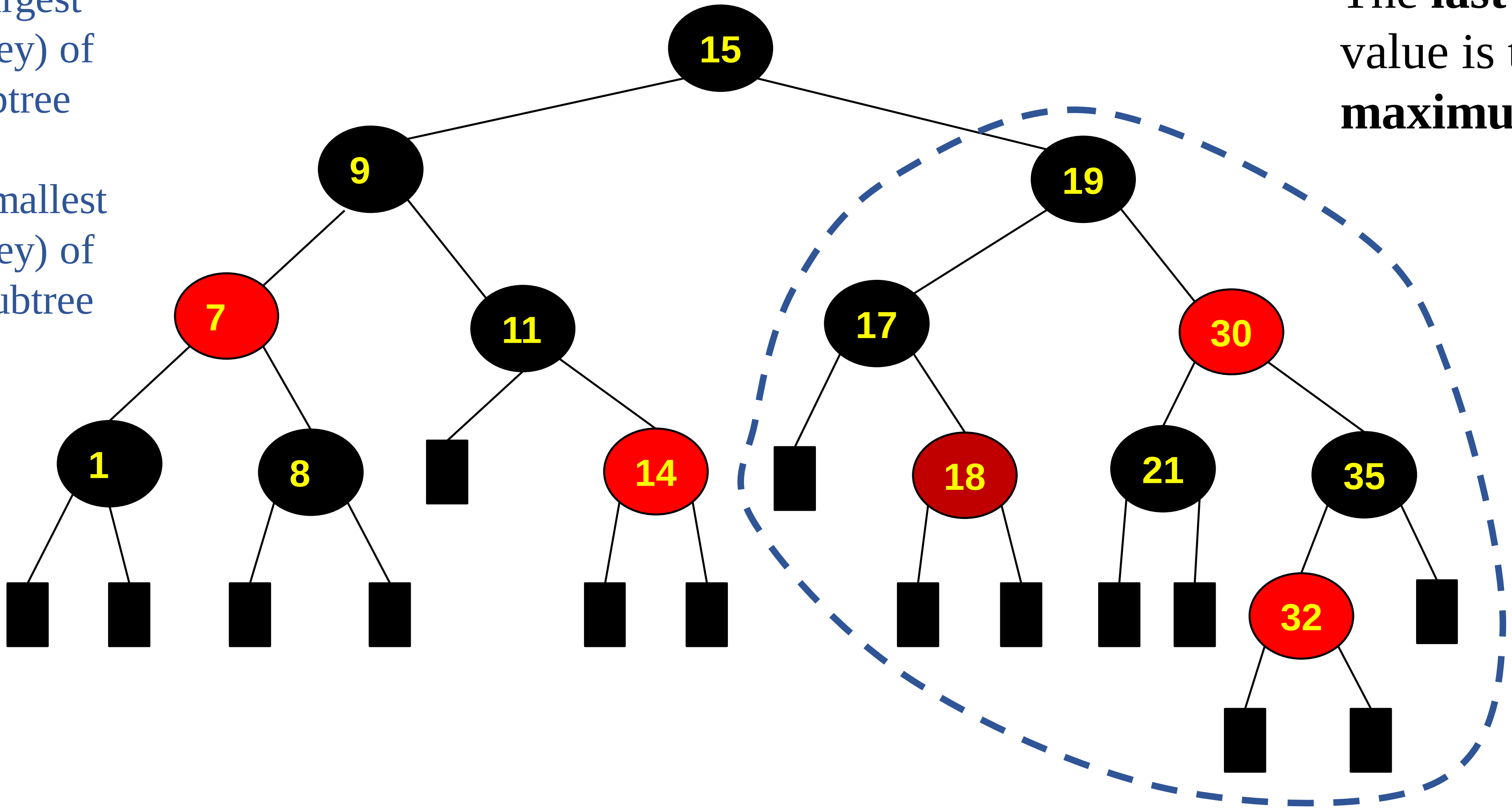


In-order traversal. The **last** output value is the **maximum** key.

Delete a node? (2)
Delete 19

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree

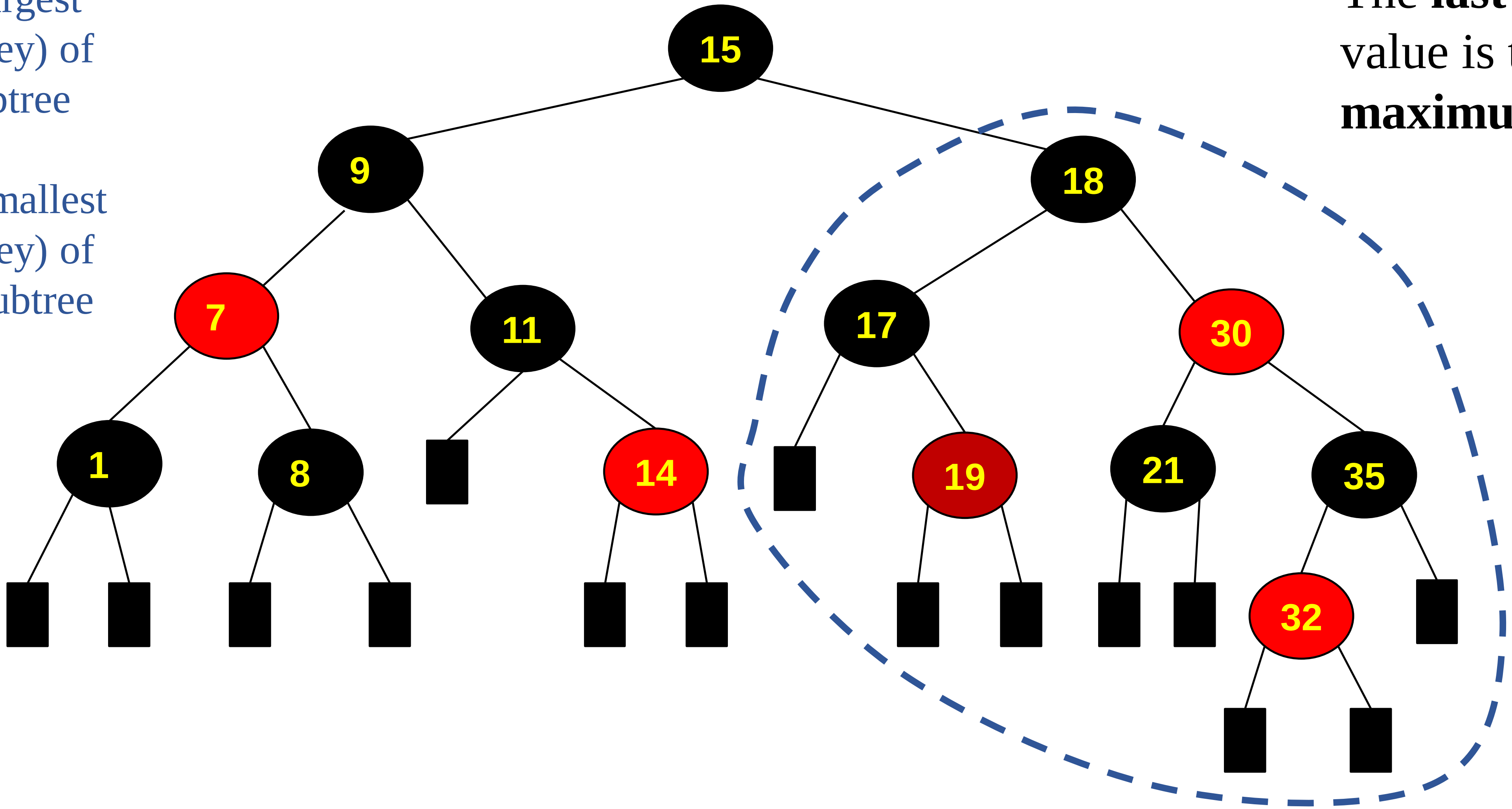


In-order traversal. The **last** output value is the **maximum** key.

Delete a node? (2)
Delete 19

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree



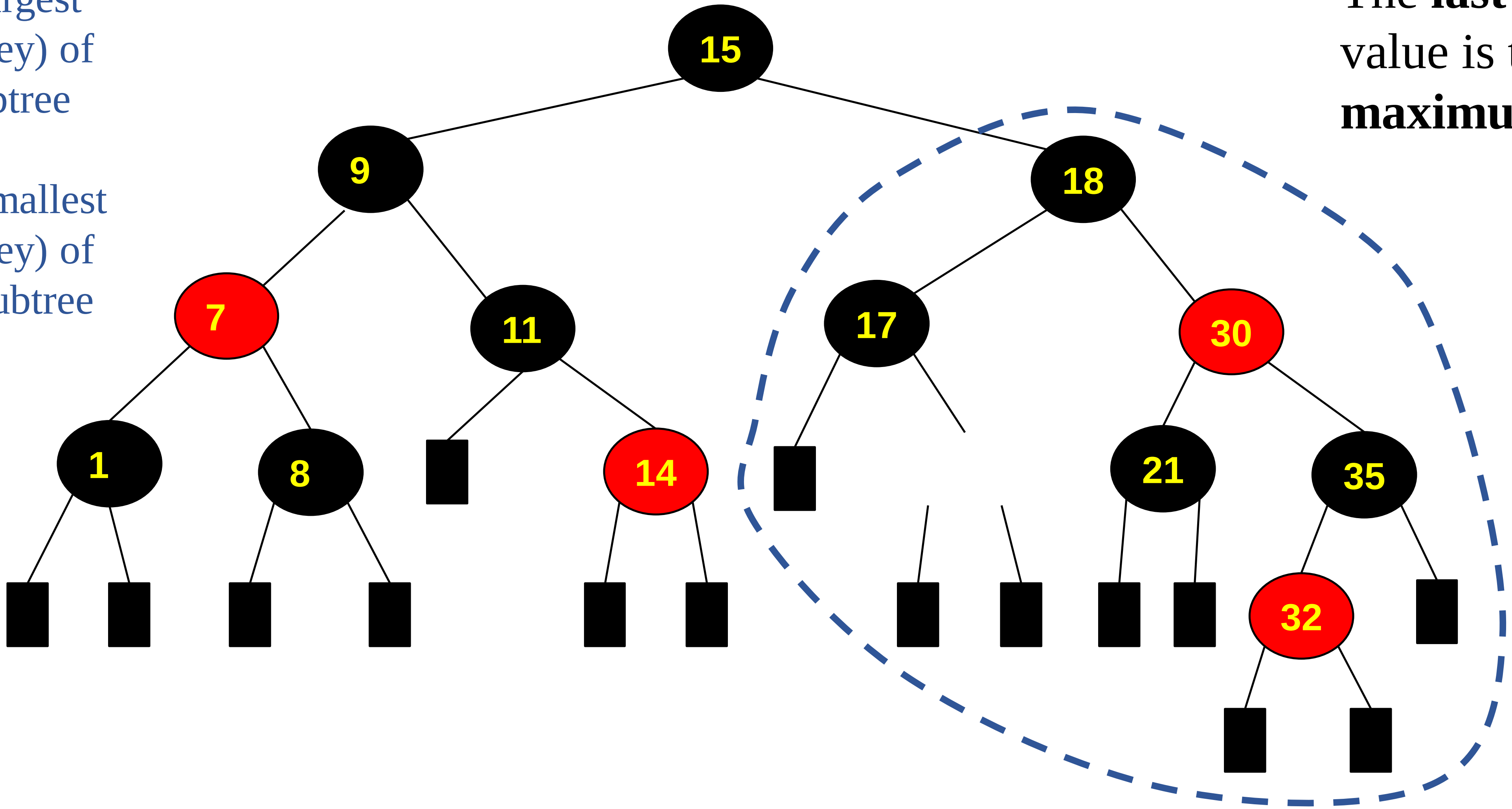
In-order traversal. The **last** output value is the **maximum** key.

Delete a node? (2)
Delete 19

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree

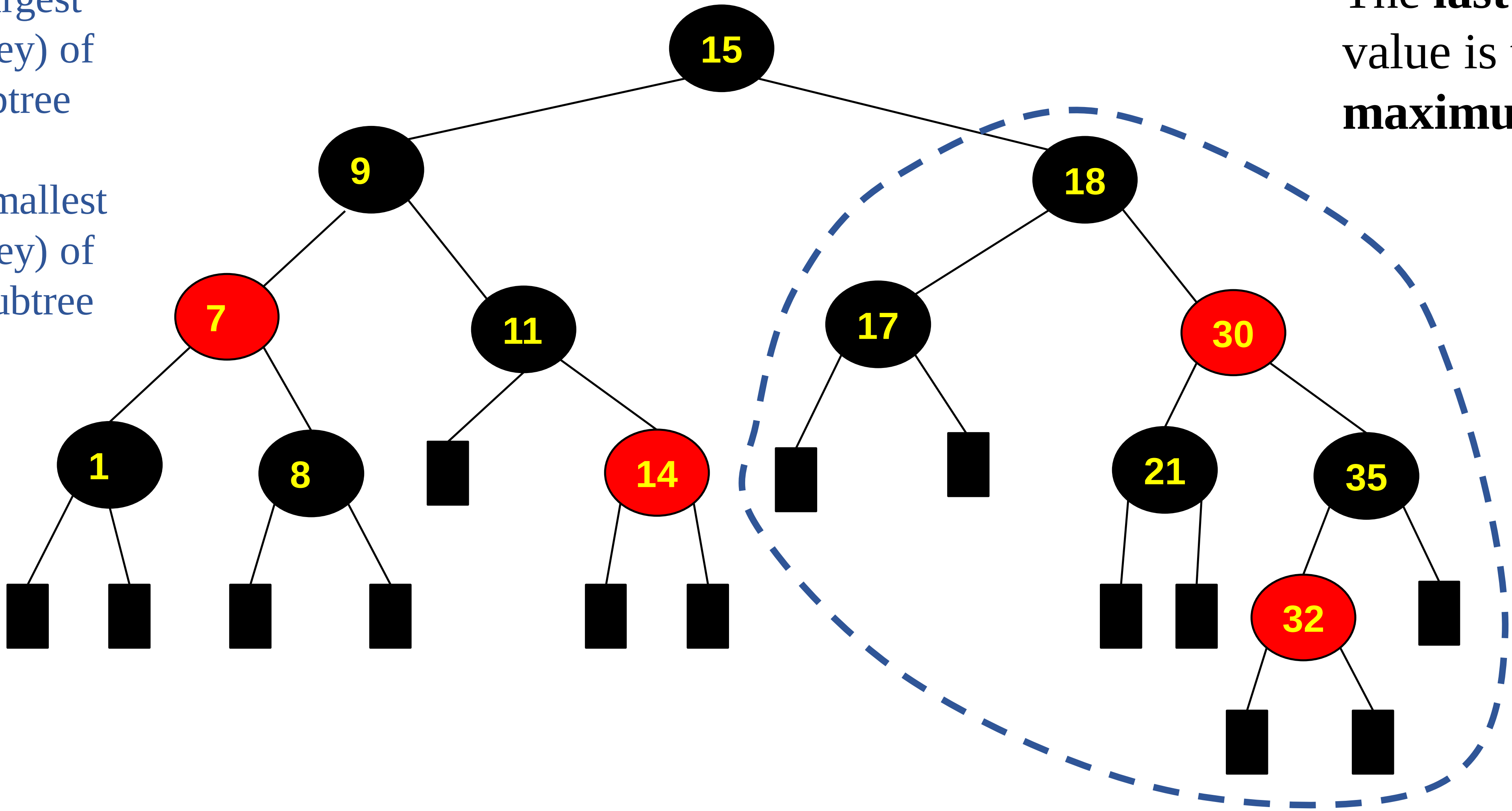
In-order traversal.
The **last** output value is the **maximum** key.



Delete a node? (2)
Delete 19

- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree



In-order traversal. The **last** output value is the **maximum** key.

Write a recursive function to return the maximum key by using the modified “in-order” traversal?

- Right subtree first
- Left subtree next

Write a recursive function to return the maximum key by using the modified “in-order” traversal?

- Right subtree first
- Left subtree next

```
int getMaxKey( const Node *n ) {
   if ( n == 0 ) return -1; // not exist
   if (n→left == 0 )
   if (n→right == 0 ) ??????????
   ?????????????
}
```

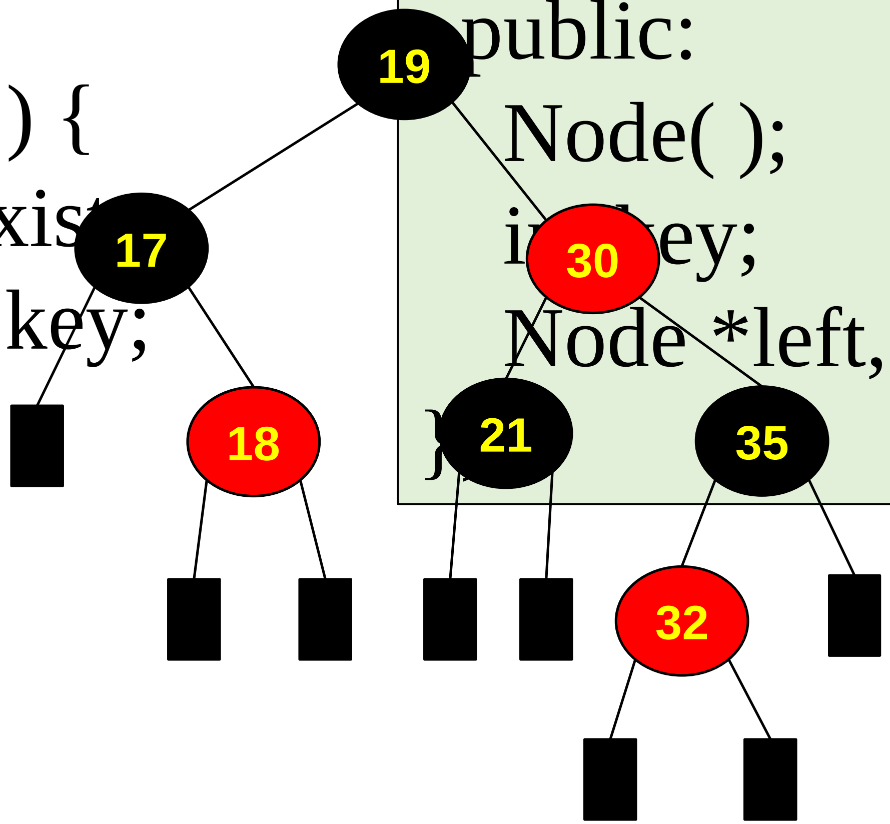
```
class Node {
  public:
    Node( );
    Node *left, *right;
};
```

Write a recursive function to return the maximum key by using the modified “in-order” traversal?

- Right subtree first
- Left subtree next

```
int getMaxKey( const Node *n ) {
   if ( n == 0 ) return -1; // not exist
   if ( n→right == 0 ) return n→key;
   getMaxKey( n→right );
}
```

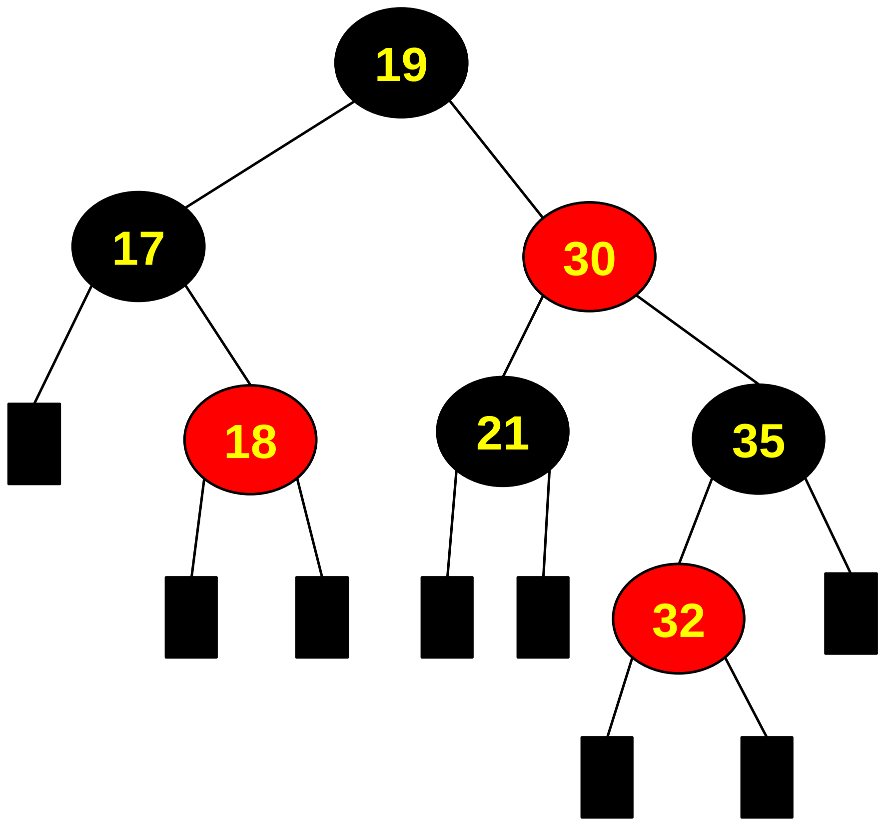
```
class Node {
 public:
   Node( );
   int key;
   Node *left, *right;
};
```



Write a recursive function to return the maximum key by using the modified "in-order" traversal?

- Right subtree first
- Left subtree next

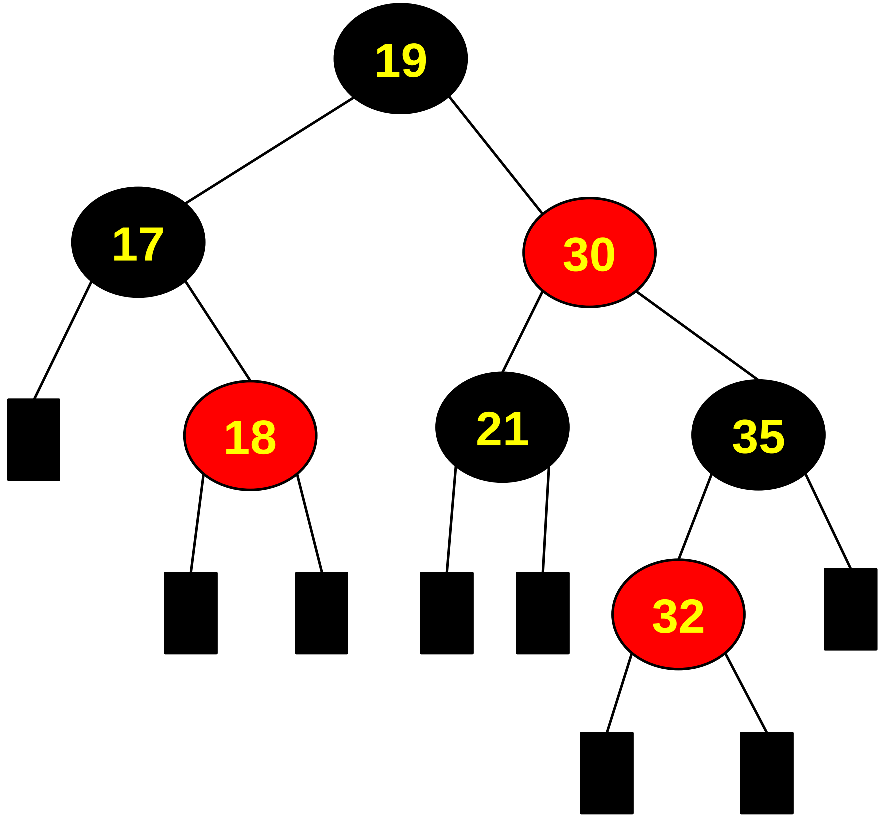
```
int getMaxKey( const Node *n ) {
   if ( n == 0 ) return -1; // not exist
   if ( n→right == 0 ) return n→key;
   return getMaxKey( n→right );
}
```



Write a recursive function to return the maximum key by using the modified "in-order" traversal?

- Right subtree first
- Left subtree next

```
Node *getMaxKey( const Node *n ) {
   if ( n == 0 ) return nullptr; // not exist
   if ( n→right == 0 ) return n;
   return getMaxKey( n→right );
}
```



Write a recursive function to return the maximum key by using the modified "in-order" traversal?

- Right subtree first
- Left subtree next

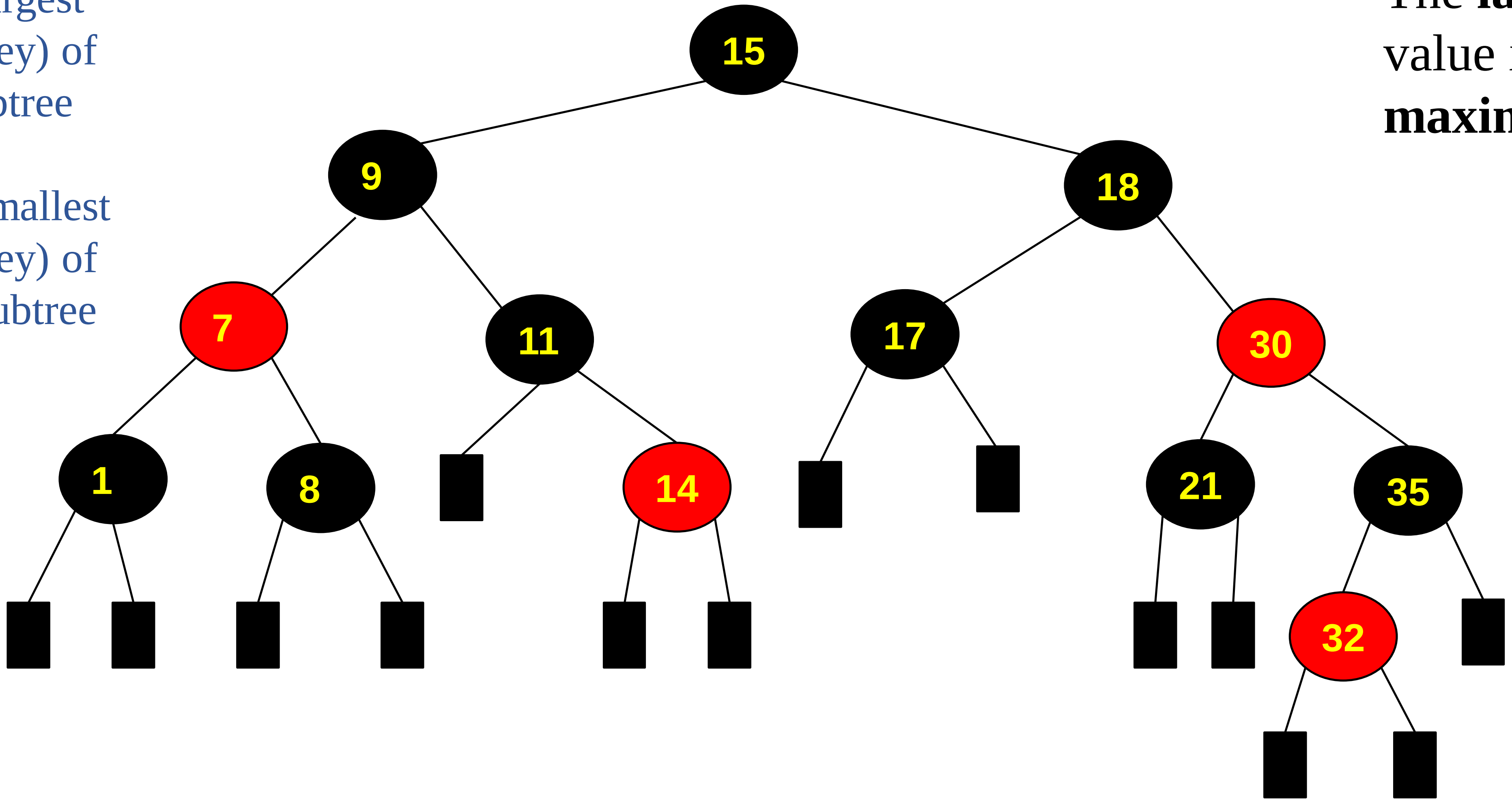
```
int getMaxKey( const Node *n ) {
   if (n == 0)return-1; // not exist
   if (n→right==0)return n→key;
 return getMaxKey(n→right);
}
```

```
class Node {
  public:
    Node( );
    int key;
    Node *left, *right;
};
```

Delete a node? (2)
Delete 19

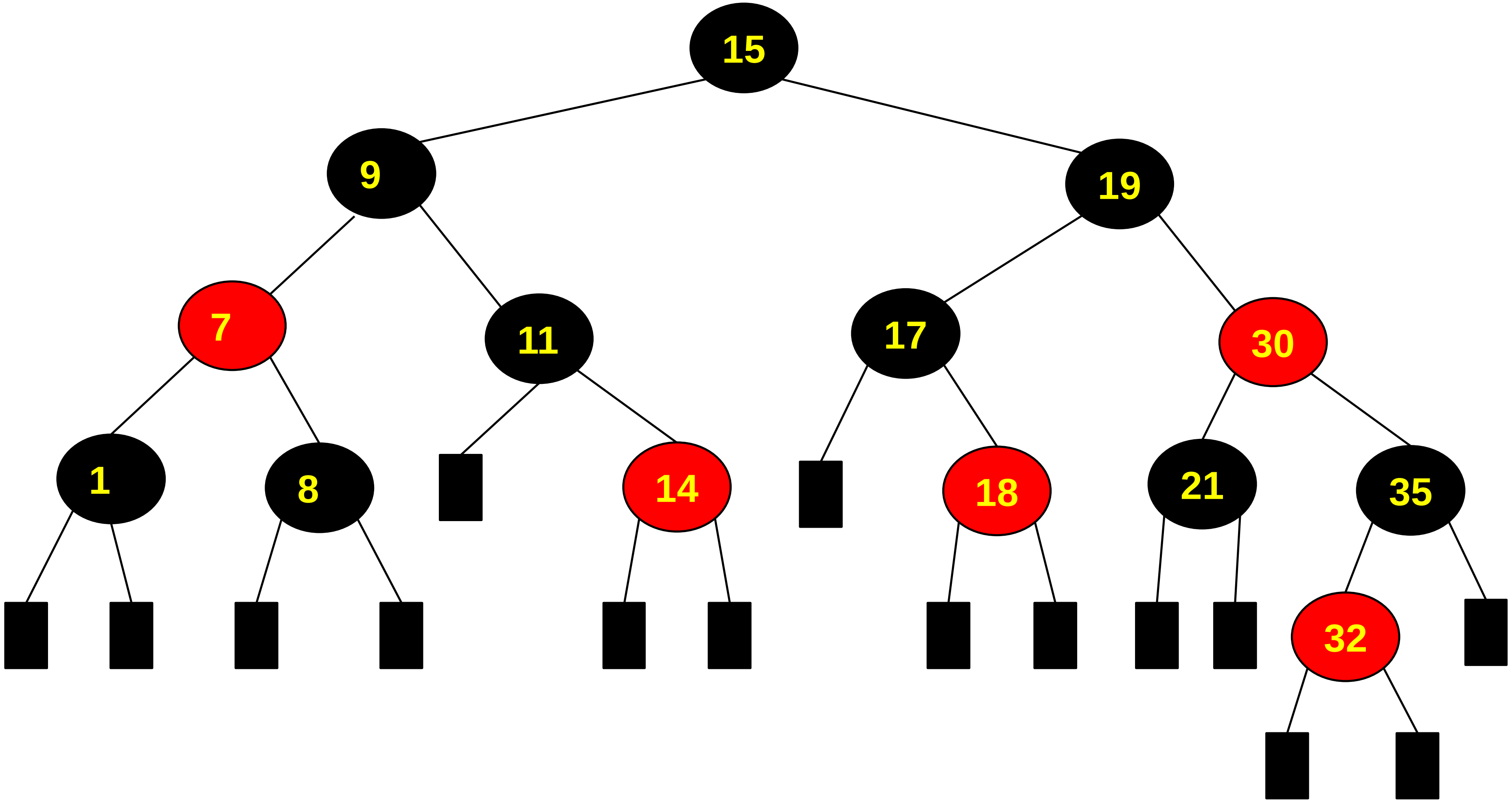
- Find the largest number (key) of the left subtree

- Find the smallest number (key) of the right subtree

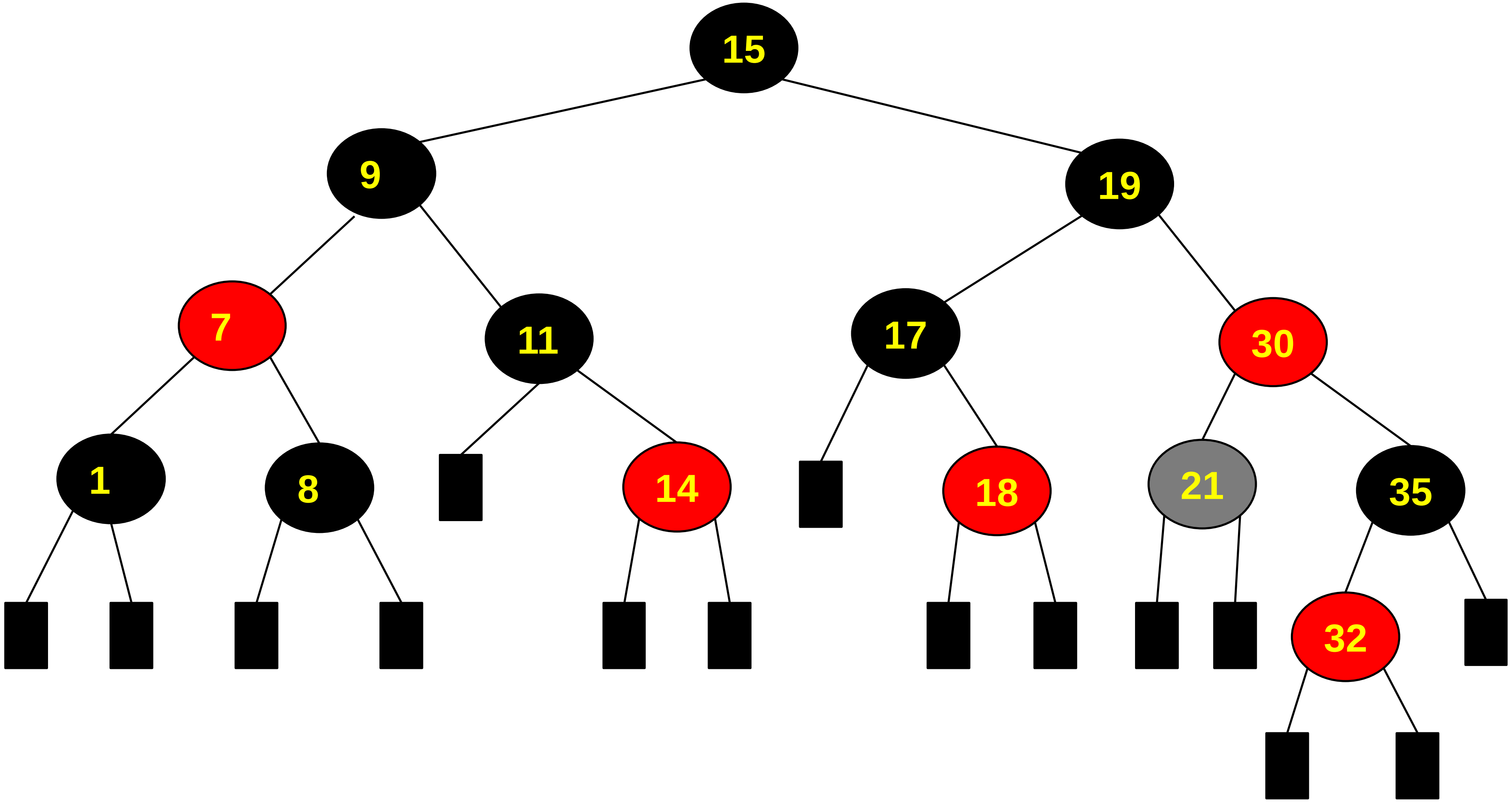


In-order traversal. The **last** output value is the **maximum** key.

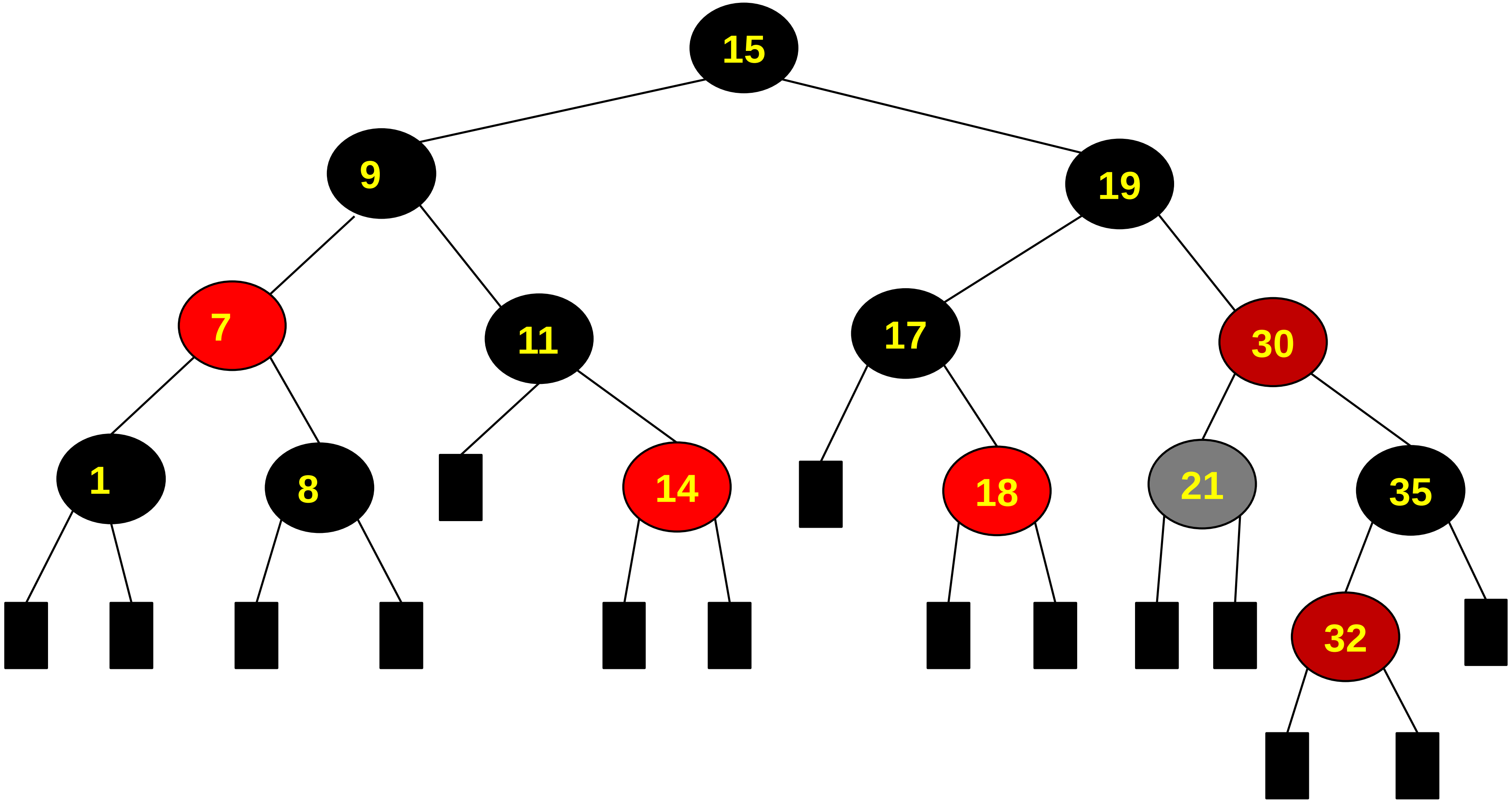
Delete a node? (1)
Delete 21



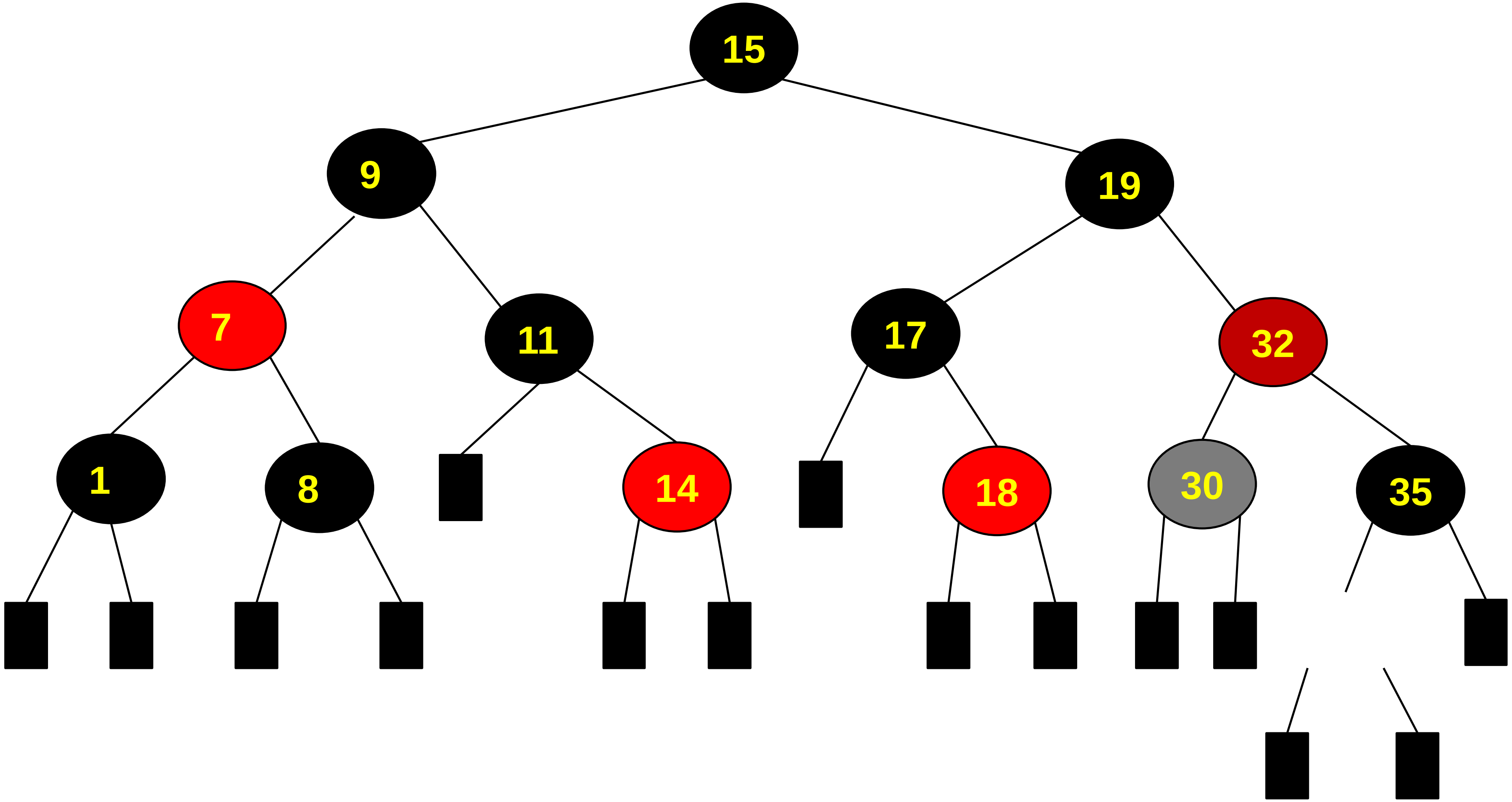
Delete a node? (2)
Delete 21



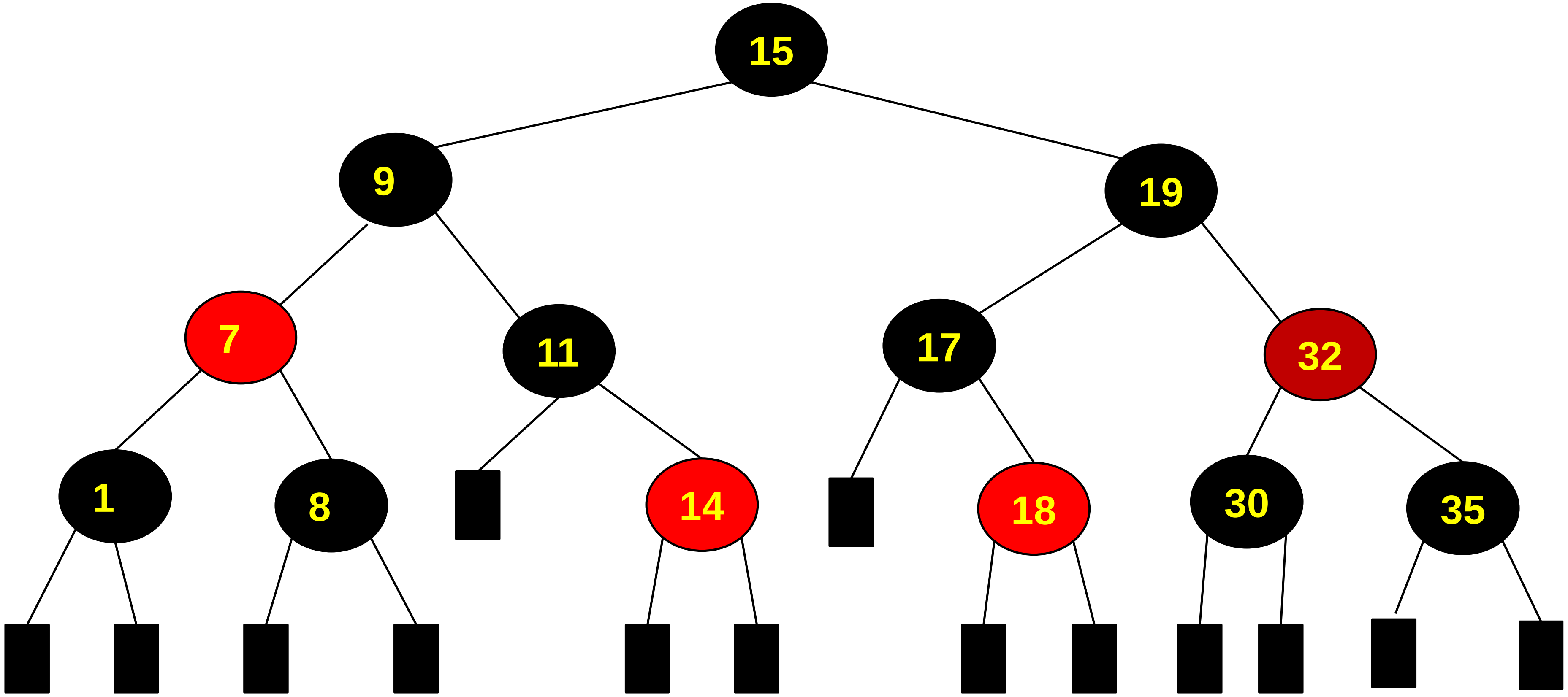
Delete a node? (3)
Delete 21



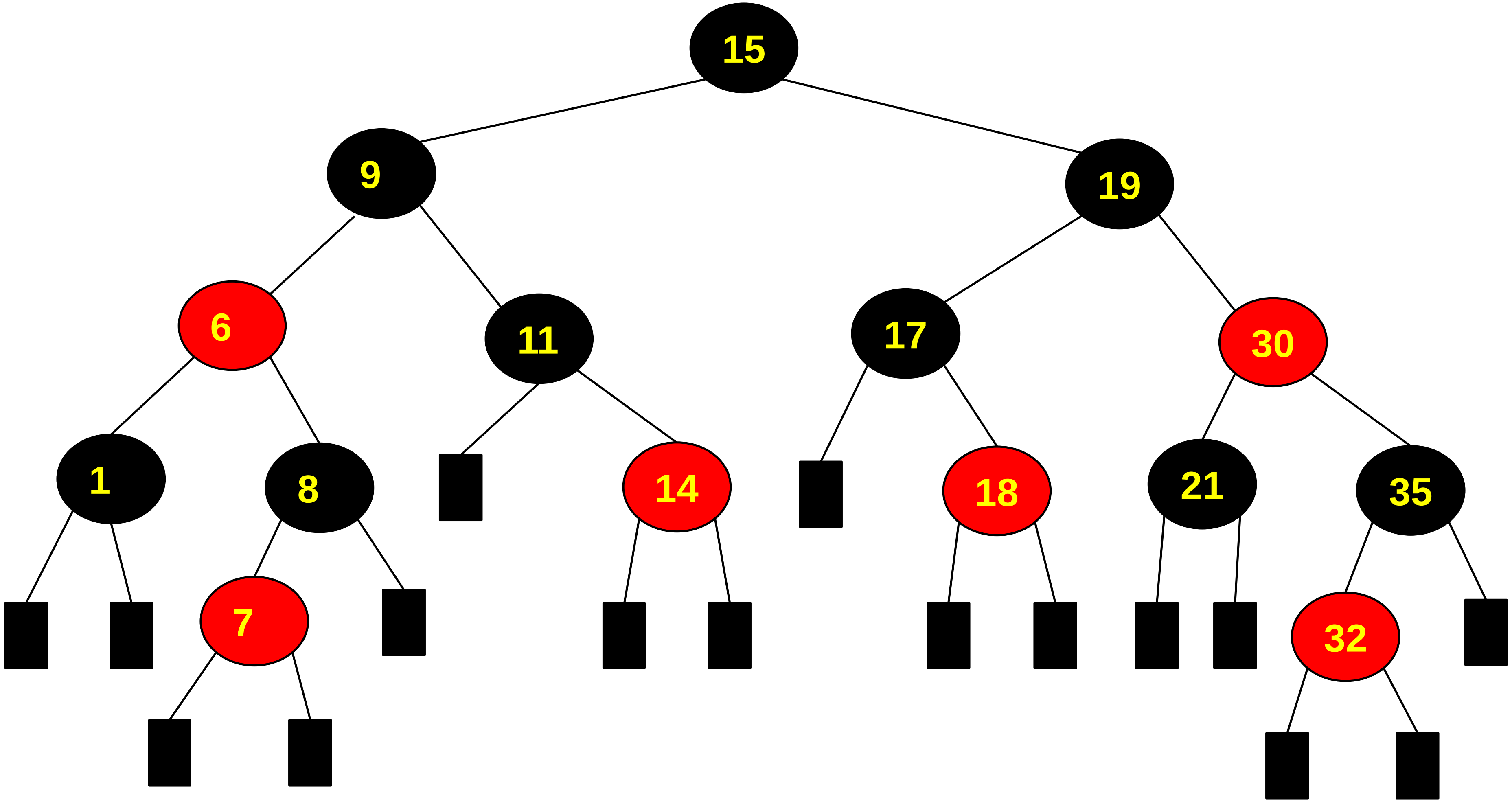
Delete a node? (3)
Delete 21



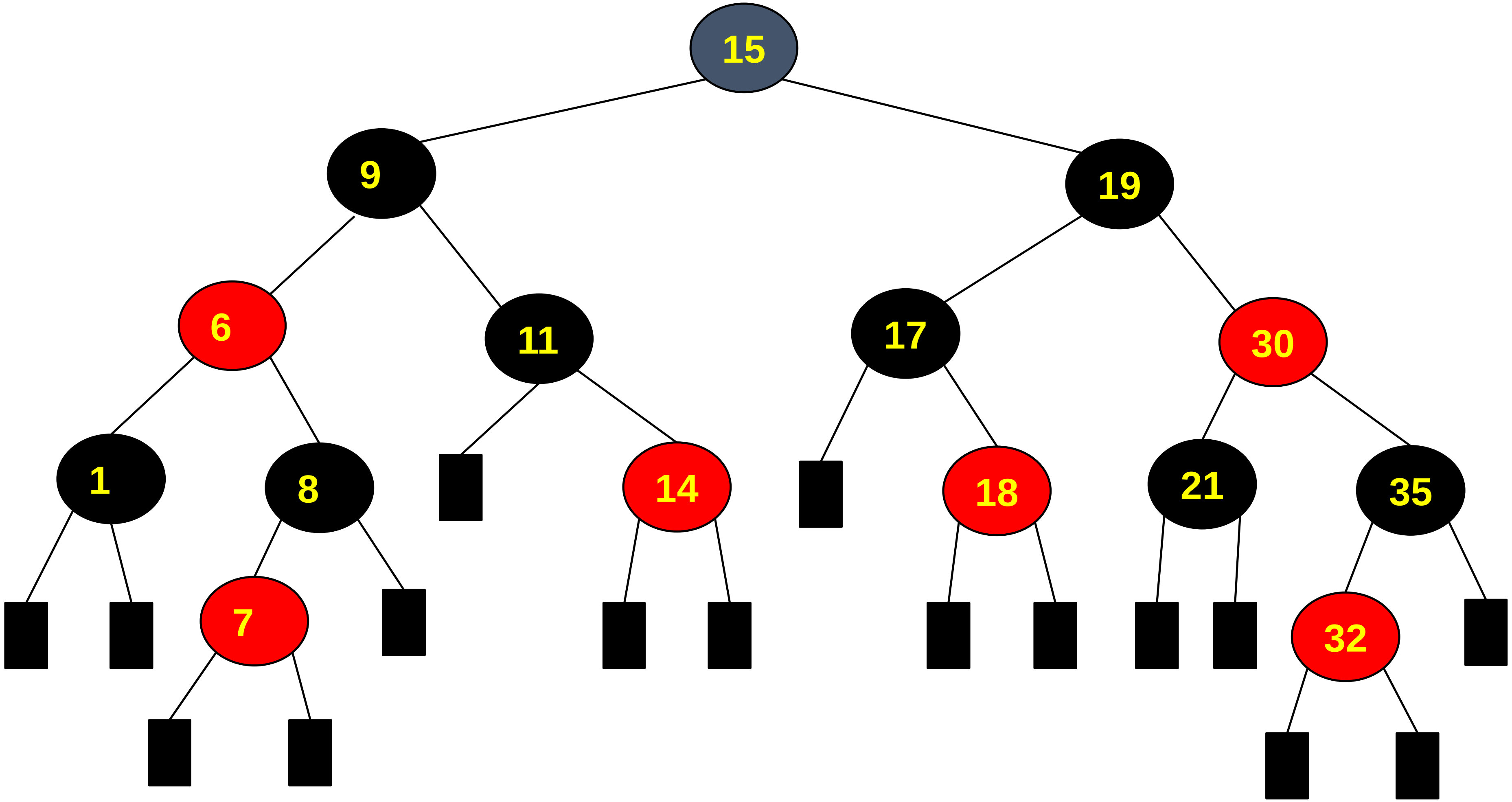
Delete a node? (3)
Delete 21



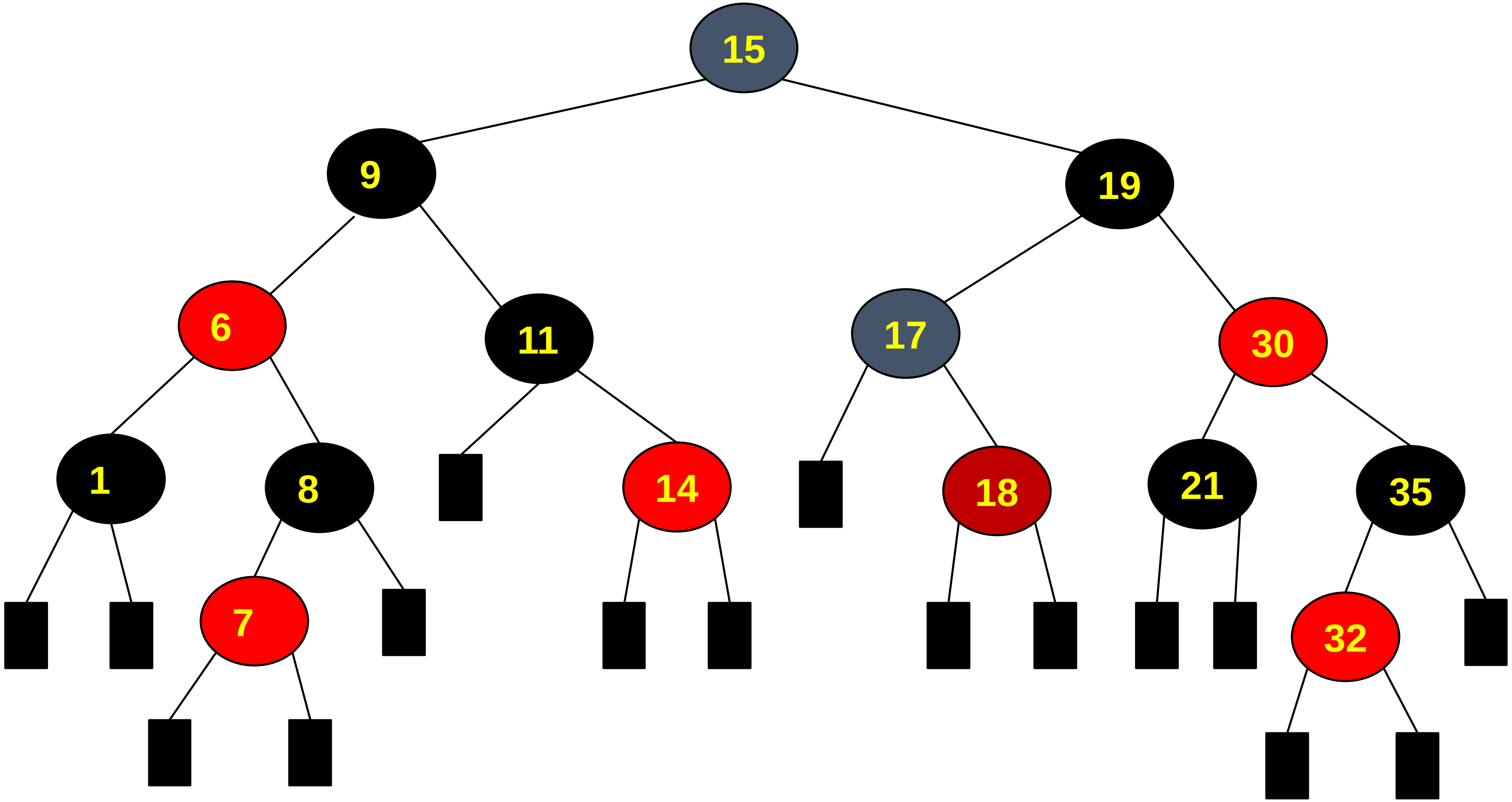
Delete a node? (1)
Delete 15



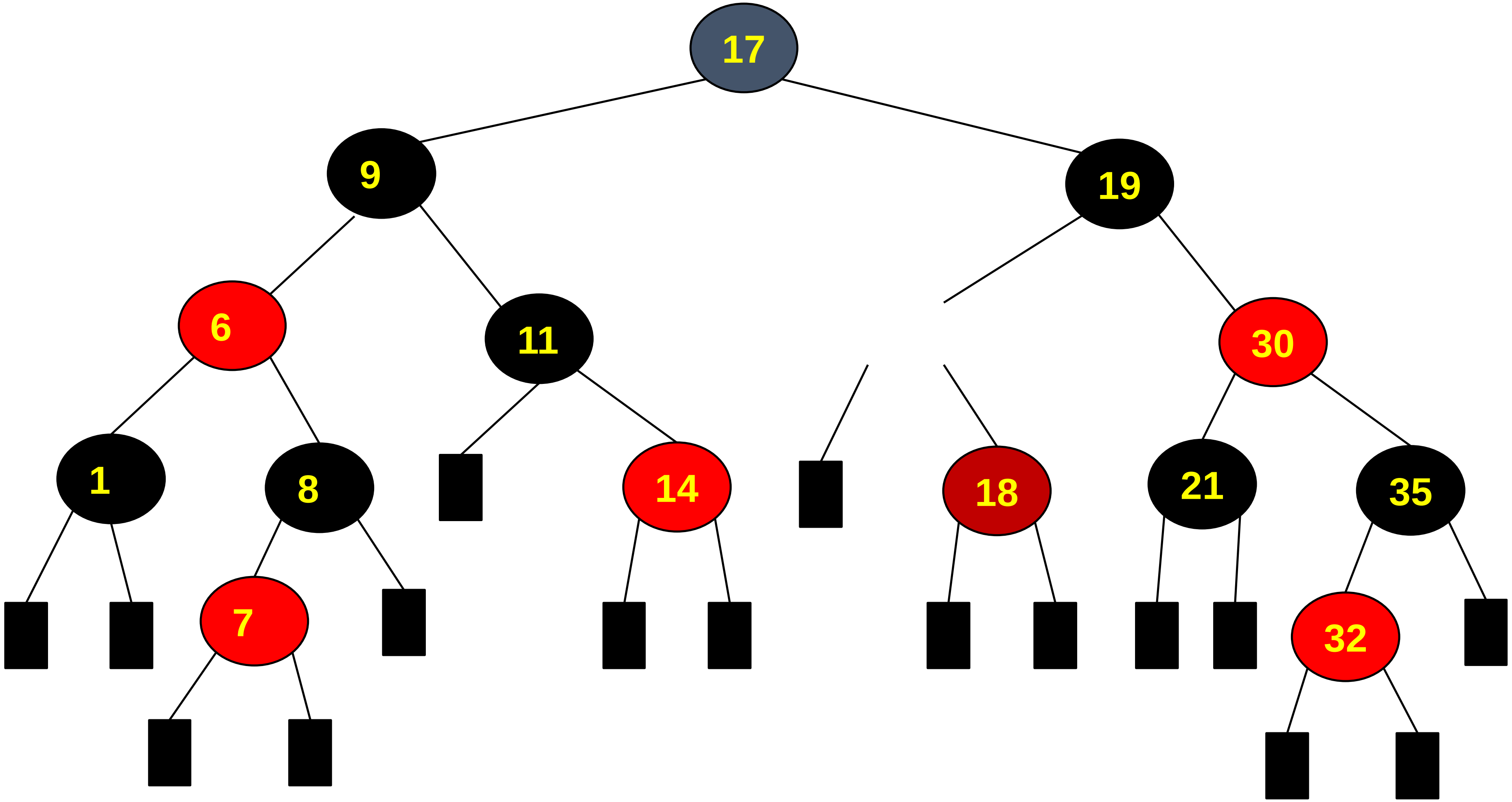
Delete a node? (2)
Delete 15



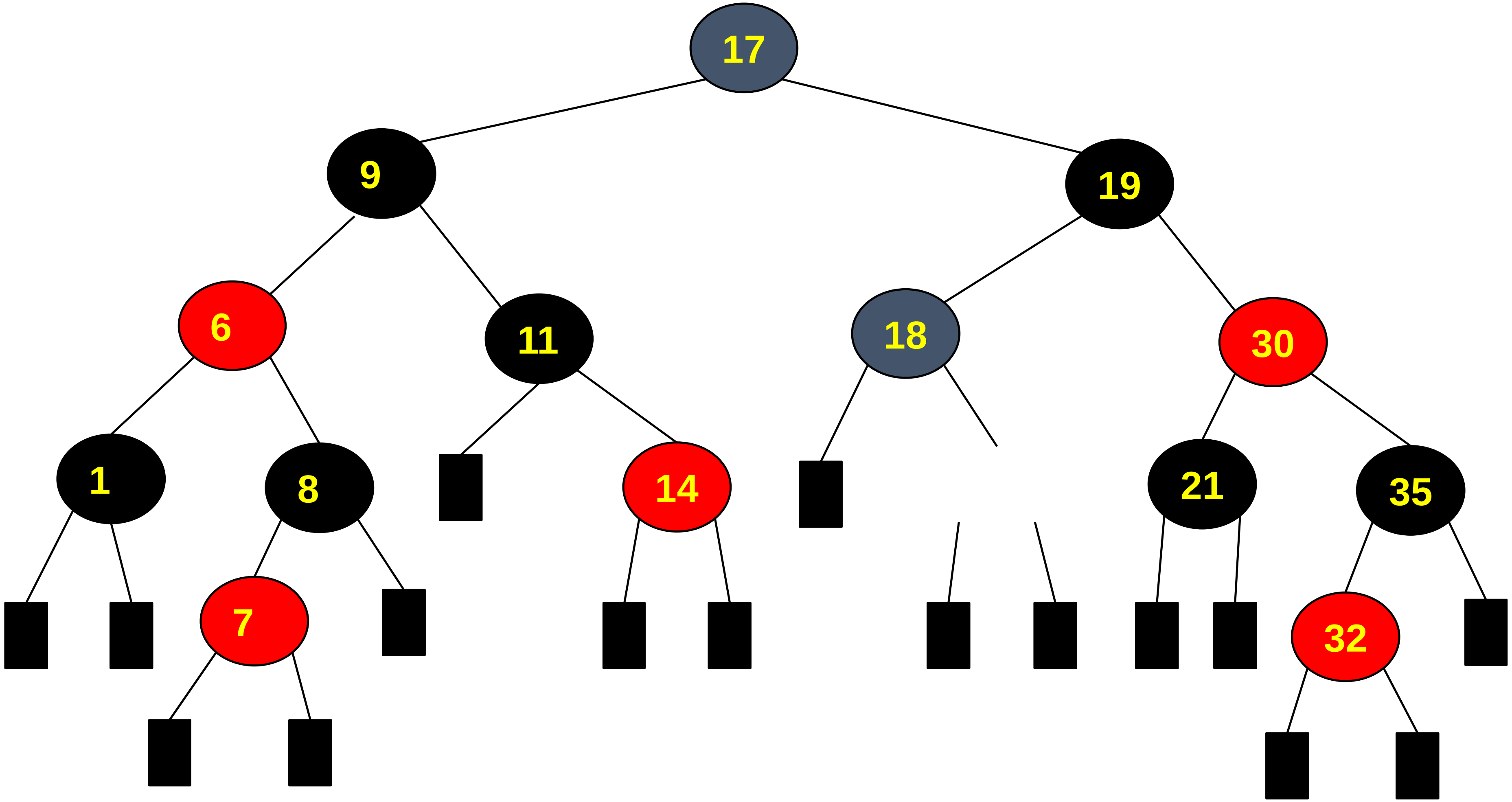
Delete a node? (3)
Delete 15



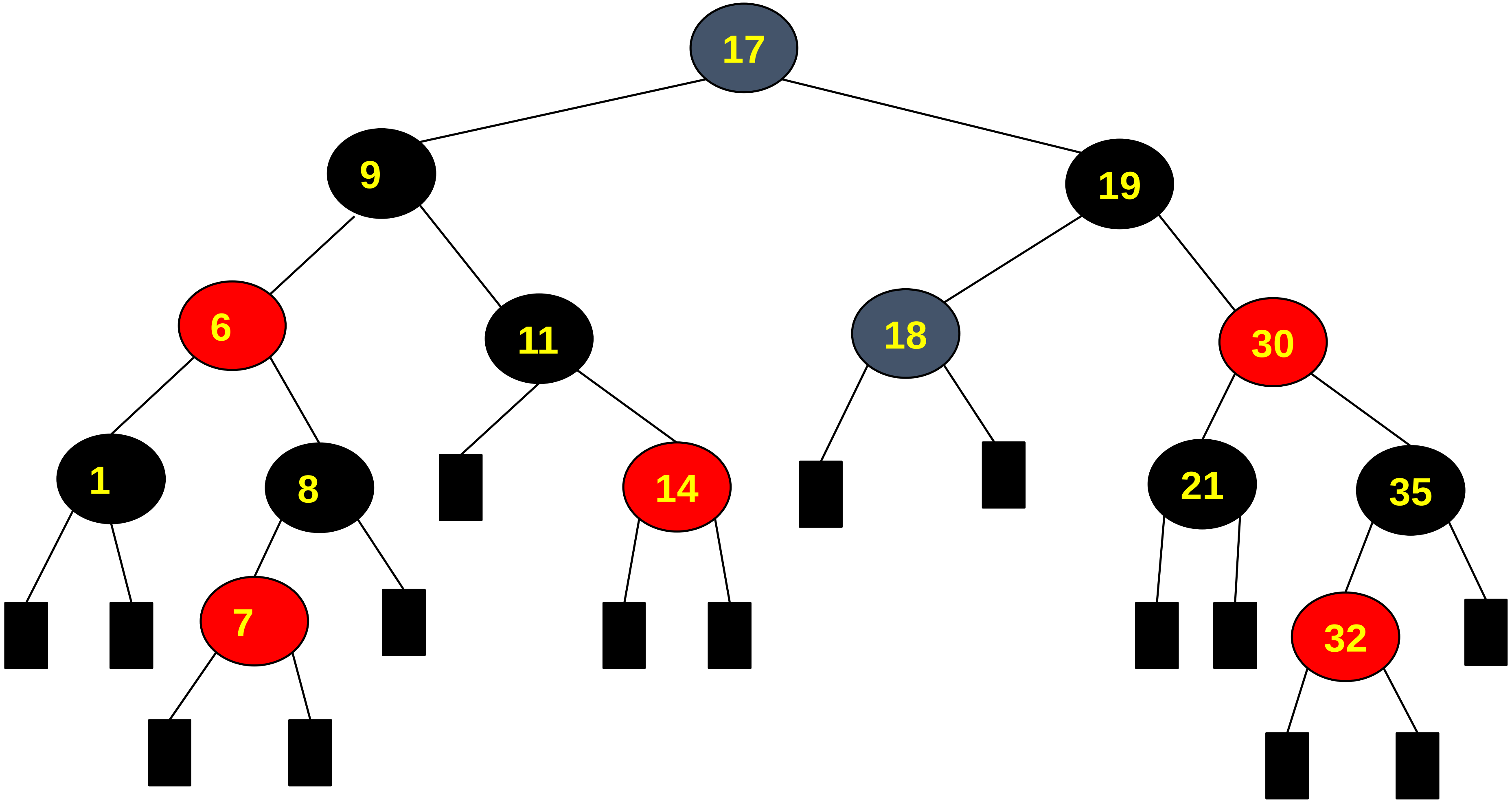
Delete a node? (4)
Delete 15



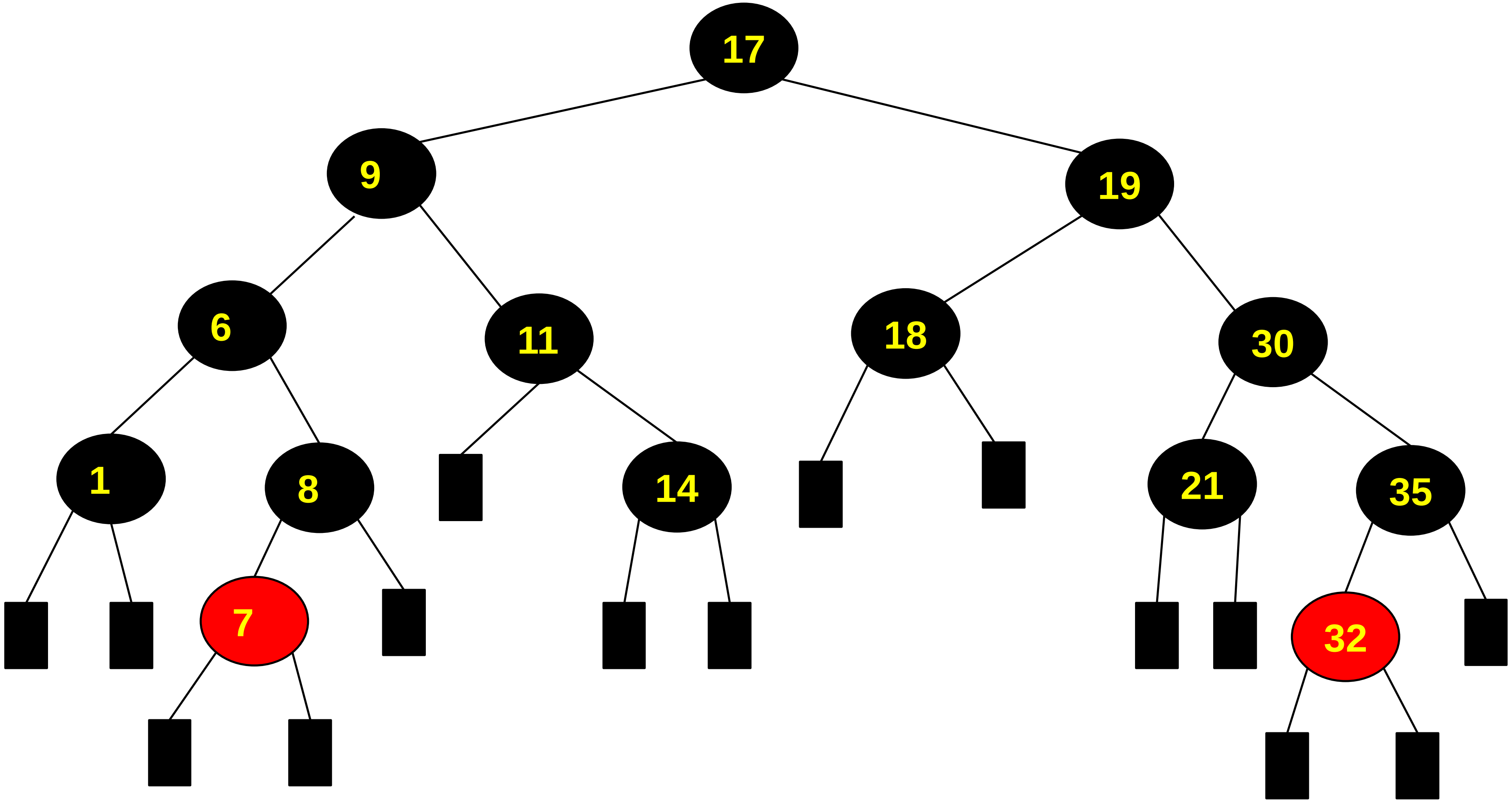
Delete a node? (5)
Delete 15



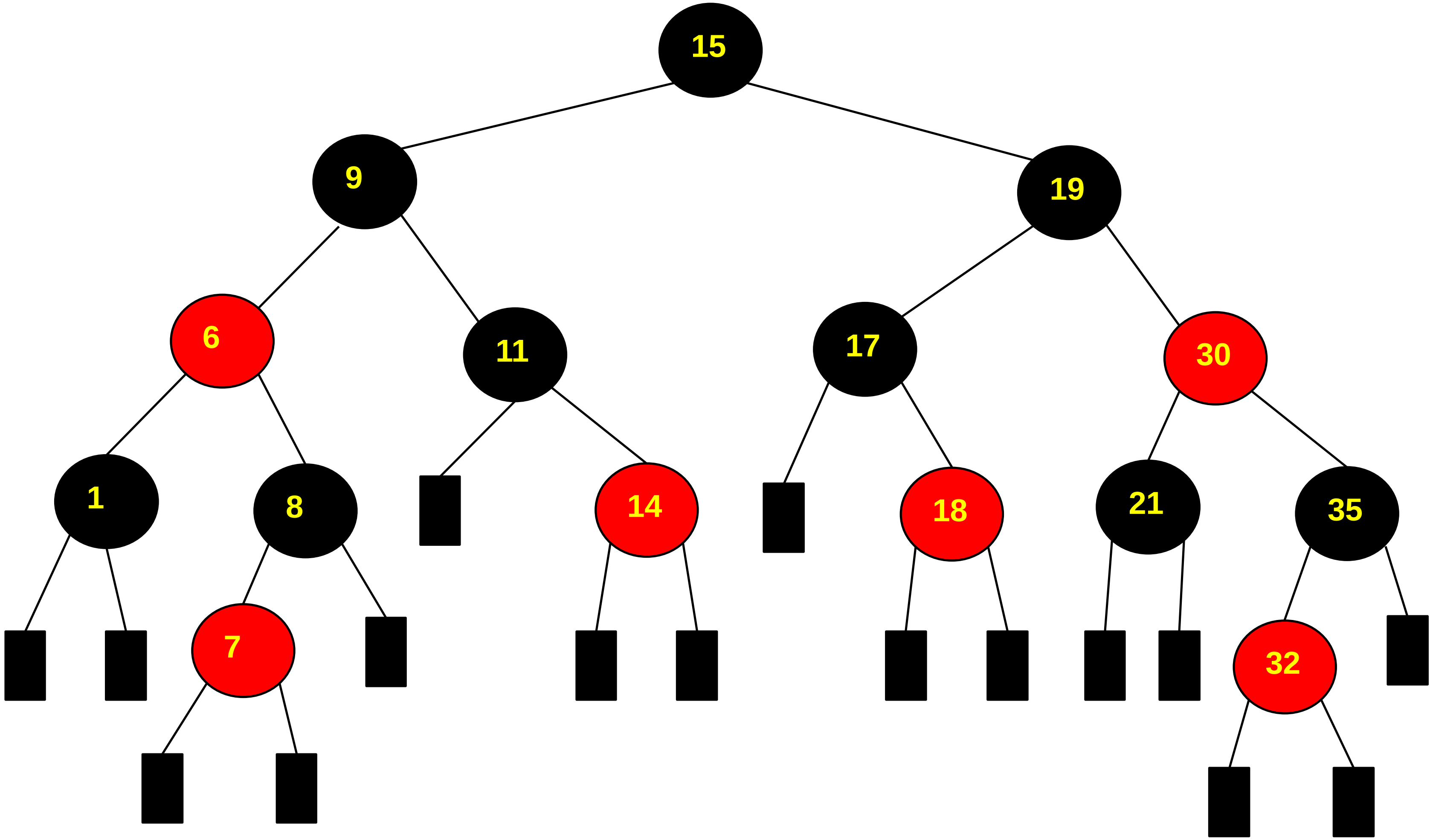
Delete a node? (6)
Delete 15



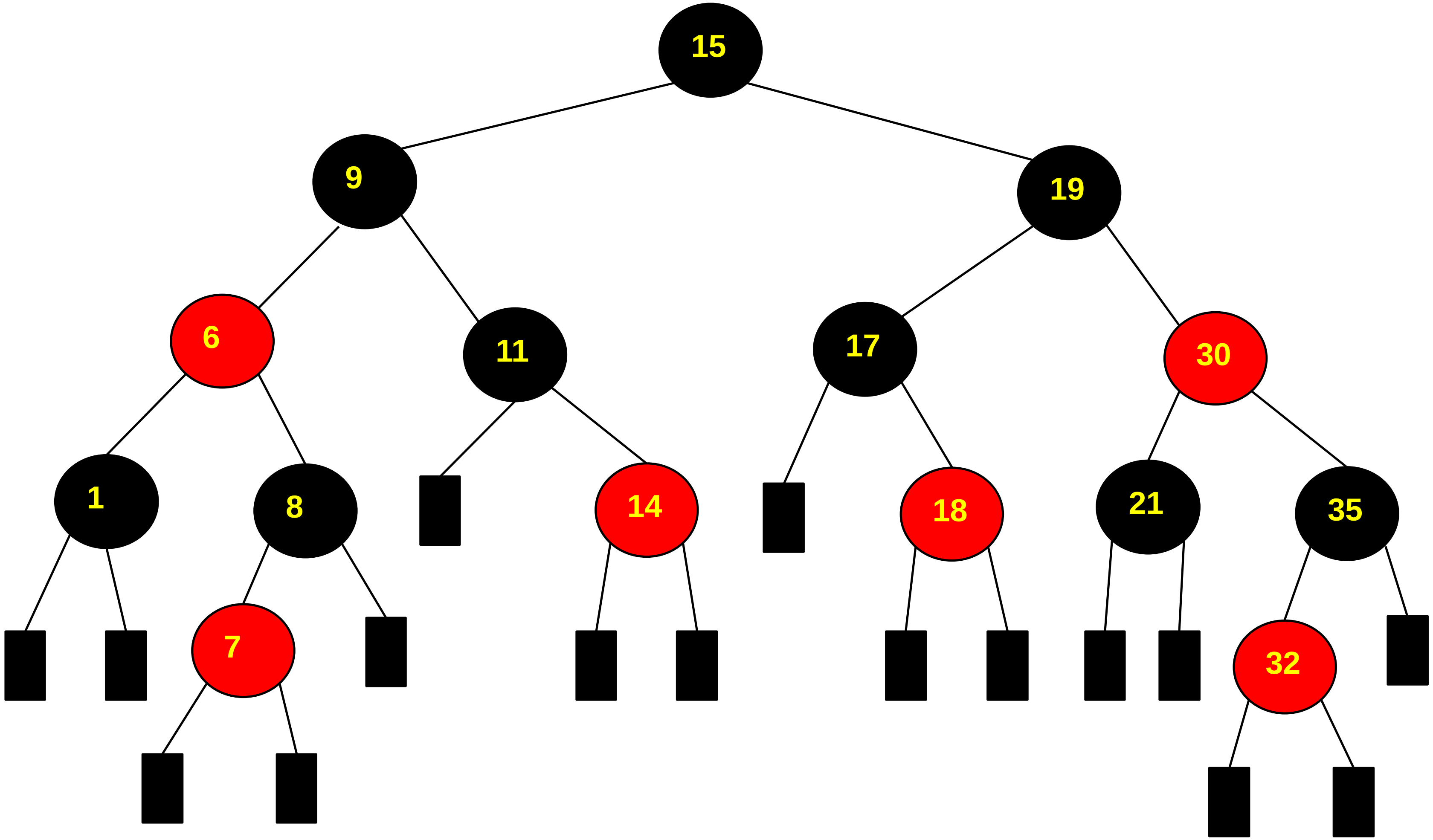
Delete a node? (7)
Delete 15



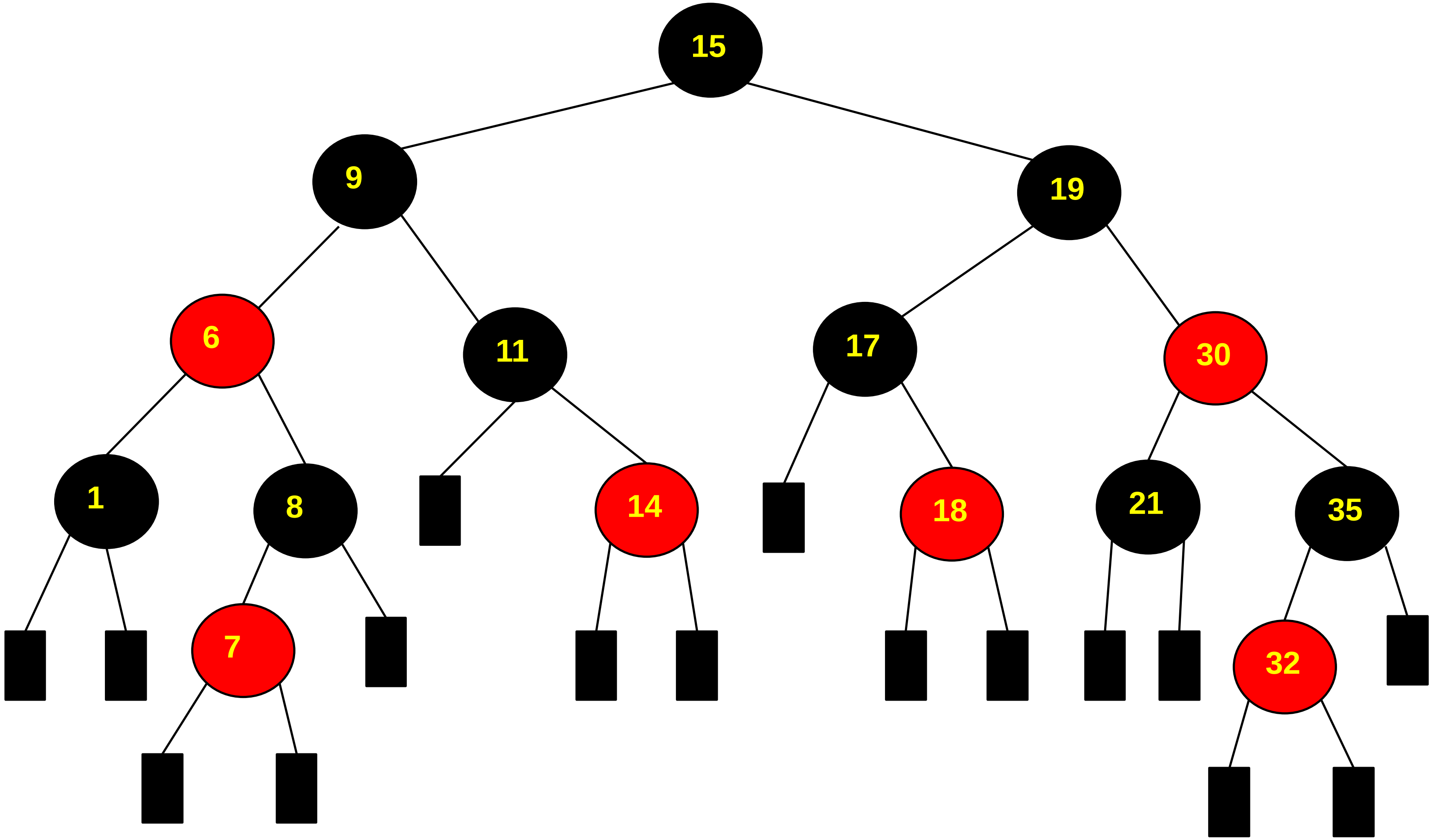
Node Insertion
Please read the book for details



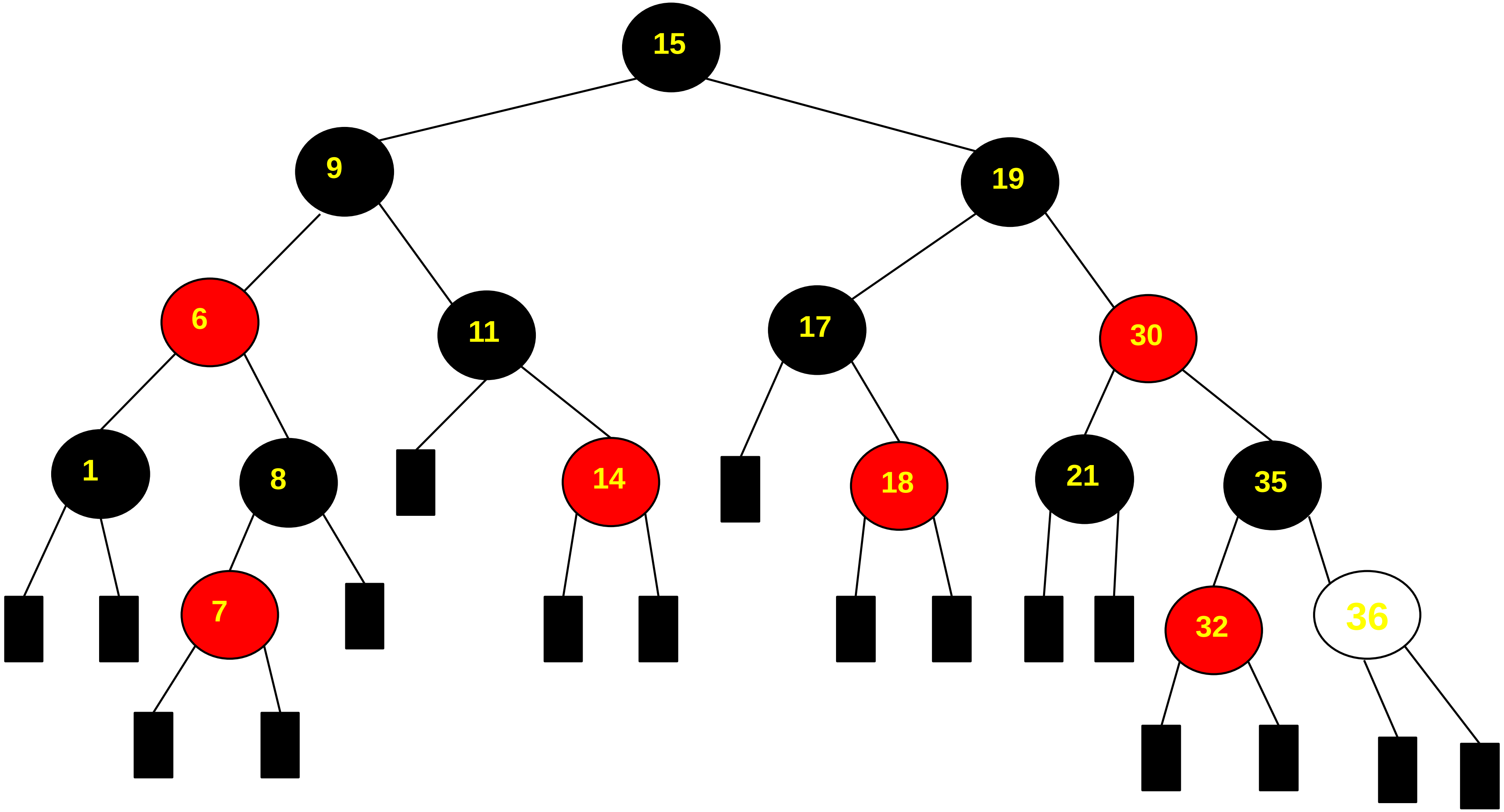
Node Insertion
Insert 36



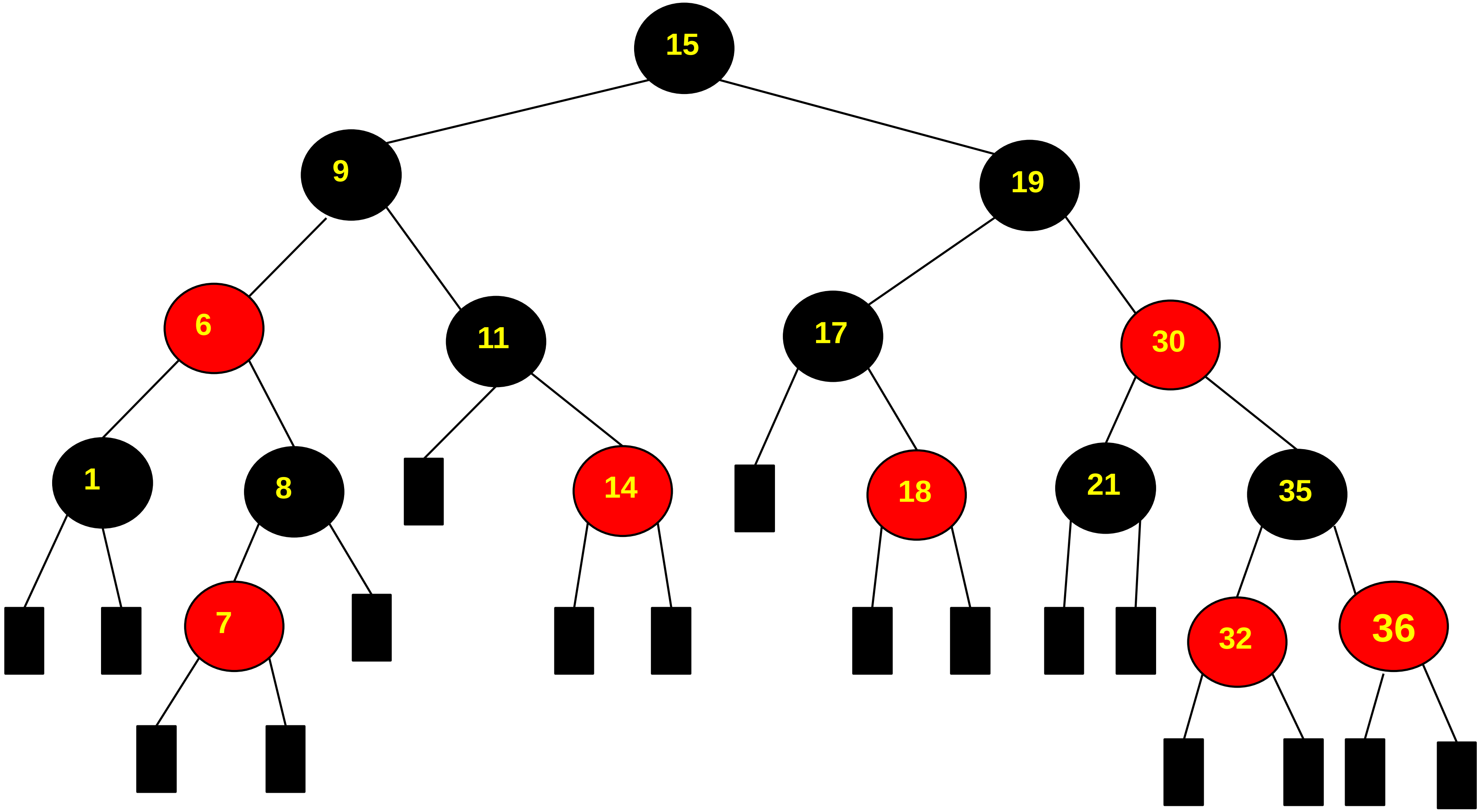
Node Insertion
Insert 36 (1)



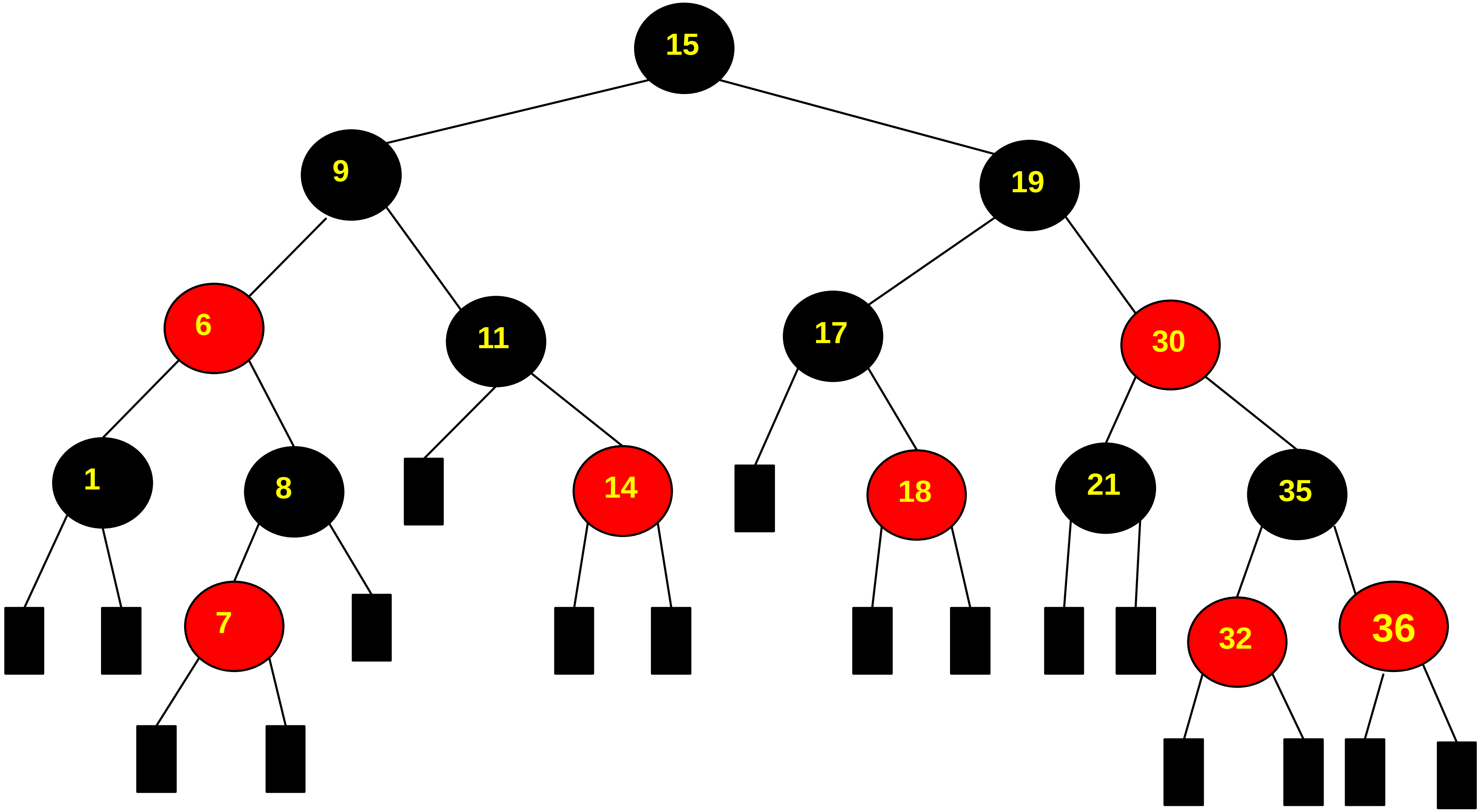
Node Insertion
Insert 36 (2)



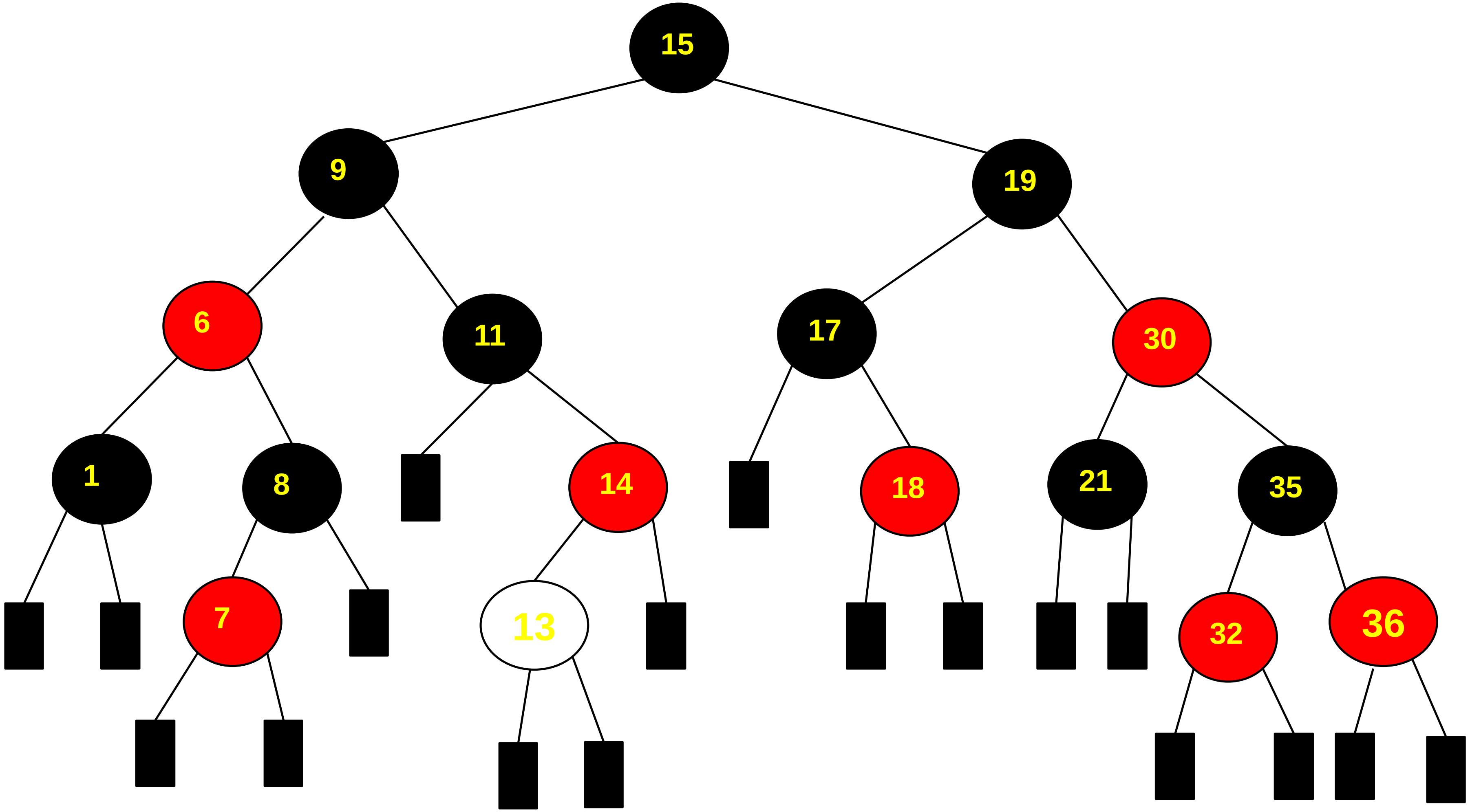
Node Insertion
Insert 36 (3)



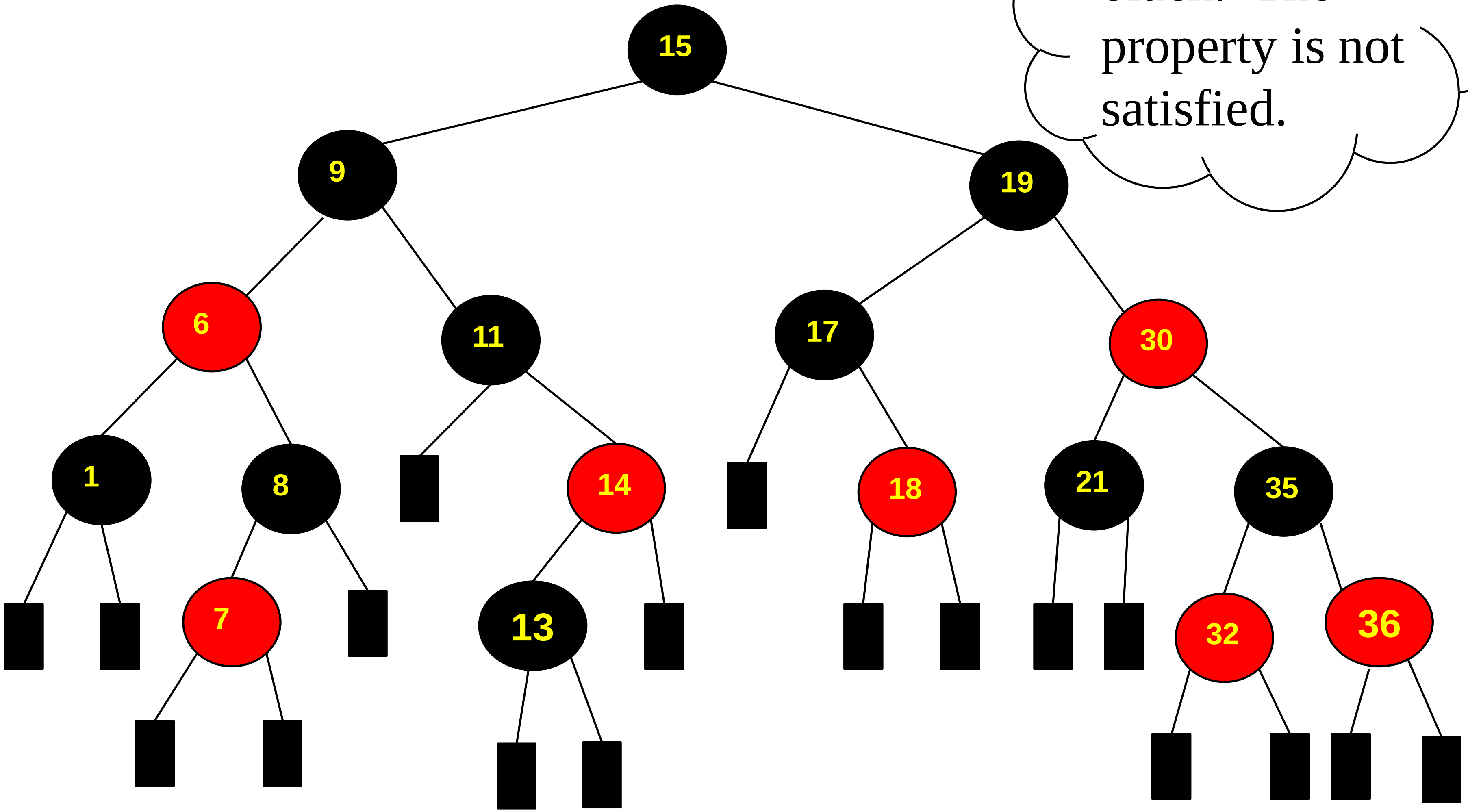
Node Insertion
Insert 13 (1)



Node Insertion
Insert 13 (2)

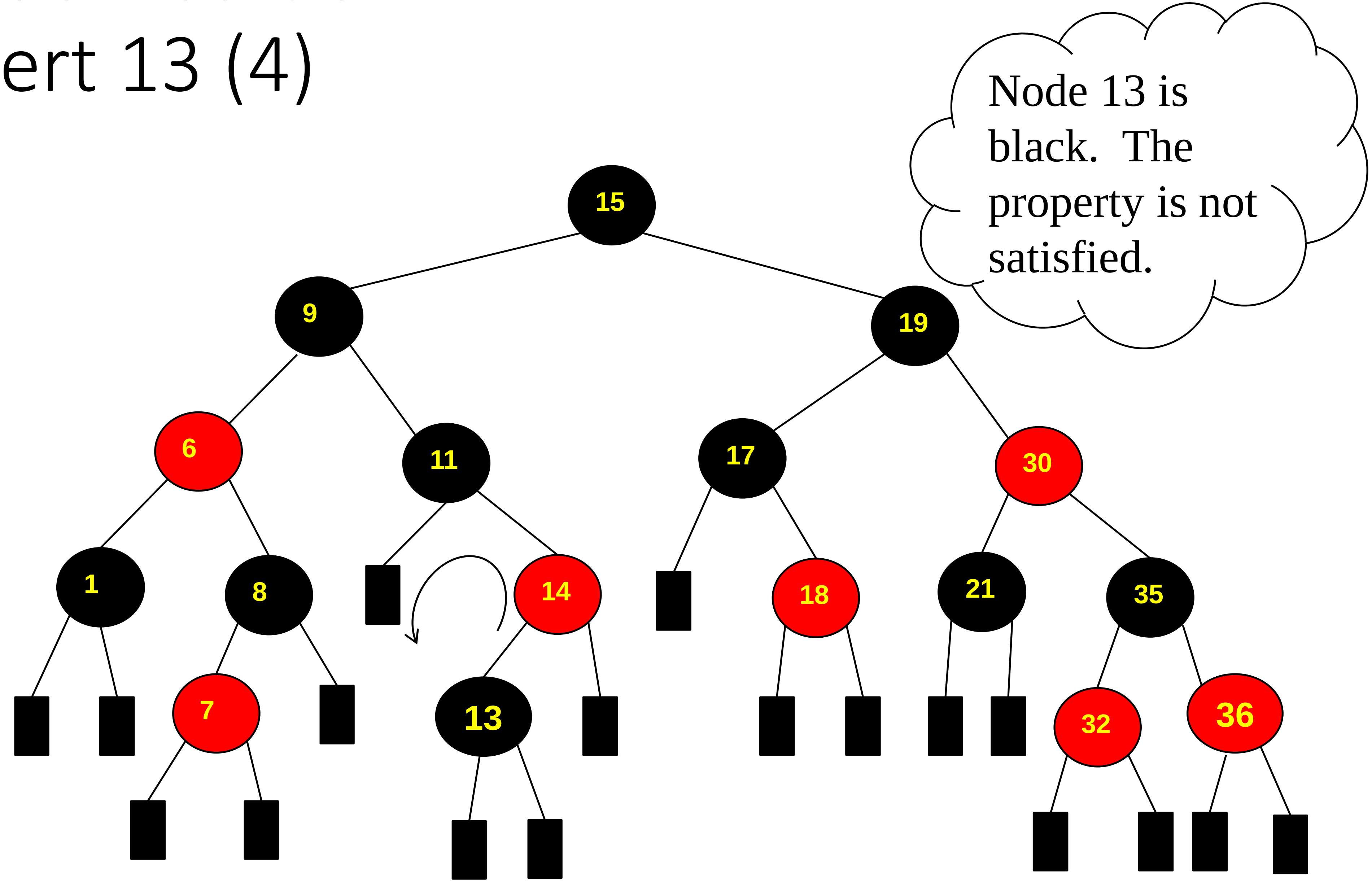


Node Insertion
Insert 13 (3)

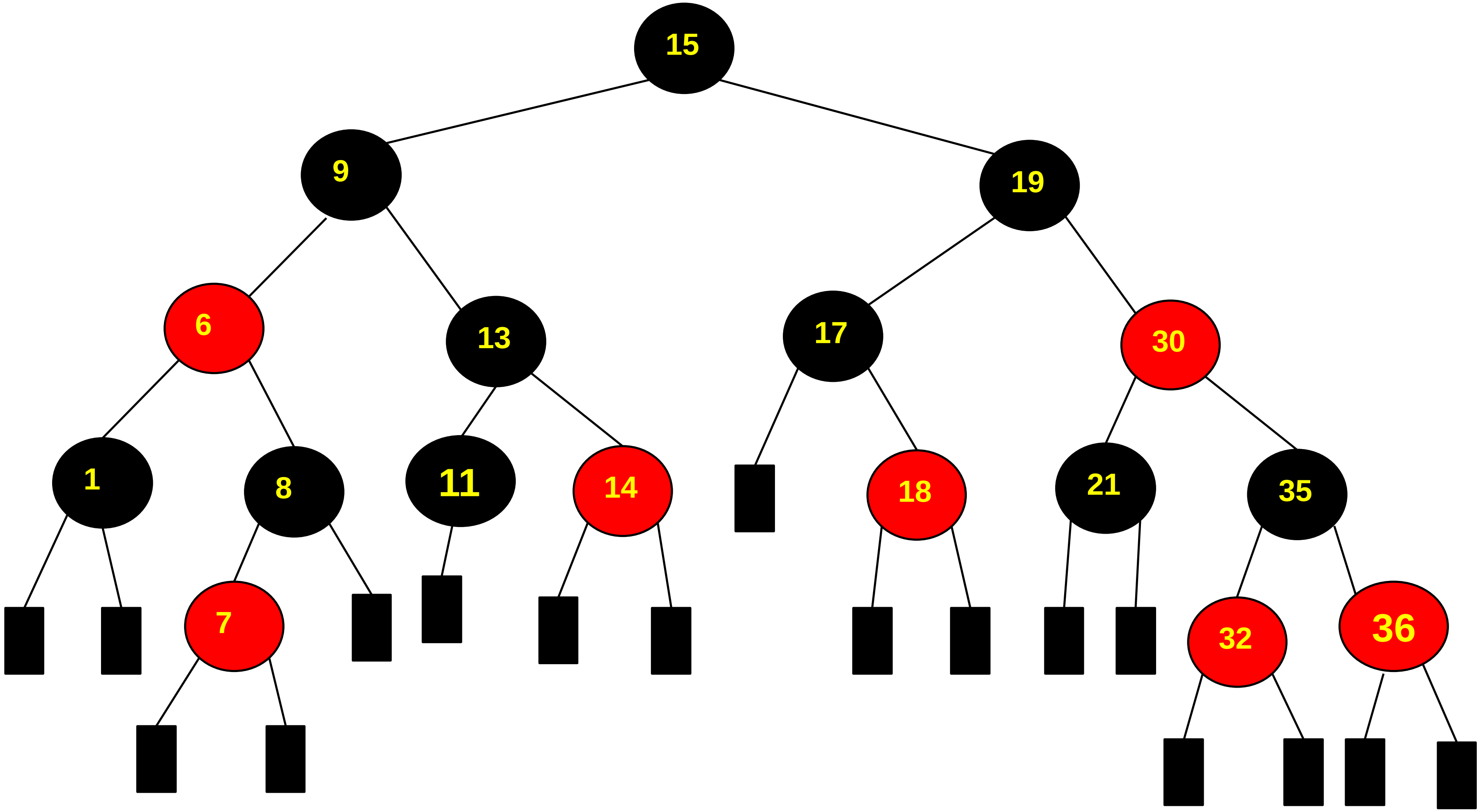


Node 13 is black. The property is not satisfied.

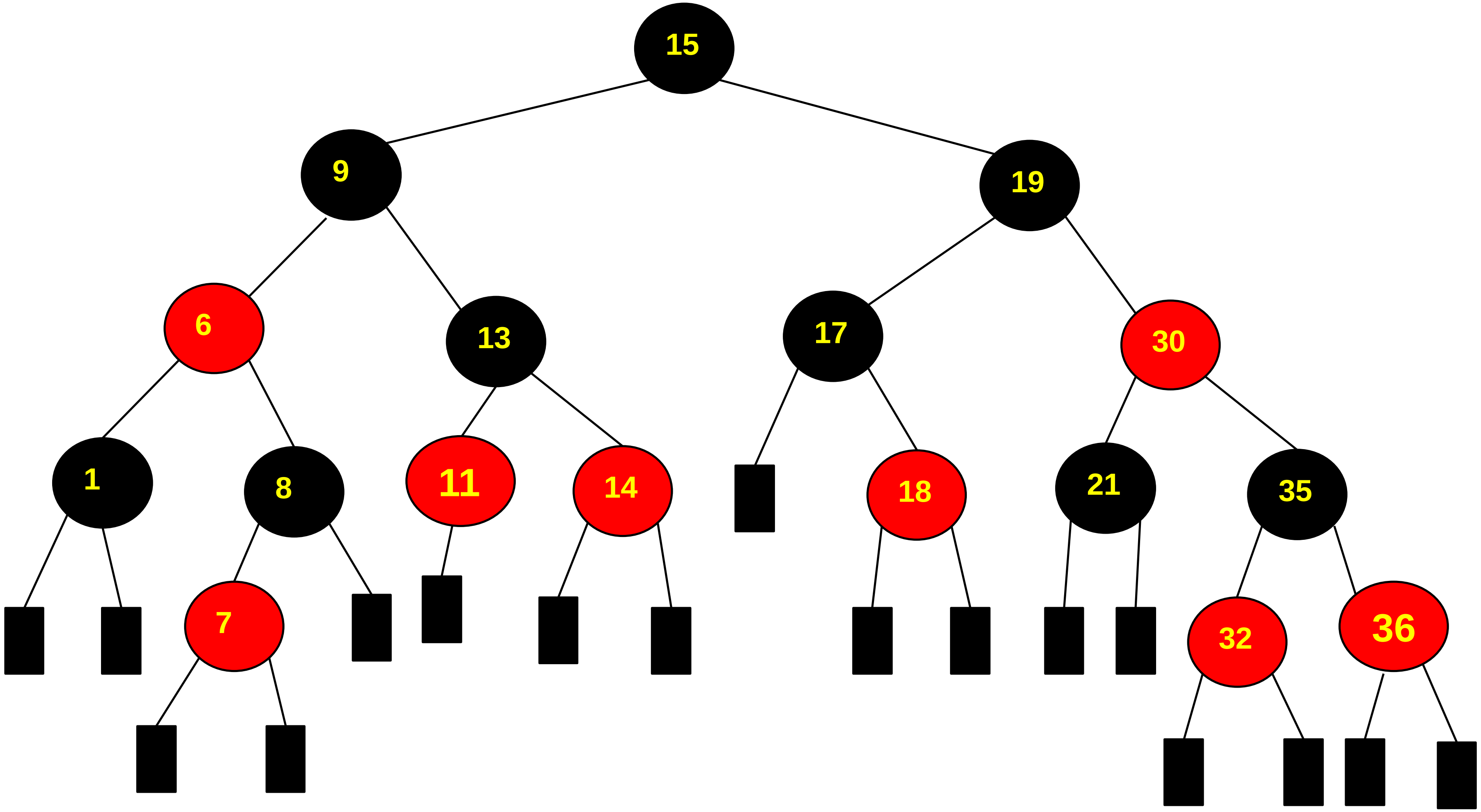
Node Insertion
Insert 13 (4)



Node Insertion
Insert 13 (5)



Node Insertion
Insert 13 (6)



Node Insertion

- New pair is placed in a new node. Insert it into the red-black tree.
- New node color options.
  - Black node: one root-to-external-node path has an extra black node (black pointer).
    - Difficult to remedy
  - Red node: one root-to-external-node path may have two consecutive red nodes (pointers).
    - Two ways: 1) perform color flips; 2) perform a rotation.

Node Insertion

- We can classify the cases into several types.
- Based on each type, certain rules are applied.
- Main idea: Maintain the tree properties at the end of the insertion process.