Binary Search Trees: Motivation

Runway Reservations

Problem: You are in charge of an airport with a single runway. Planes radio in to request a landing time. You need to schedule the landing times so that no planes land $< k$ minutes of each other.

Example with $k = 3$:

current set | request | decision

Runway Reservations

Problem: You are in charge of an airport with a single runway. Planes radio in to request a landing time. You need to schedule the landing times so that no planes land $< k$ minutes of each other.

Example with $k = 3$:

current set	request	decision
$\emptyset$	13	scheduled

Runway Reservations

Problem: You are in charge of an airport with a single runway. Planes radio in to request a landing time. You need to schedule the landing times so that no planes land $< k$ minutes of each other.

Example with $k = 3$:

current set	request	decision
$\emptyset$	13	scheduled
$\{13\}$	7	scheduled

Runway Reservations

Problem: You are in charge of an airport with a single runway. Planes radio in to request a landing time. You need to schedule the landing times so that no planes land $< k$ minutes of each other.

Example with $k = 3$:

current set	request	decision
$\emptyset$	13	scheduled
$\{13\}$	7	scheduled
$\{7, 13\}$	9	denied

Runway Reservations

Problem: You are in charge of an airport with a single runway. Planes radio in to request a landing time. You need to schedule the landing times so that no planes land $< k$ minutes of each other.

Example with $k = 3$:

current set	request	decision
$\emptyset$	13	scheduled
$\{13\}$	7	scheduled
$\{7, 13\}$	9	denied
$\{7, 13\}$	22	scheduled

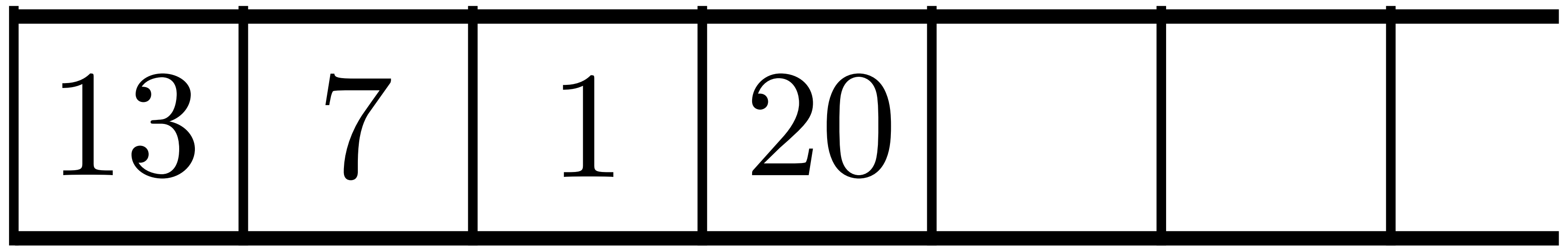
Abstracting the problem

We want to maintain a set of keys (landing times). We want to check if a key satisfies the time constraint, and if so insert it into the database. We also want to be able to remove keys.

What data structure is good for this problem?

Unordered Vector

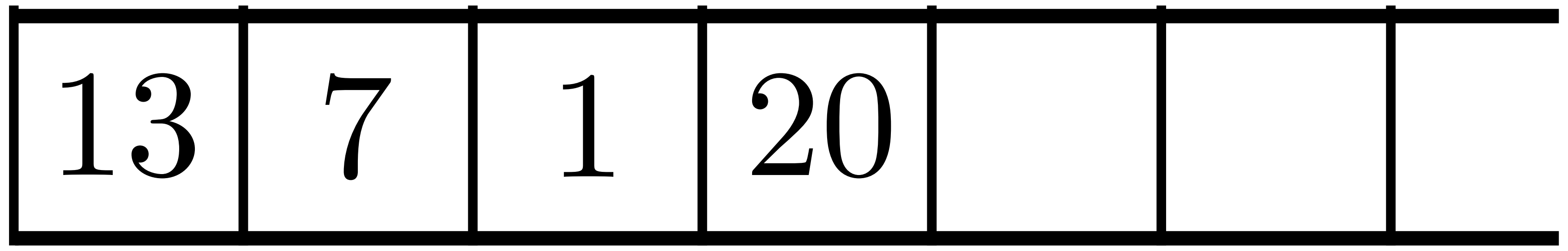
We check if a request is valid and if so insert it at the end of a vector.



What is the problem with this solution?

Unordered Vector

We check if a request is valid and if so insert it at the end of a vector.



What is the problem with this solution?

insertion is $\Theta(1)$.

Unordered Vector

We check if a request is valid and if so insert it at the end of a vector.

13	7	1	20			

What is the problem with this solution?

insertion is $\Theta(1)$.

checking the constraint is $\Theta(n)$.

Sorted Vector

How about if we maintain a sorted vector?

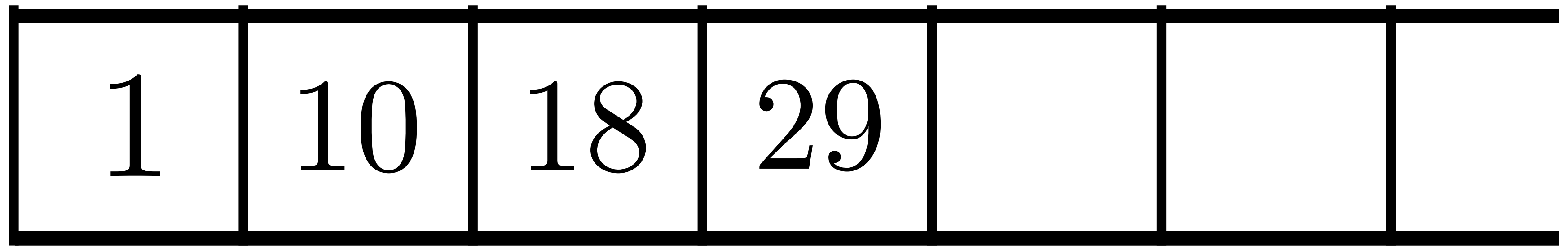
1	10	18	29			

Now we can check if a request is valid via binary search.

Checking the constraint is $\Theta(\log n)$.

Sorted Vector

How about if we maintain a sorted vector?

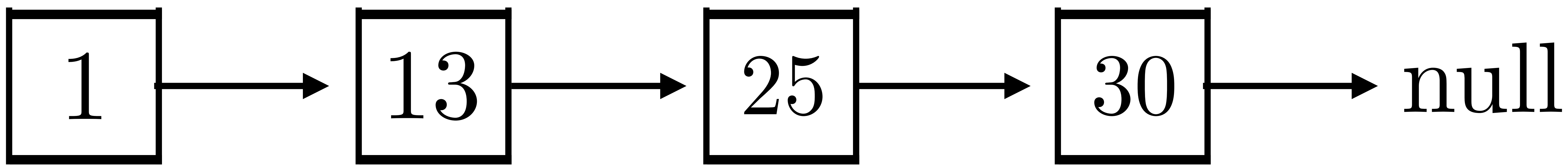


Now we can check if a request is valid via binary search.

Checking the constraint is $\Theta(\log n)$.

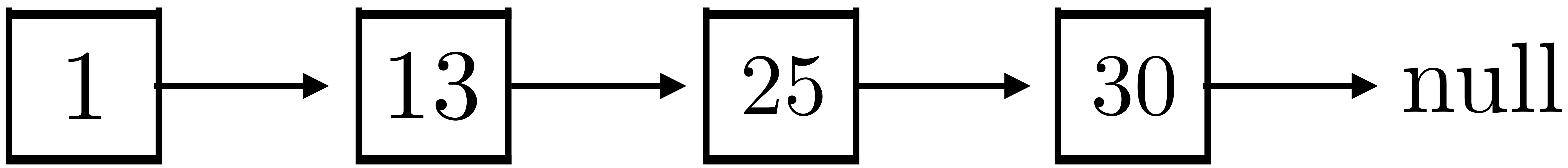
What is the problem with this solution?

Sorted Linked List



What is the problem with this solution?

Sorted Linked List



What is the problem with this solution?

Checking the constraint is $\Theta(n)$.

Set/Map

If you thought of std::set or std::map, you are exactly right!

A map provides the functionality needed for the runway reservation problem.

Map is typically implemented with a (balanced) binary search tree, the data structure we will see today.

Binary search trees combine the benefits of the sorted array and linked list approaches.

Formalising Runway Reservations

Abstract Data Type

Let's develop a set of operations to solve the runway reservation problem.

We want to maintain a set of keys. Each key can be associated with some data. We think of them as $\{\text{key}, \text{record}\}$ pairs.

We have two dynamic operations that modify the database:

$$\texttt{insert}(\text{key}, \text{record})$$

$$\texttt{remove}(\text{key})$$

Abstract Data Type

We want to maintain a set of keys. Each key can be associated with some data. We think of them as $(\text{key}, \text{record})$ pairs.

Operations to extract information from the database:

$\texttt{contains}(\text{key})$	check if a key is in the database
$\texttt{successor}(\text{key})$	find the next largest key in the database
$\texttt{predecessor}(\text{key})$	find the next smallest key in the database

For successor and predecessor we will assume the argument is already in the database.

Abstract Data Type

A somewhat roundabout solution to the runway reservation problem.

Request for landing time t comes in with plane info in data.

Run $\texttt{contains}(t)$. If t is already in the database, reject. Otherwise do:

$\texttt{insert}(t, \text{data})$

$\text{prev} = \texttt{predecessor}(t)$

$\text{next} = \texttt{successor}(t)$

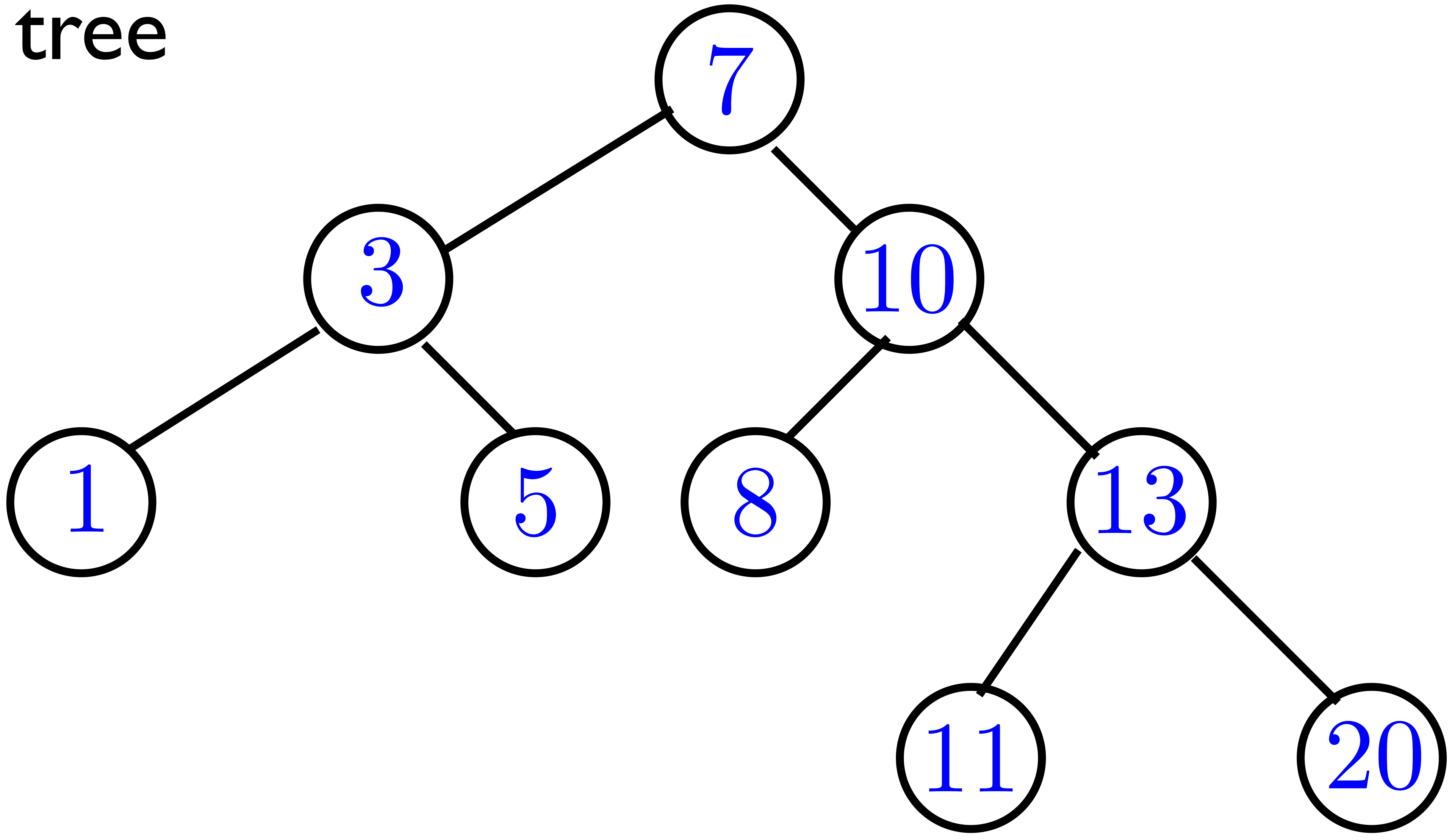
If $t - \text{prev} \geq k$ and $\text{next} - t \geq k$ then we are done. Otherwise reject and

$\texttt{remove}(t)$

Binary Search Trees

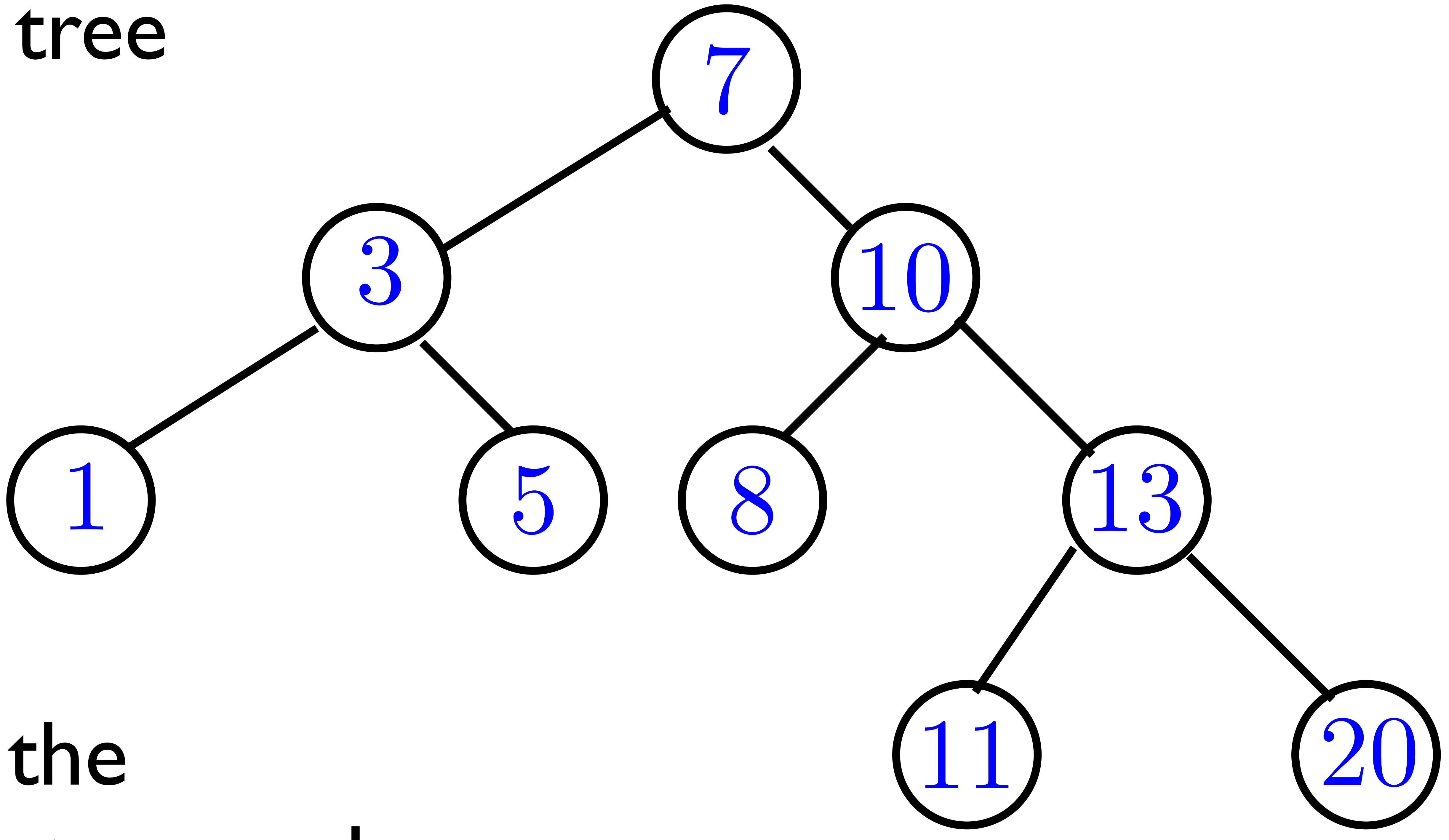
Binary Search Tree

A binary search tree (BST) is a binary tree with keys labelling the vertices.



Binary Search Tree

A binary search tree (BST) is a binary tree with keys labelling the vertices.



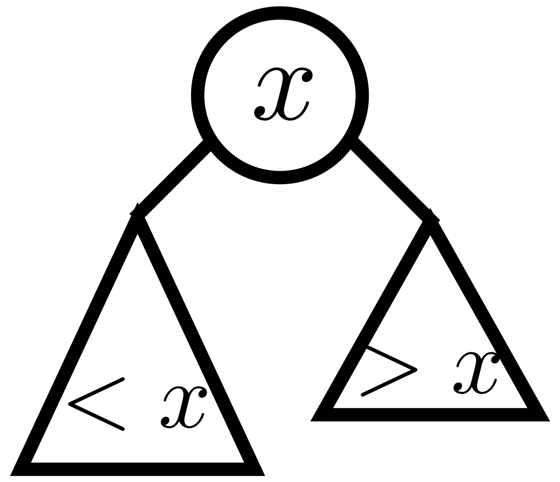
BST property: for any vertex v,

1) all keys in the subtree rooted at the left child of v are less than the key at v , and

2) all keys in the subtree rooted at the right child of v are greater than the key at v .

Binary Search Tree

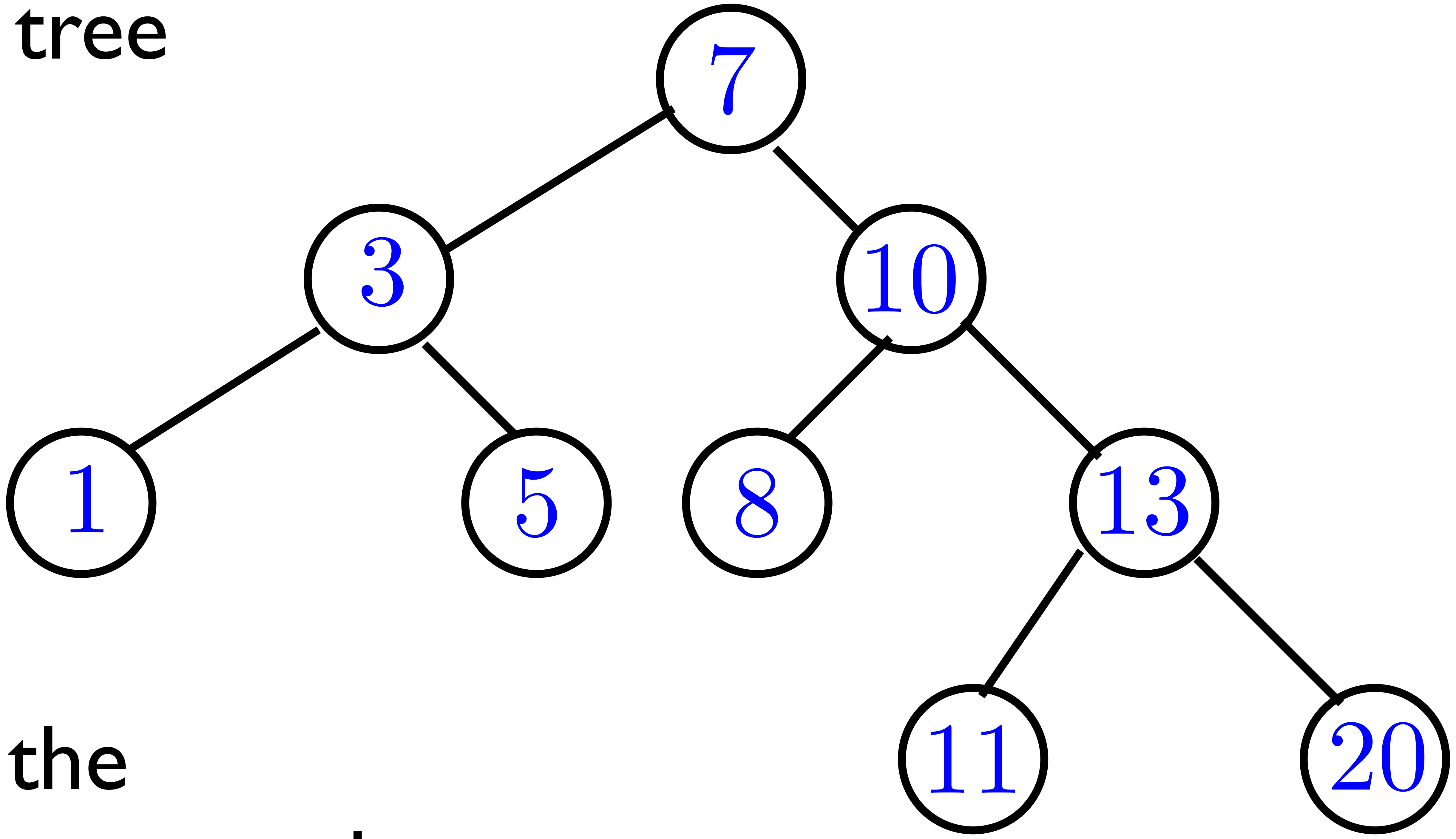
A binary search tree (BST) is a binary tree with keys labelling the vertices.



BST property: for any vertex v,

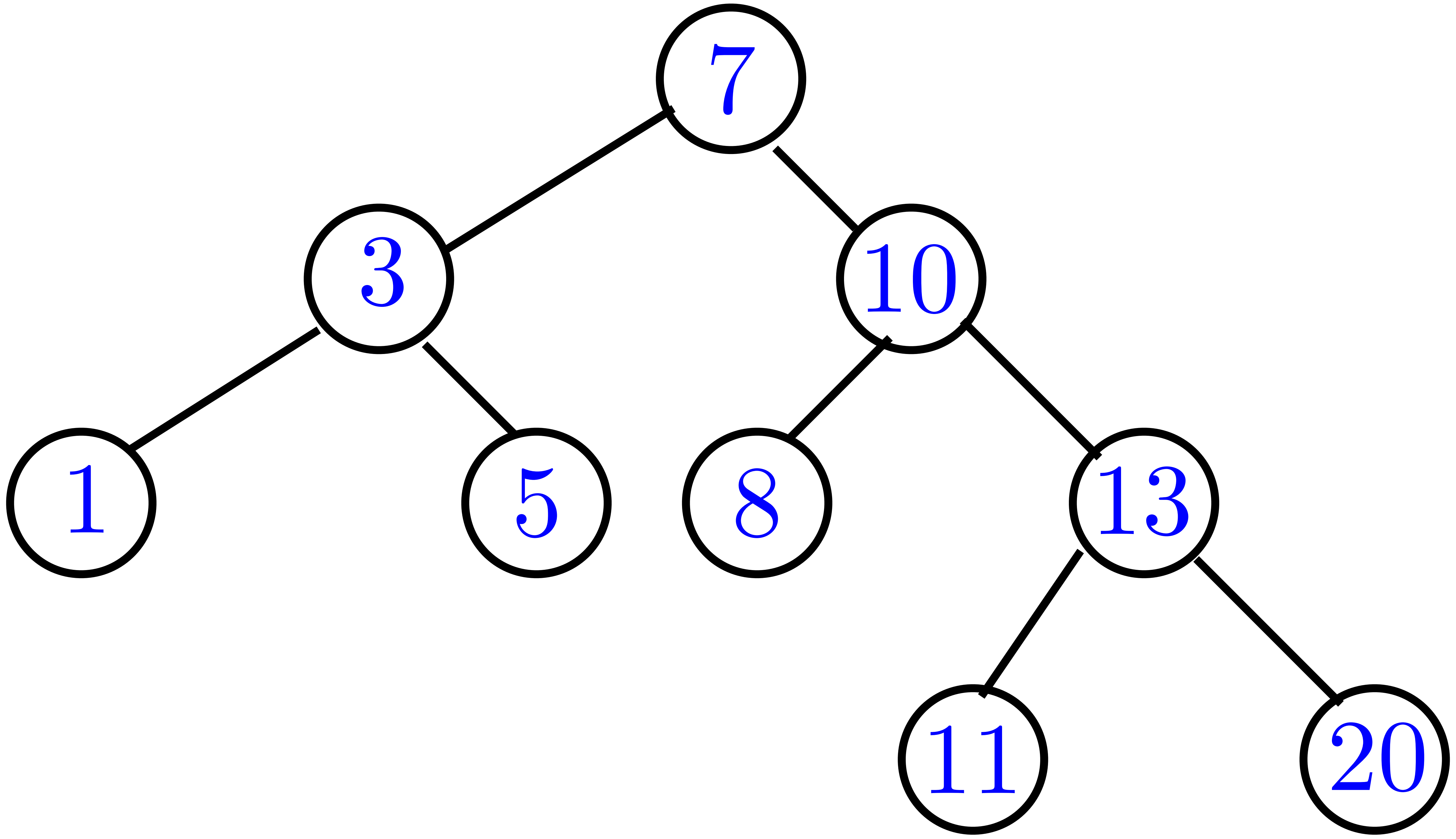
1) all keys in the subtree rooted at the left child of v are less than the key at v , and

2) all keys in the subtree rooted at the right child of v are greater than the key at v .



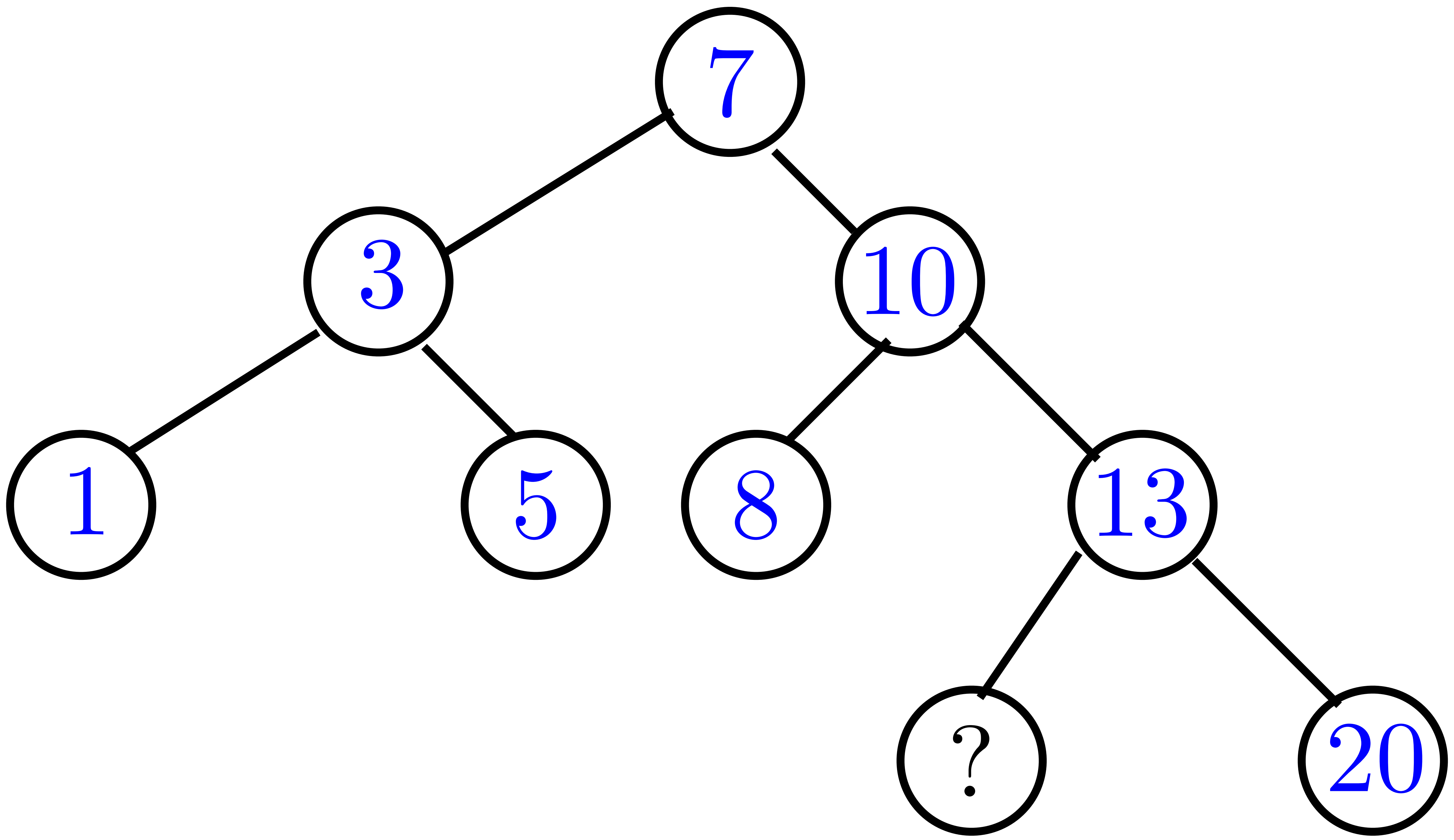
Question

Where is the minimum key in a BST?



Question

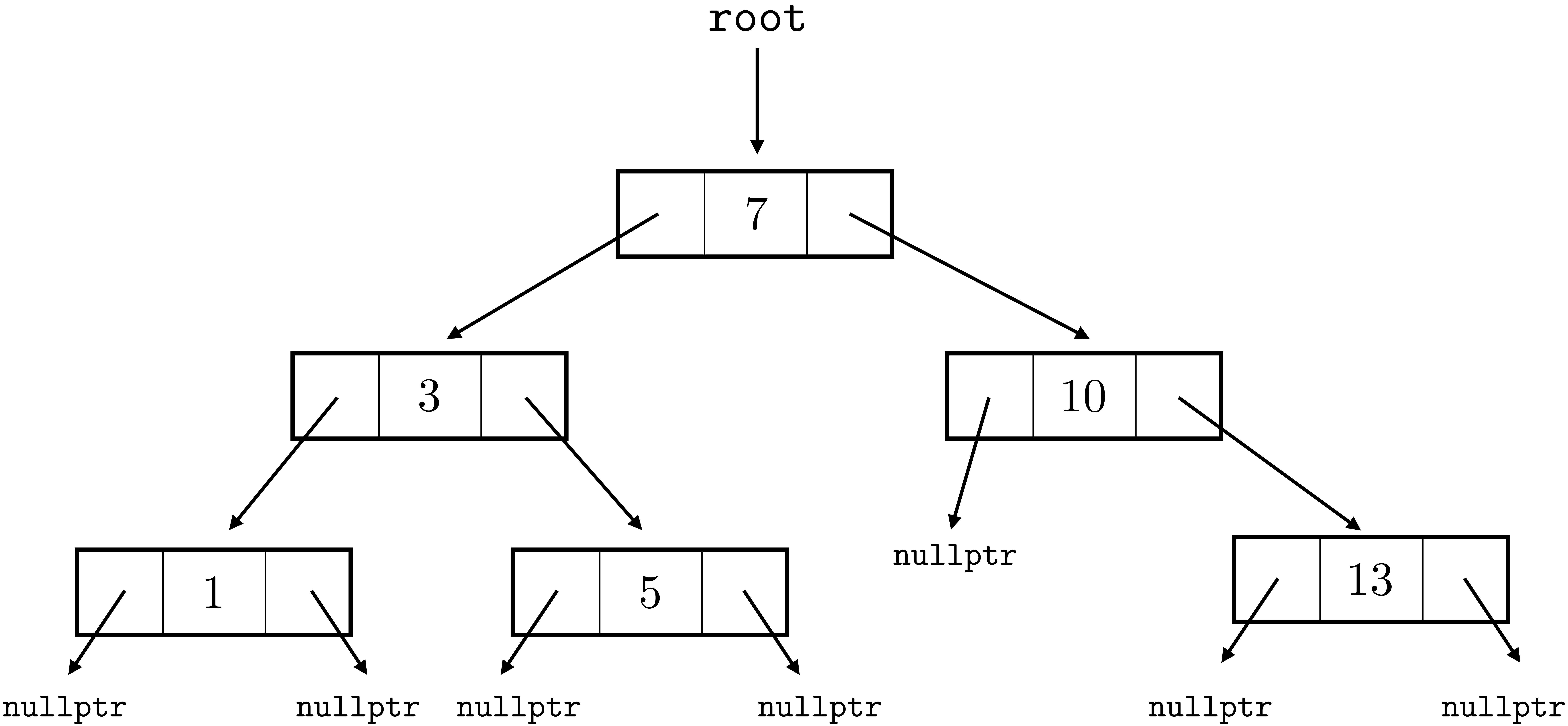
What integers could be placed at the question mark?



Representation of a BST

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

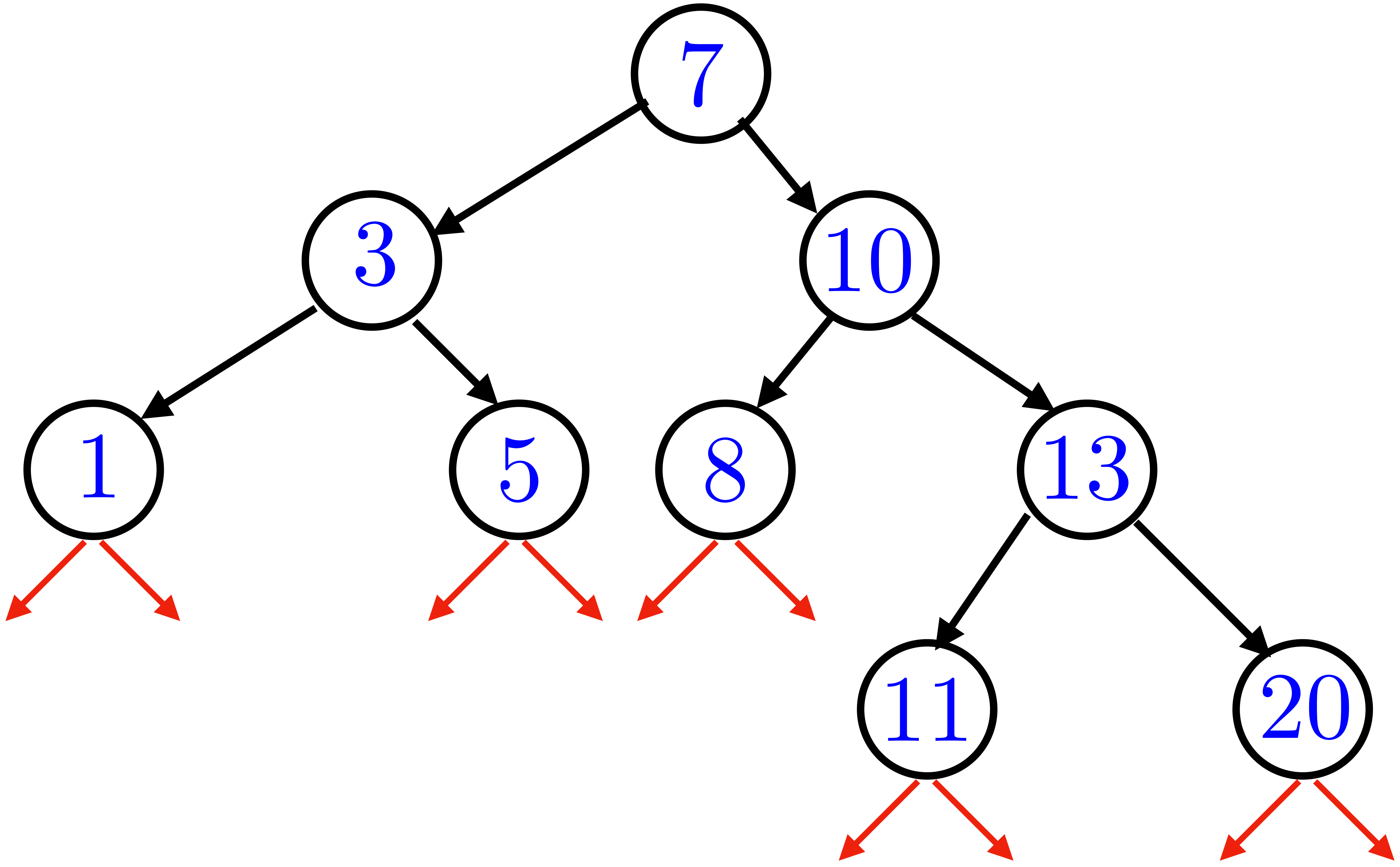
```
class BST {
  private:
    Node* root;

  public:
    BST();
    ~BST();

    void insert(int k);
    void remove(int k);
    Node* contains(int k);
    Node* successor(int k);
    Node* predecessor(int k);
};
```


root
7
3
10
1
5
nullptr
13
nullptr
nullptr
nullptr
nullptr
nullptr
nullptr

Contains

Let us first see how to check if a key is in a BST.

`contains`(11)

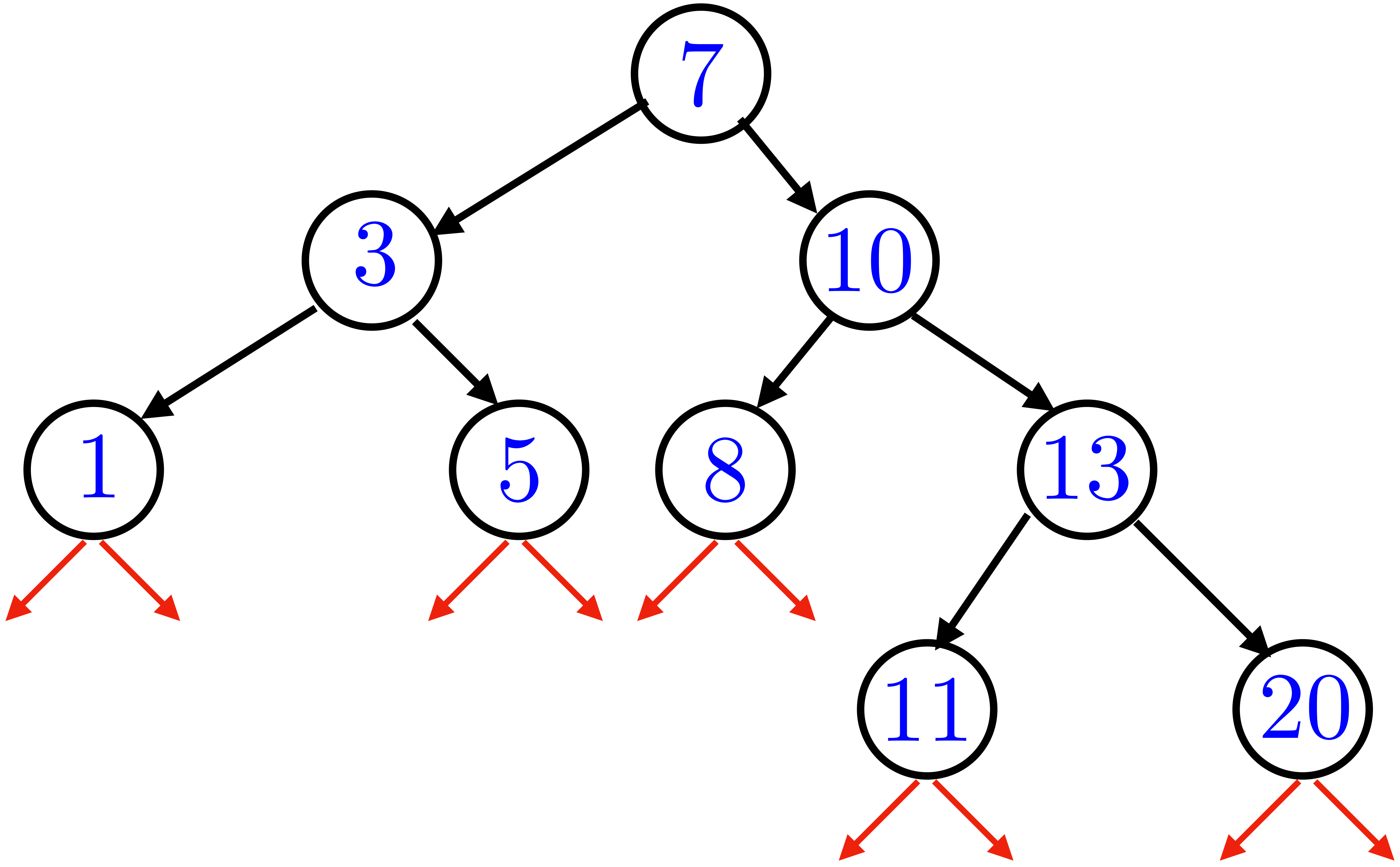


Contains

Let us first see how to check if a key is in a BST.

$\texttt{contains}(11)$

We start at the root and compare 11 to the key at the current vertex.



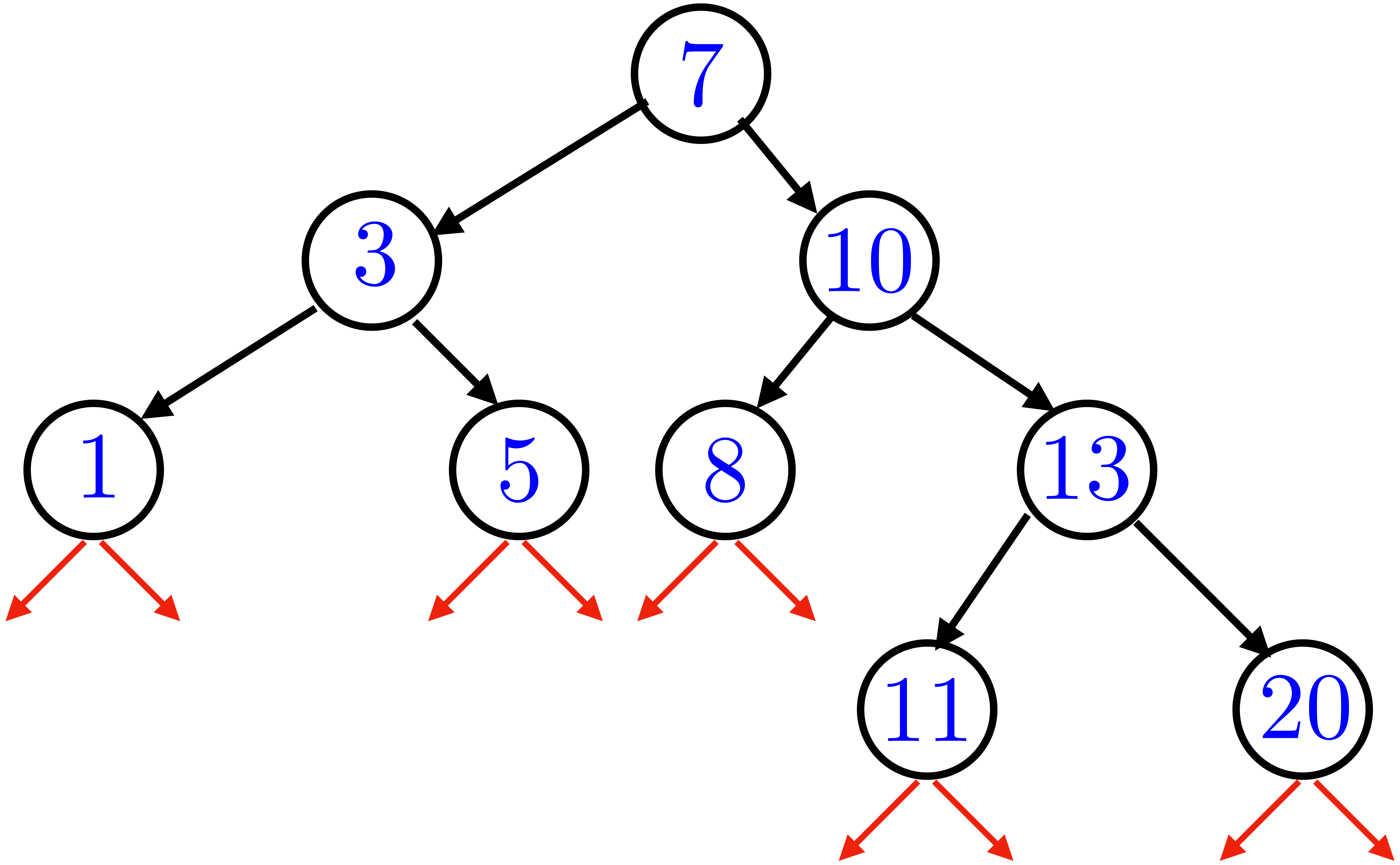
Contains

Let us first see how to check if a key is in a BST.

`contains`(11)

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



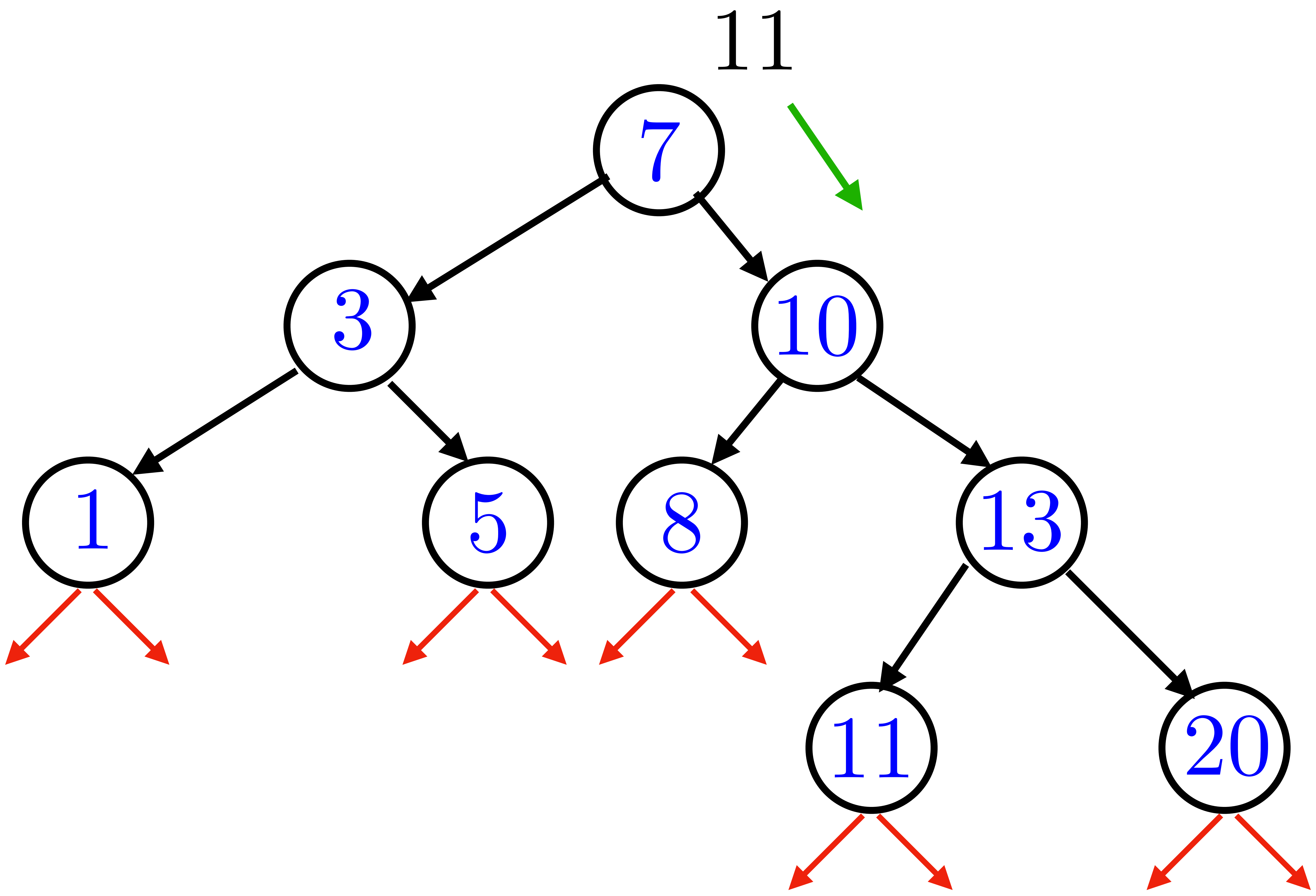
Contains

Let us first see how to check if a key is in a BST.

$\texttt{contains}(11)$

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



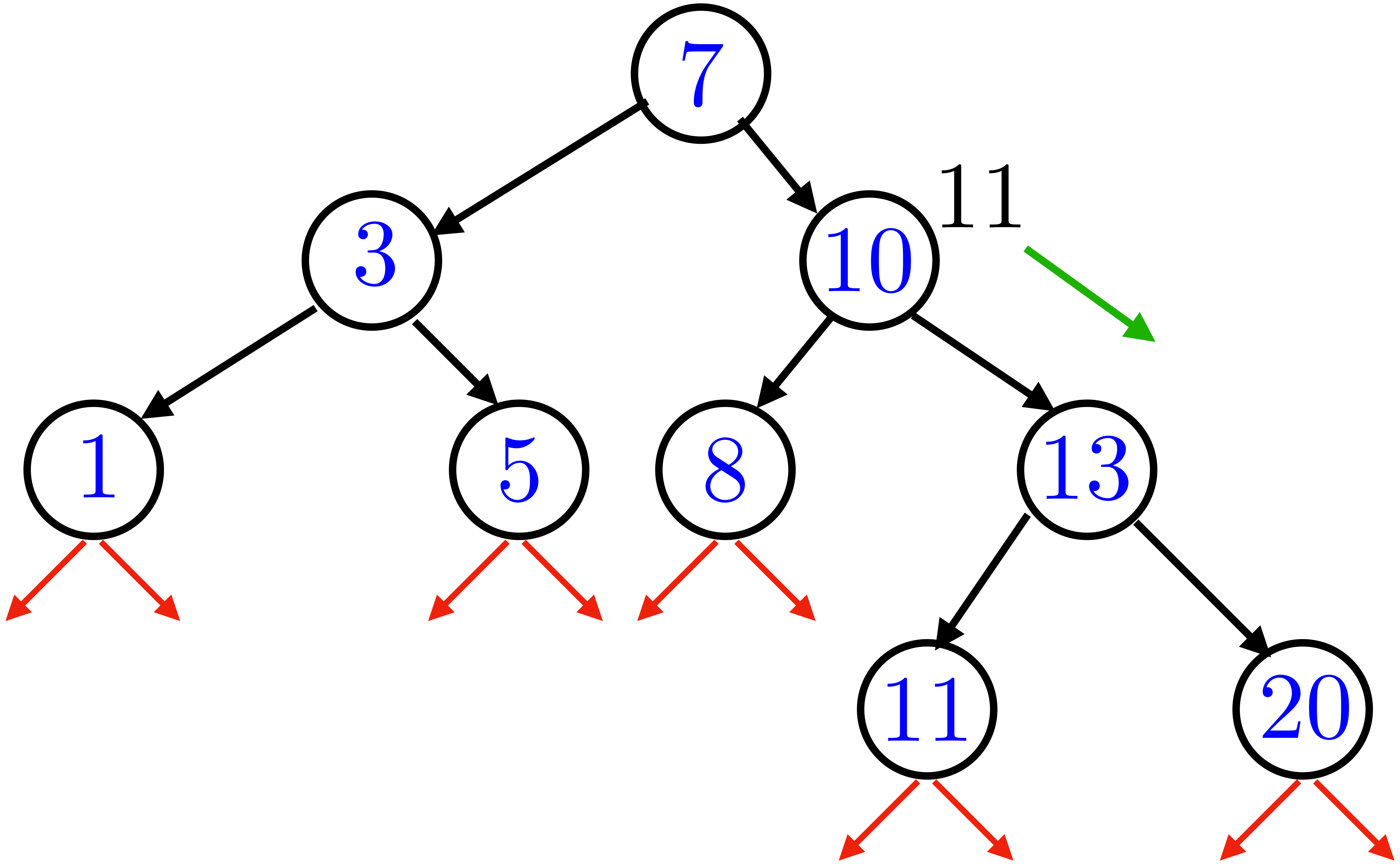
Contains

Let us first see how to check if a key is in a BST.

$\texttt{contains}(11)$

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



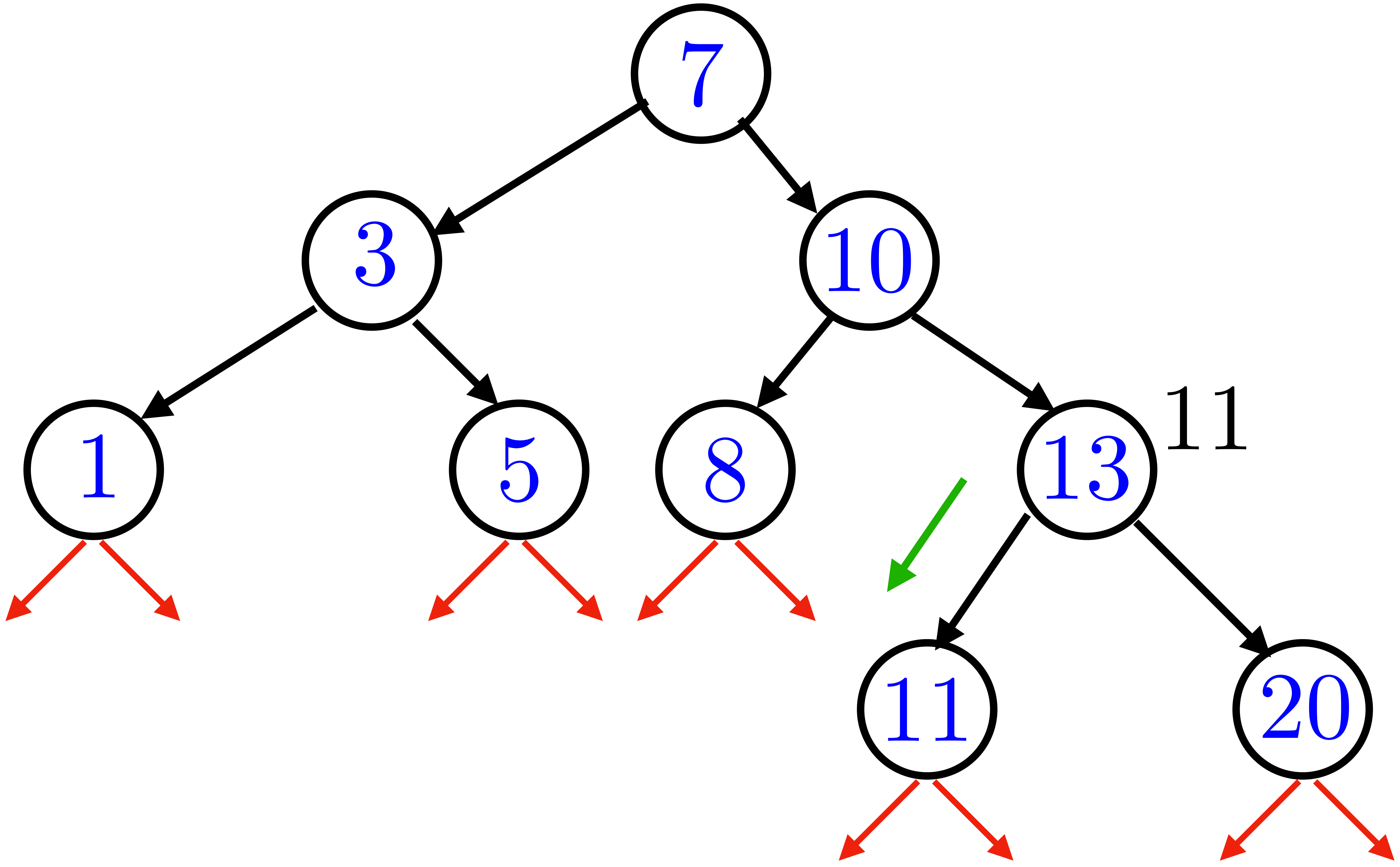
Contains

Let us first see how to check if a key is in a BST.

$\texttt{contains}(11)$

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



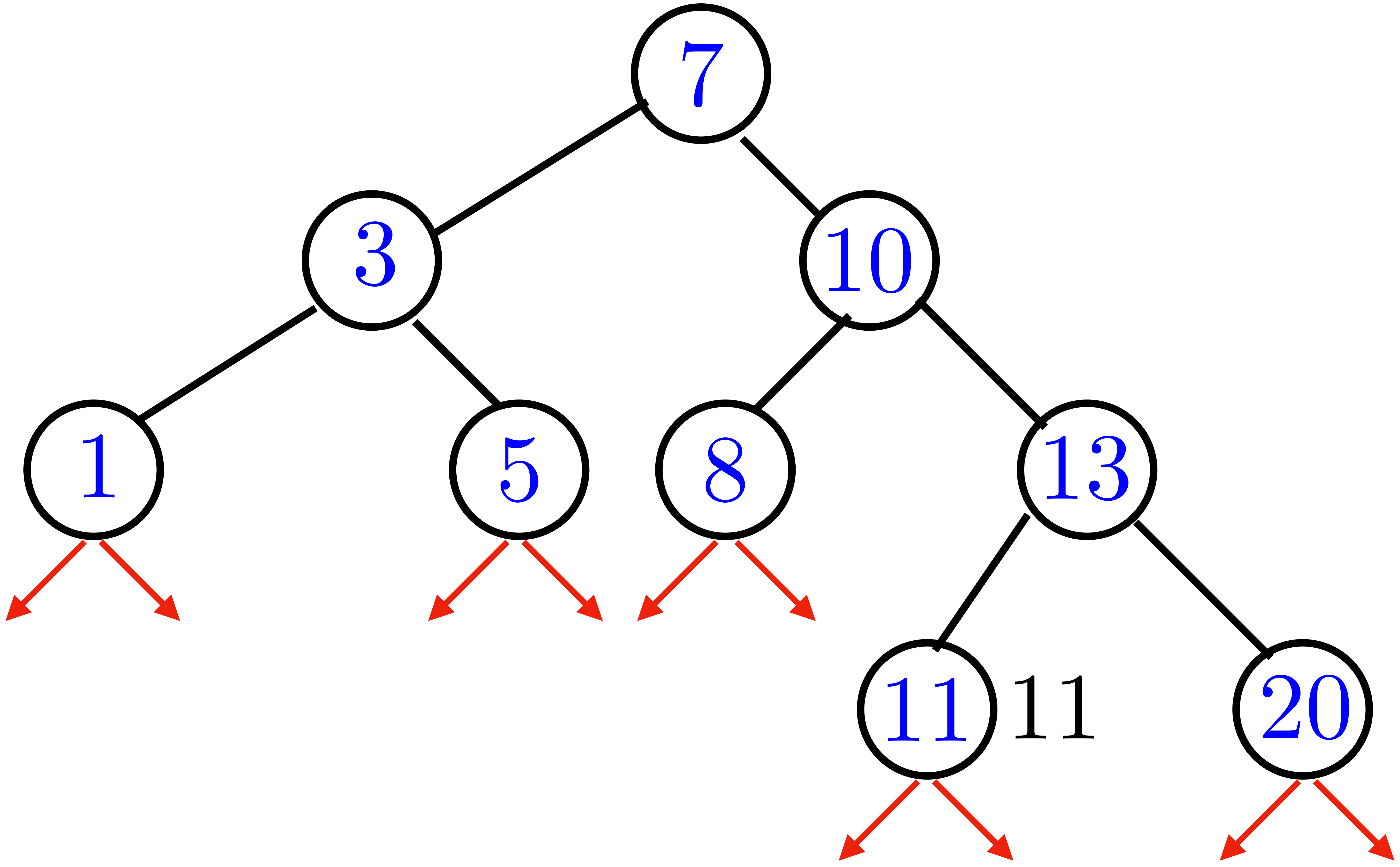
Contains

Let us first see how to check if a key is in a BST.

$\texttt{contains}(11)$

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



Contains

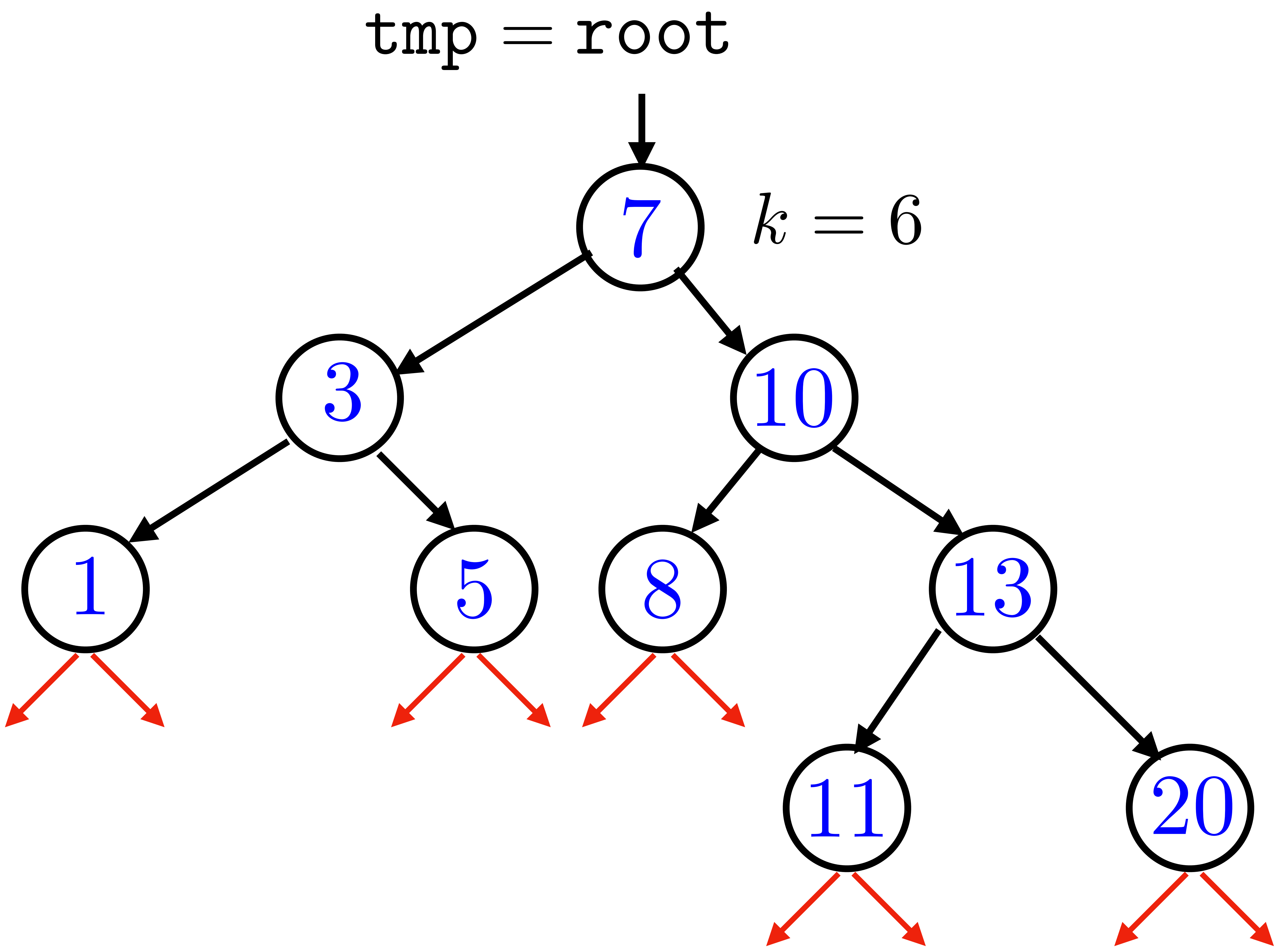
```
Node* contains(int k)
{
    Node* tmp = root;
    while(tmp != nullptr && tmp->key != k)
    {
        if(k < tmp->key)
        {
            tmp = tmp->left;
        }
        else
        {
            tmp = tmp->right;
        }
    }
    return tmp;
}
```

Contains (Miss)

Let's look at another example with a key not in the database.

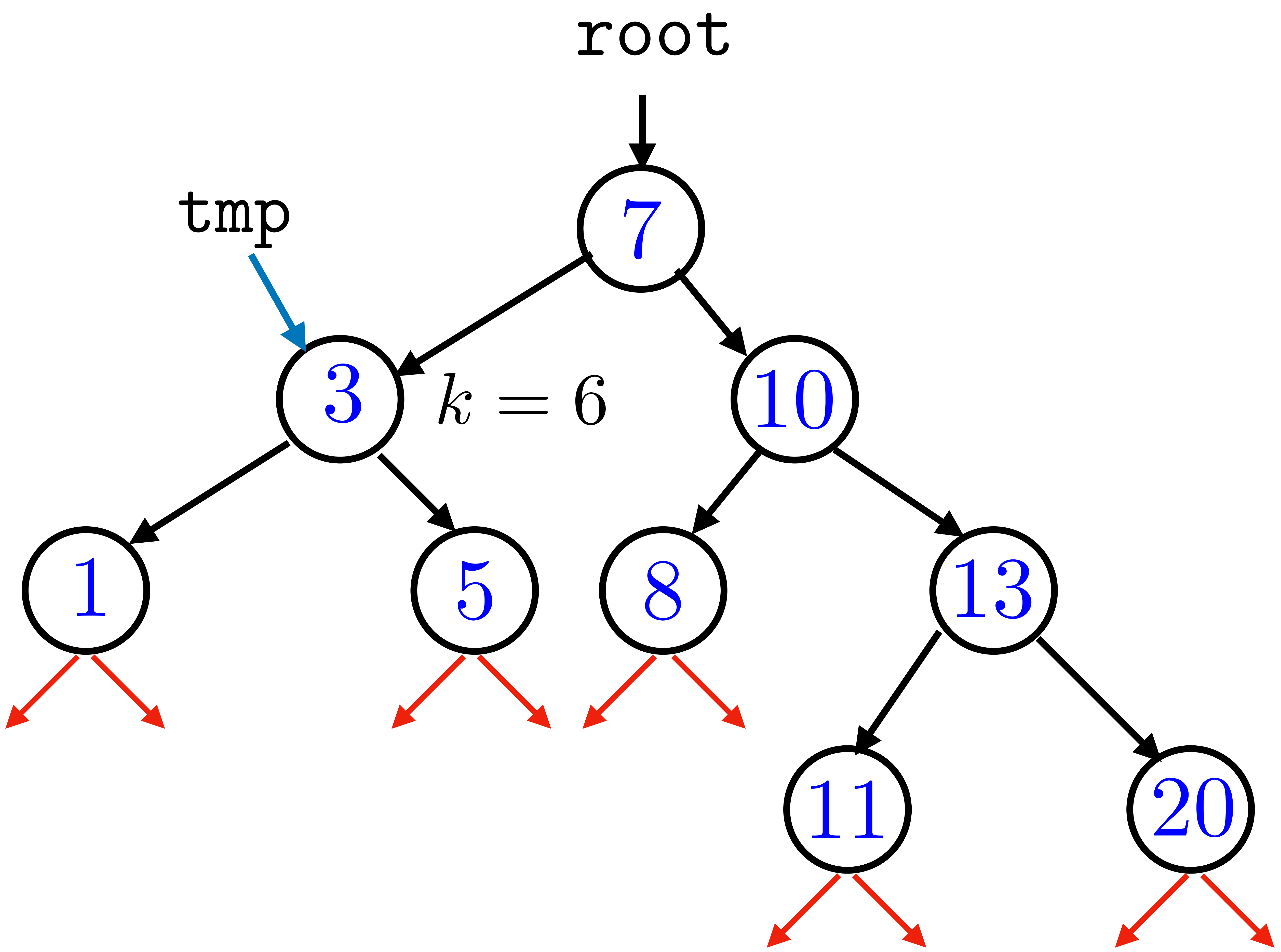
$\texttt{contains}(6)$

```
Node* contains(int k)
{
    Node* tmp = root;
    while(tmp != nullptr && tmp->key != k)
    {
        if(k < tmp->key)
        {
            tmp = tmp->left;
        }
        else
        {
            tmp = tmp->right;
        }
    }
    return tmp;
}
```



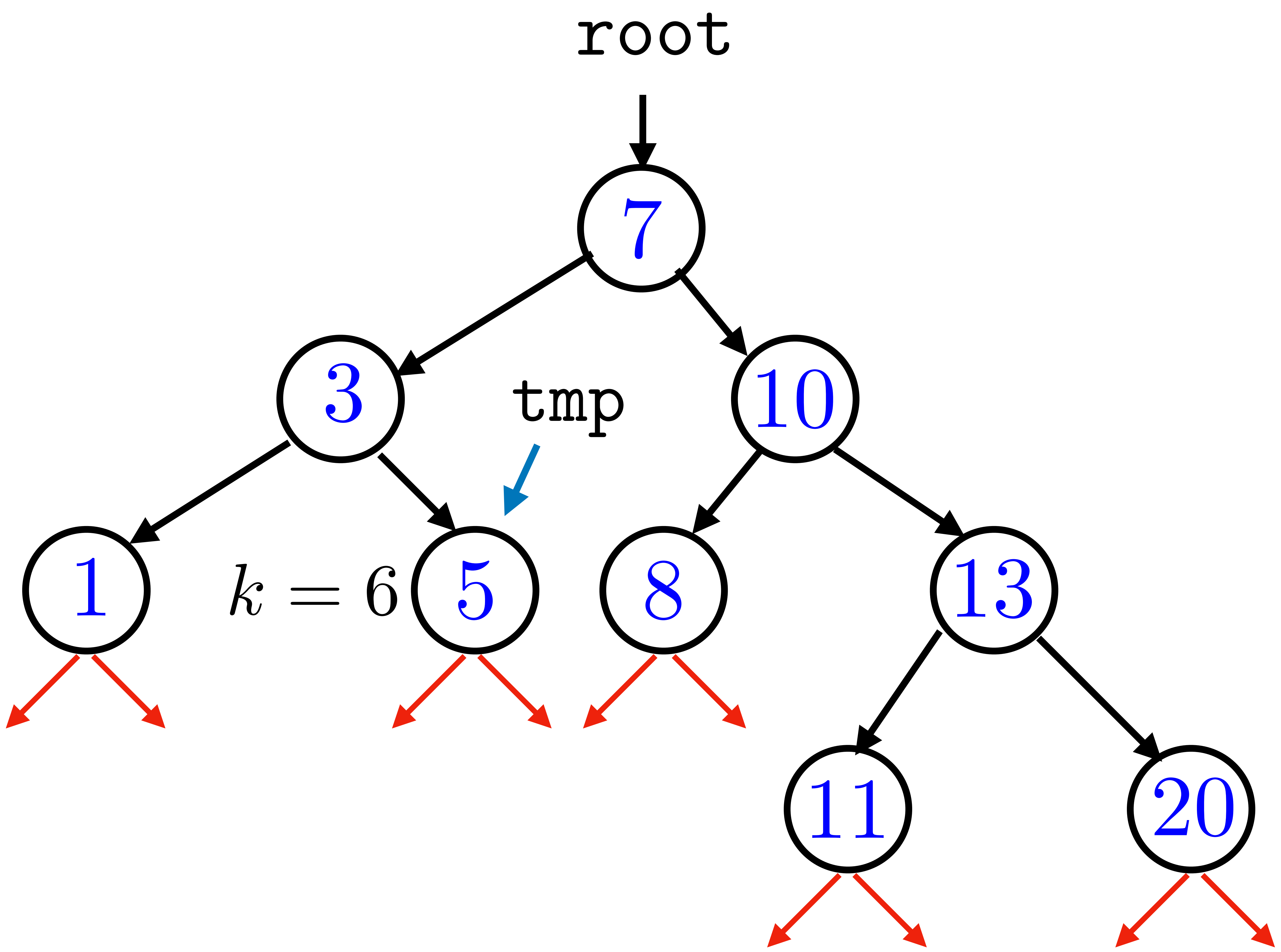
contains(6)

```
Node* contains(int k)
{
    Node* tmp = root;
    while(tmp != nullptr && tmp->key != k)
    {
        if(k < tmp->key)
        {
            tmp = tmp->left;
        }
        else
        {
            tmp = tmp->right;
        }
    }
    return tmp;
}
```



contains(6)

```
Node* contains(int k)
{
    Node* tmp = root;
    while(tmp != nullptr && tmp->key != k)
    {
        if(k < tmp->key)
        {
            tmp = tmp->left;
        }
        else
        {
            tmp = tmp->right;
        }
    }
    return tmp;
}
```

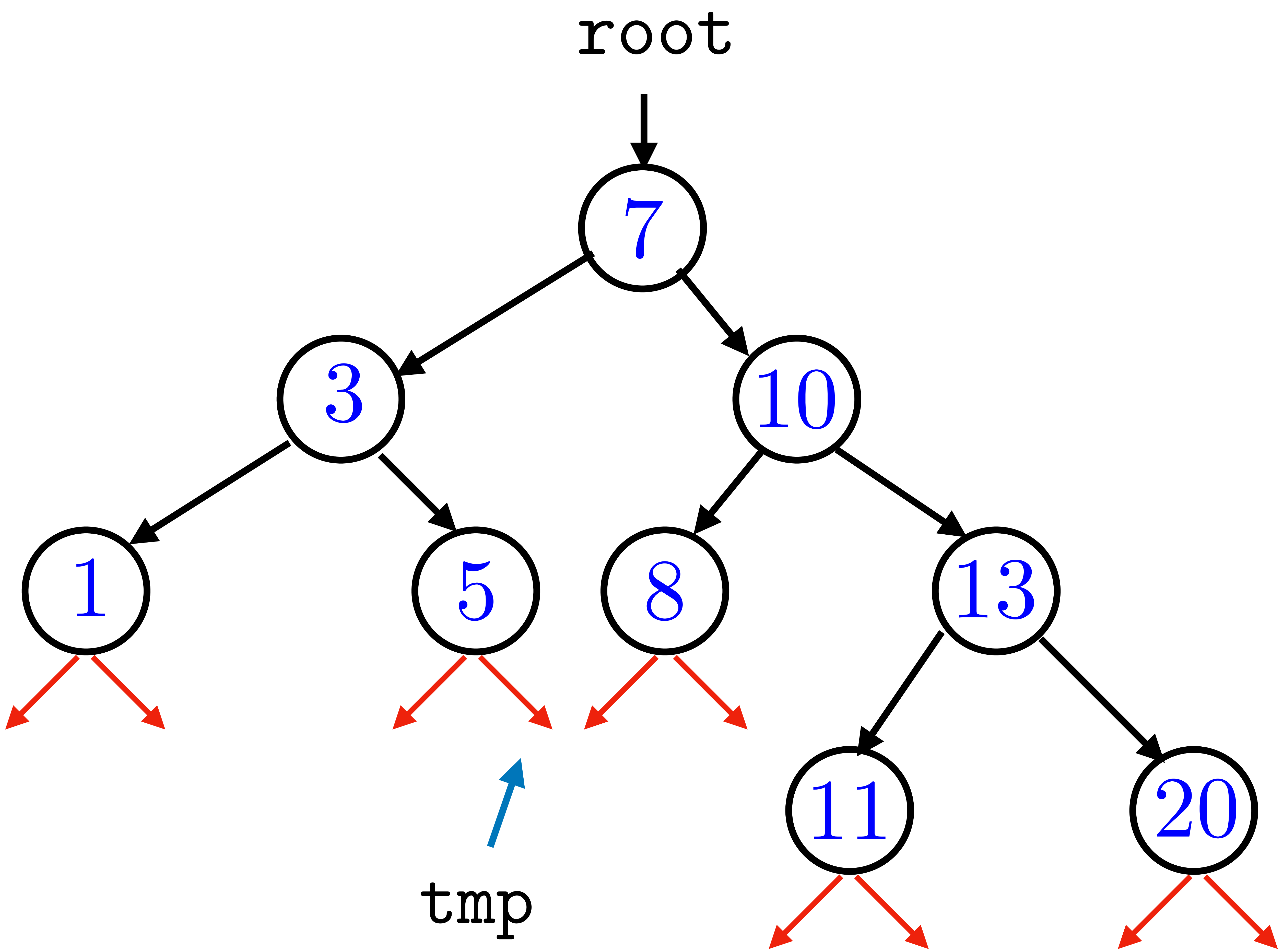


$\texttt{contains}(6)$

Now $\texttt{tmp} = \texttt{nullptr}$ and so the while loop terminates.

It returns `nullptr` .

```
Node* contains(int k)
{
    Node* tmp = root;
    while(tmp != nullptr && tmp->key != k)
    {
        if(k < tmp->key)
        {
            tmp = tmp->left;
        }
        else
        {
            tmp = tmp->right;
        }
    }
    return tmp;
}
```



Contains: complexity

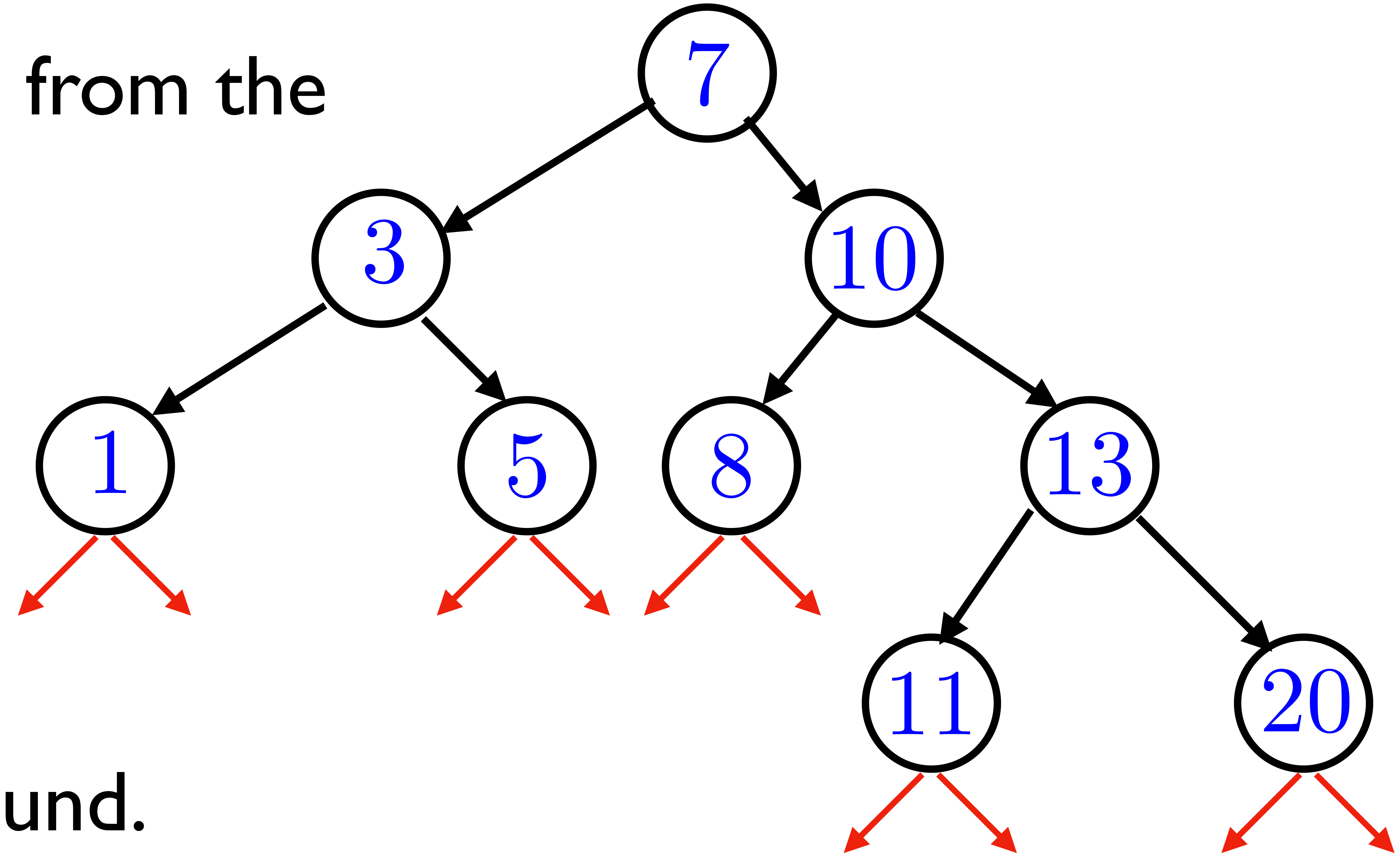
What is the complexity of `contains` ?

In the worst case, we visit every node from the root to the deepest leaf.

This takes time $\Omega(h)$ where h is the height of the tree.

The algorithm spends constant time at each node so $O(h)$ is an upper bound.

The complexity is $\Theta(h)$.



Insert

Insert is similar to a contains miss.

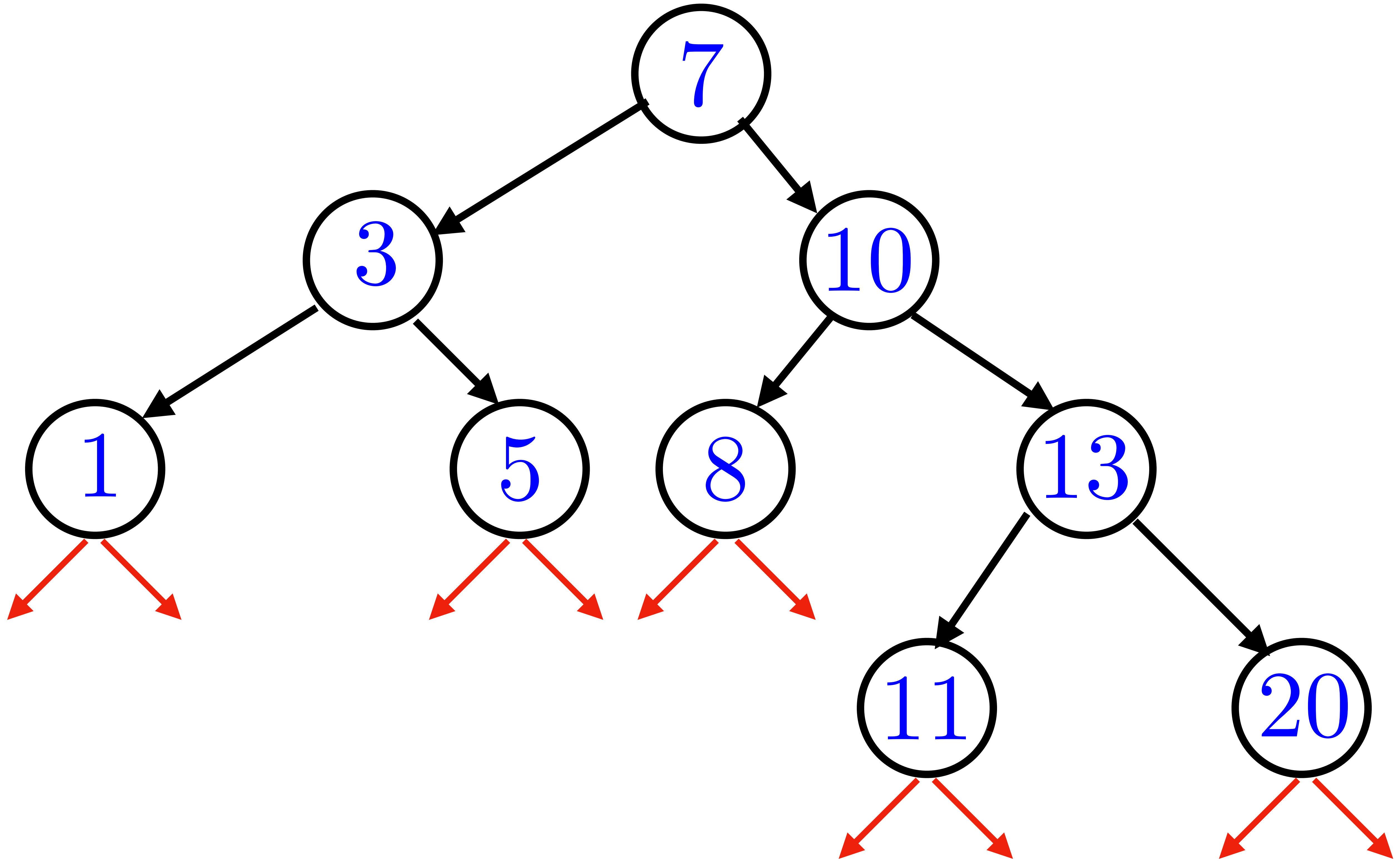
If a key is not in the set, `contains` ends up at a `nullptr` coming out of a leaf.

We instead make this `nullptr` point to a new node with the key to be inserted.

In particular, inserted nodes are always leaves.

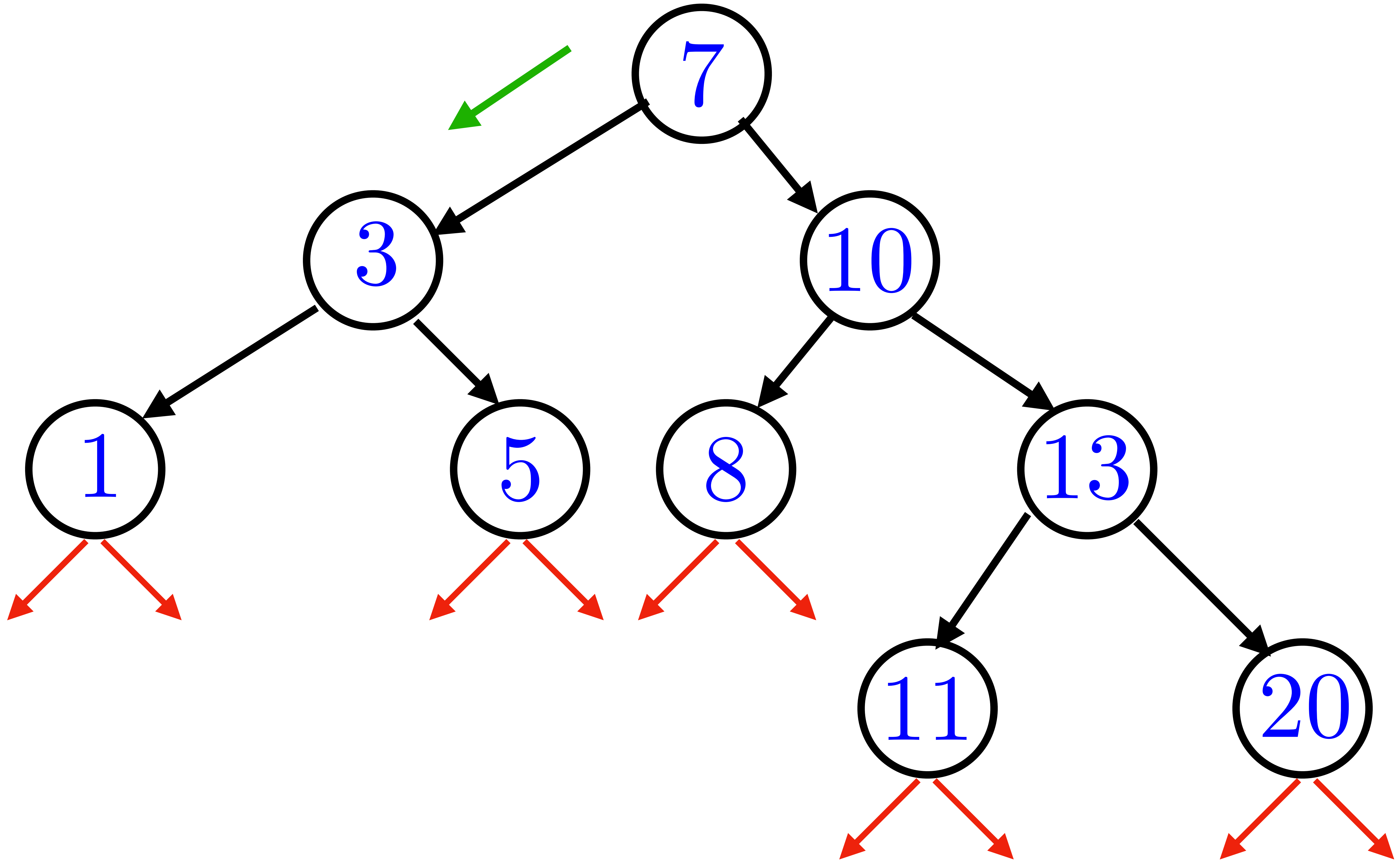
Insert: example

insert(2)



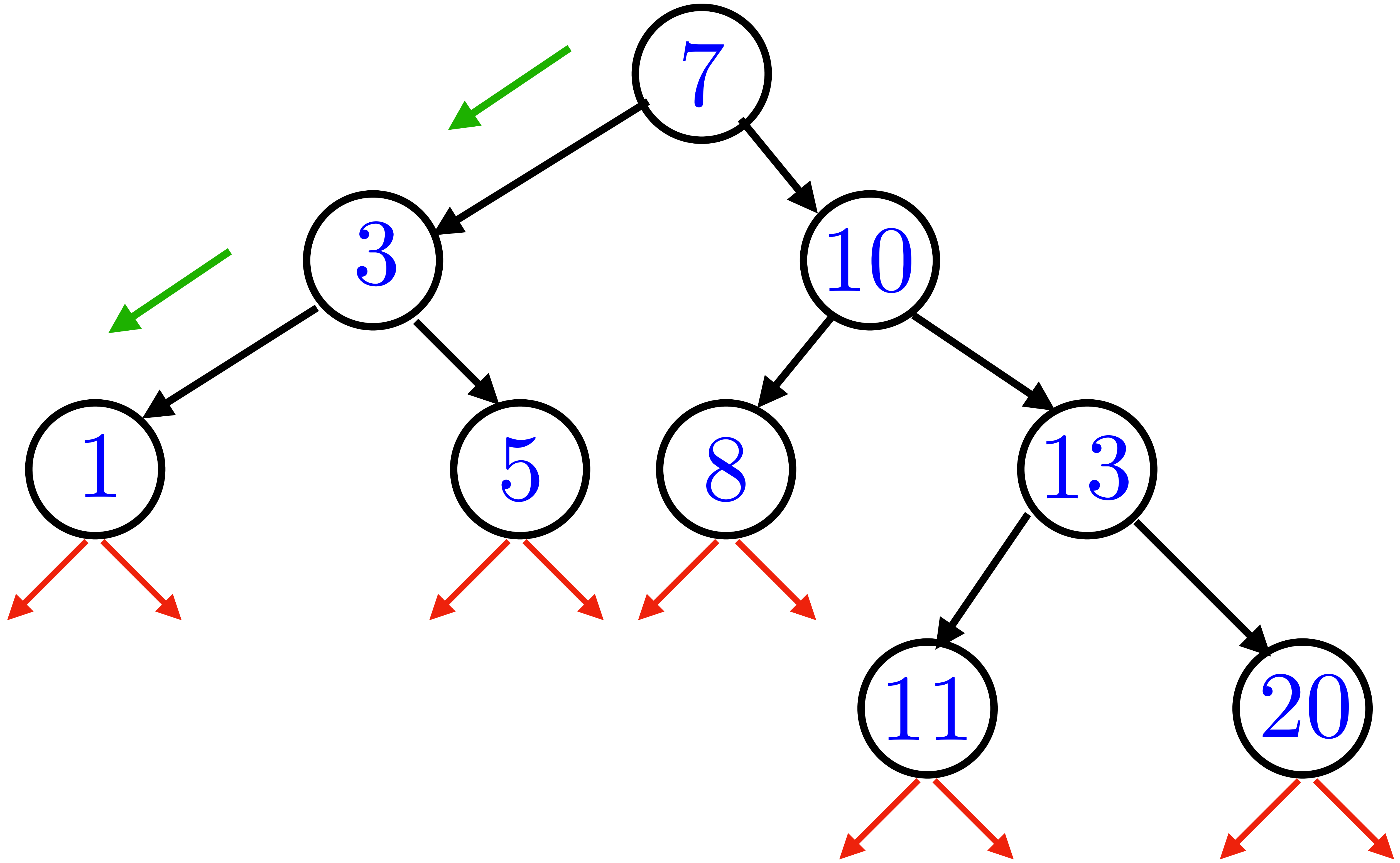
Insert: example

$\texttt{insert}(2)$



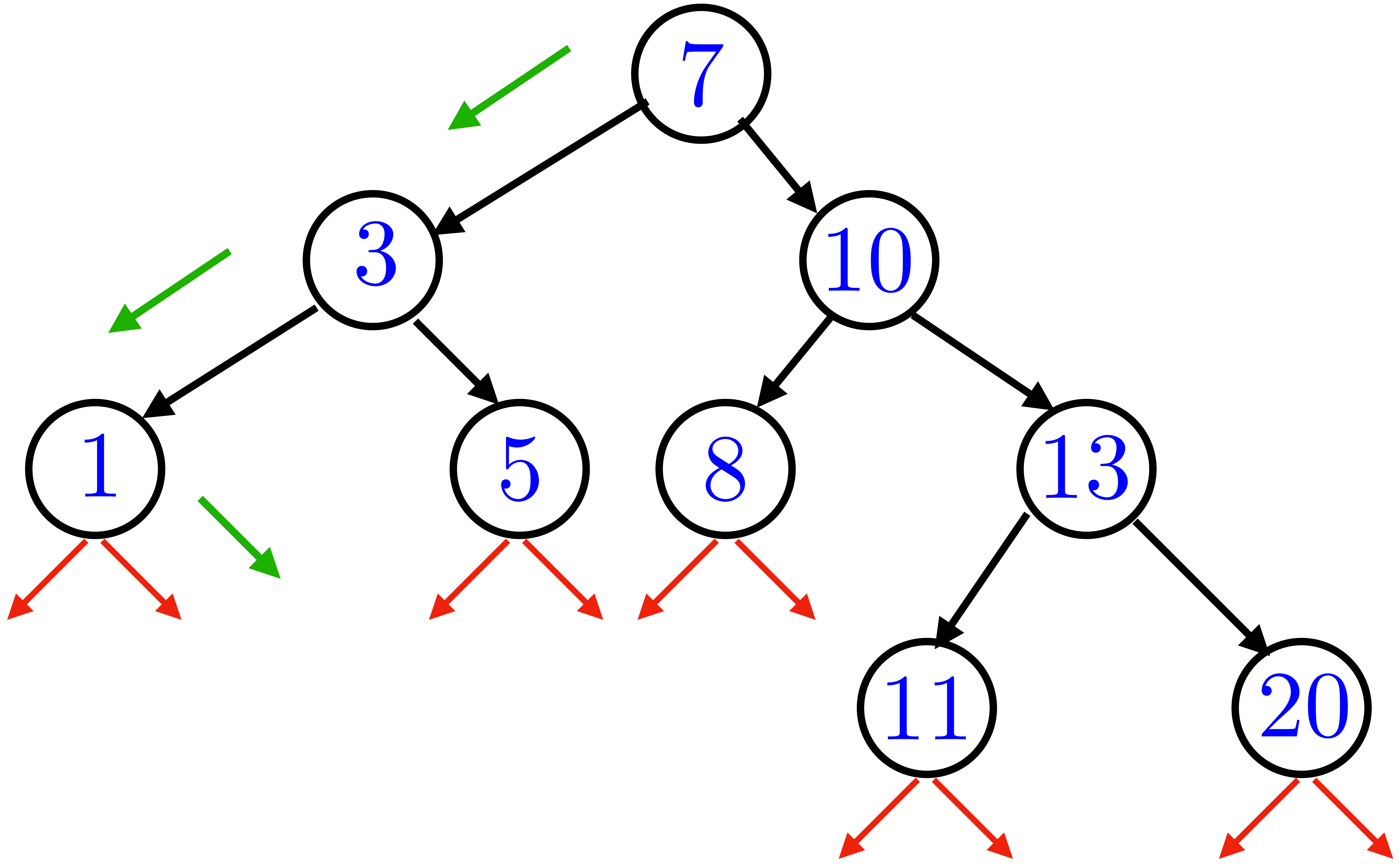
Insert: example

insert(2)



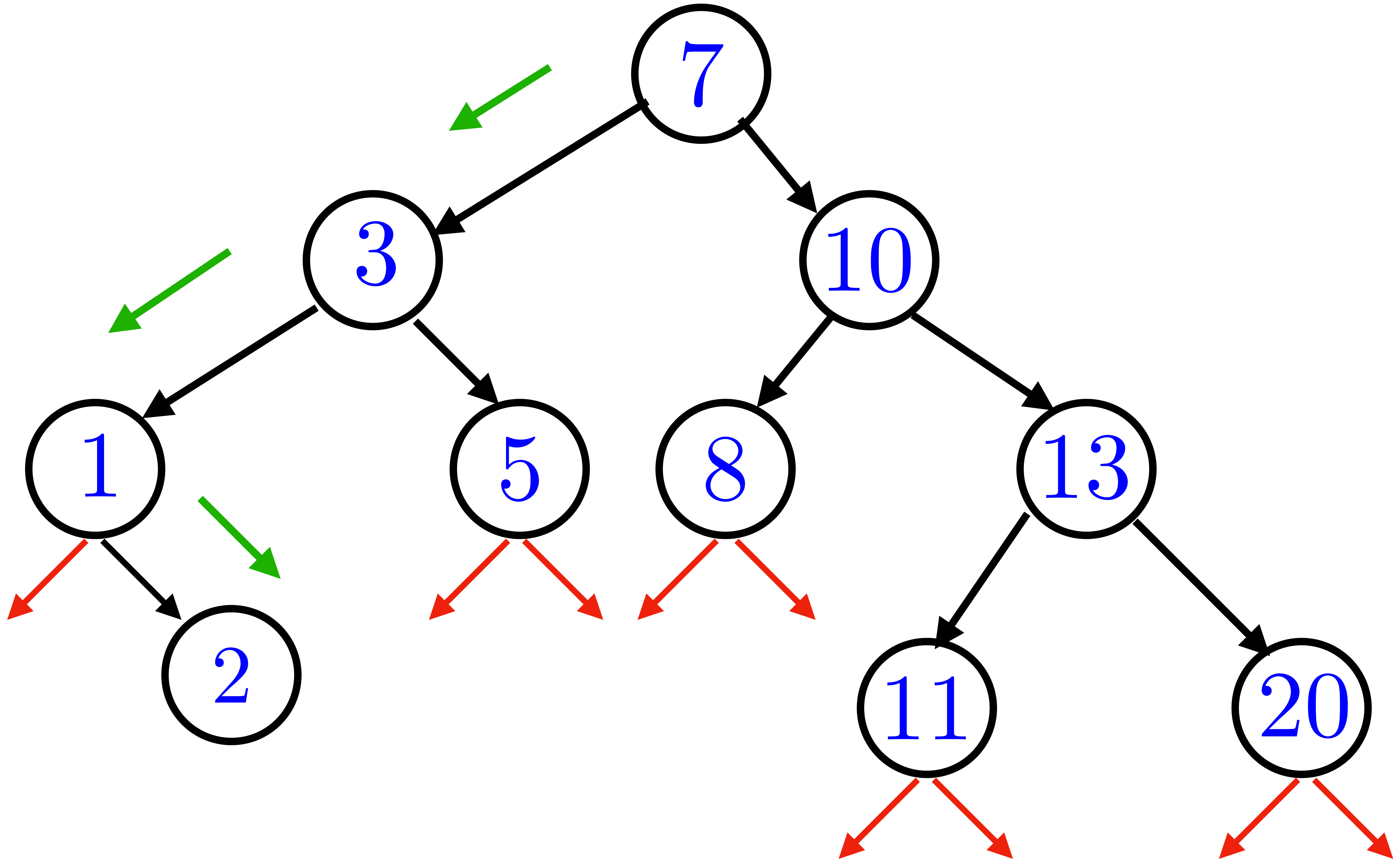
Insert: example

$\texttt{insert}(2)$



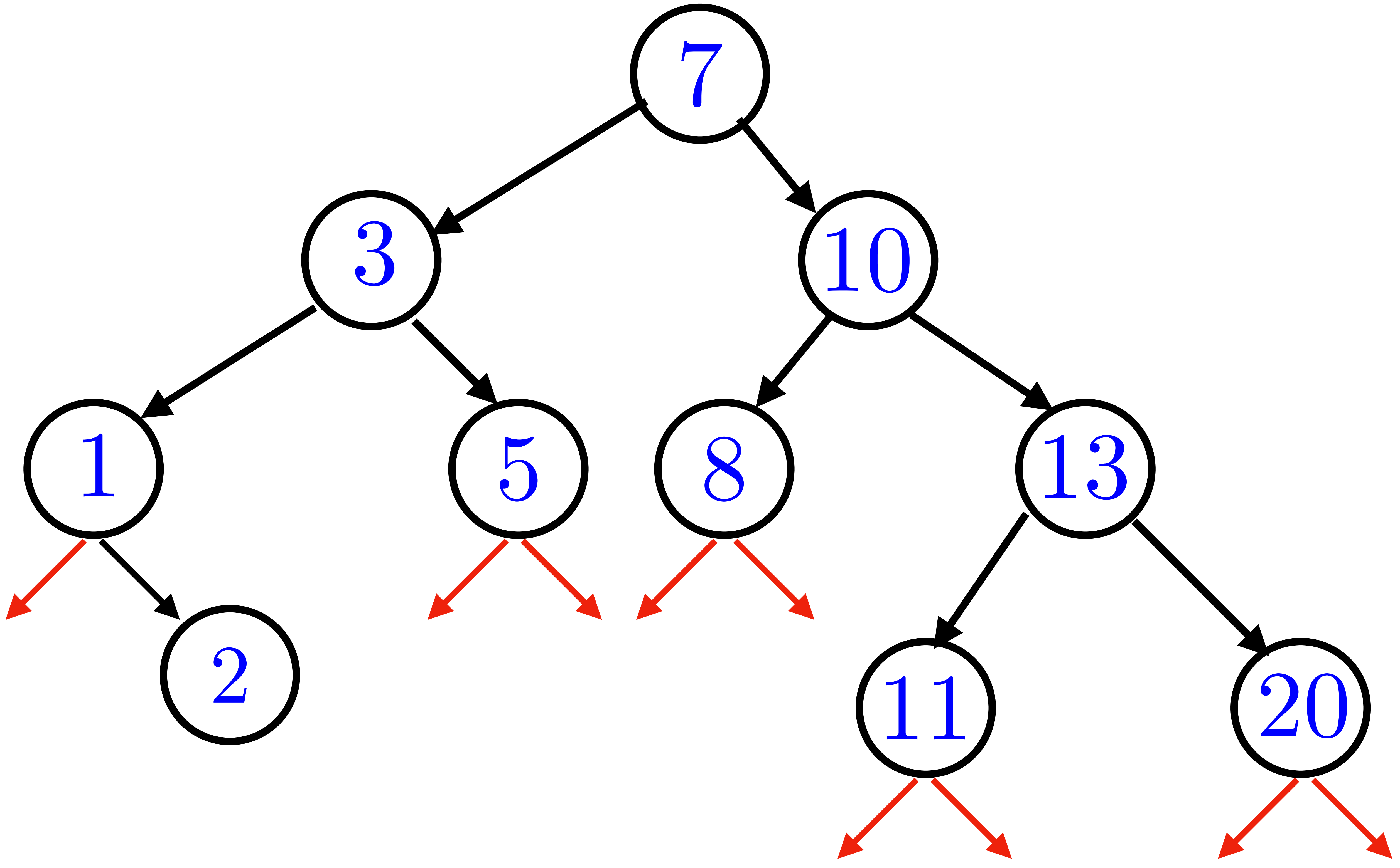
Insert: example

insert(2)



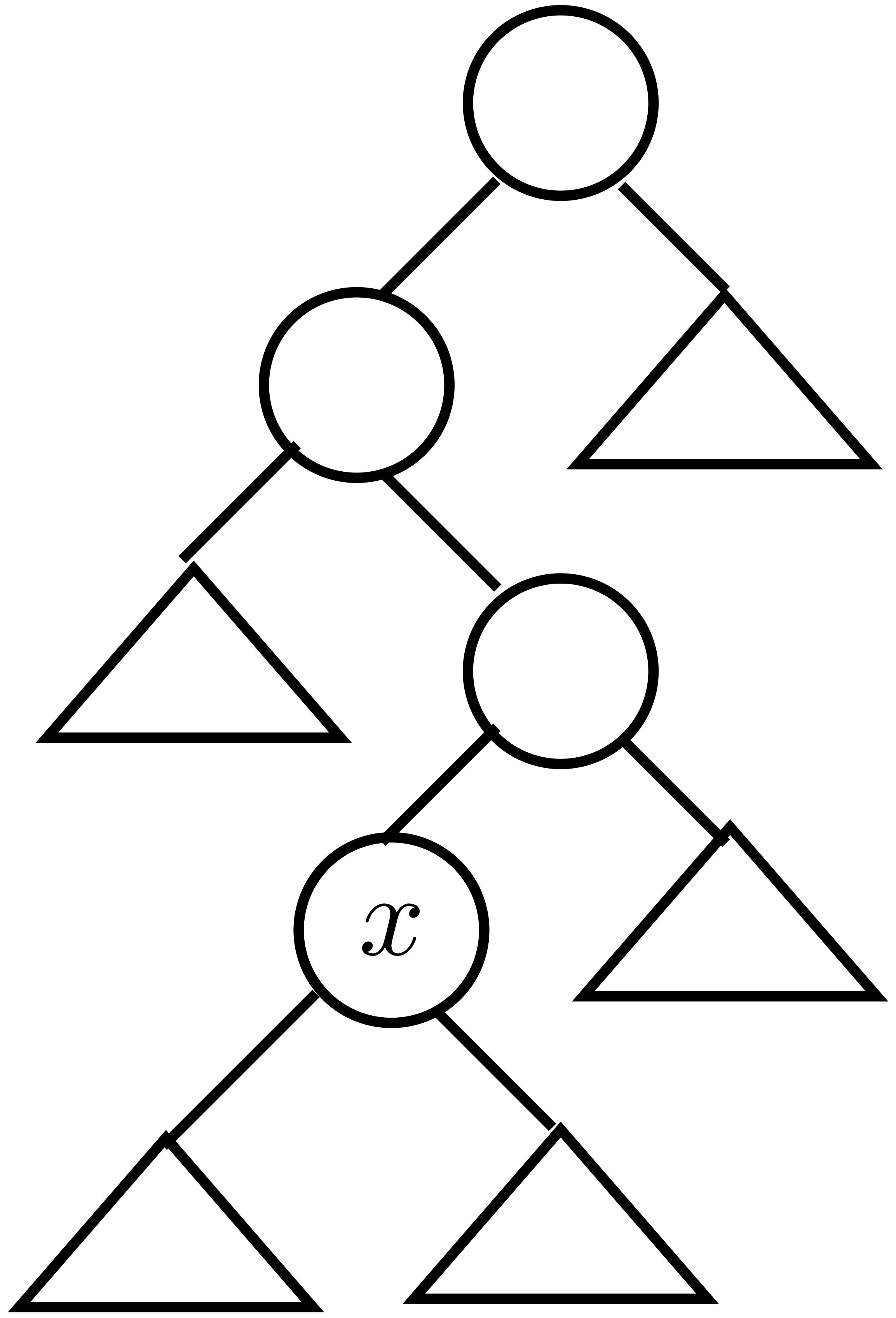
Insert: complexity

The worst case time for insertion is also $\Theta(h)$.



Successor

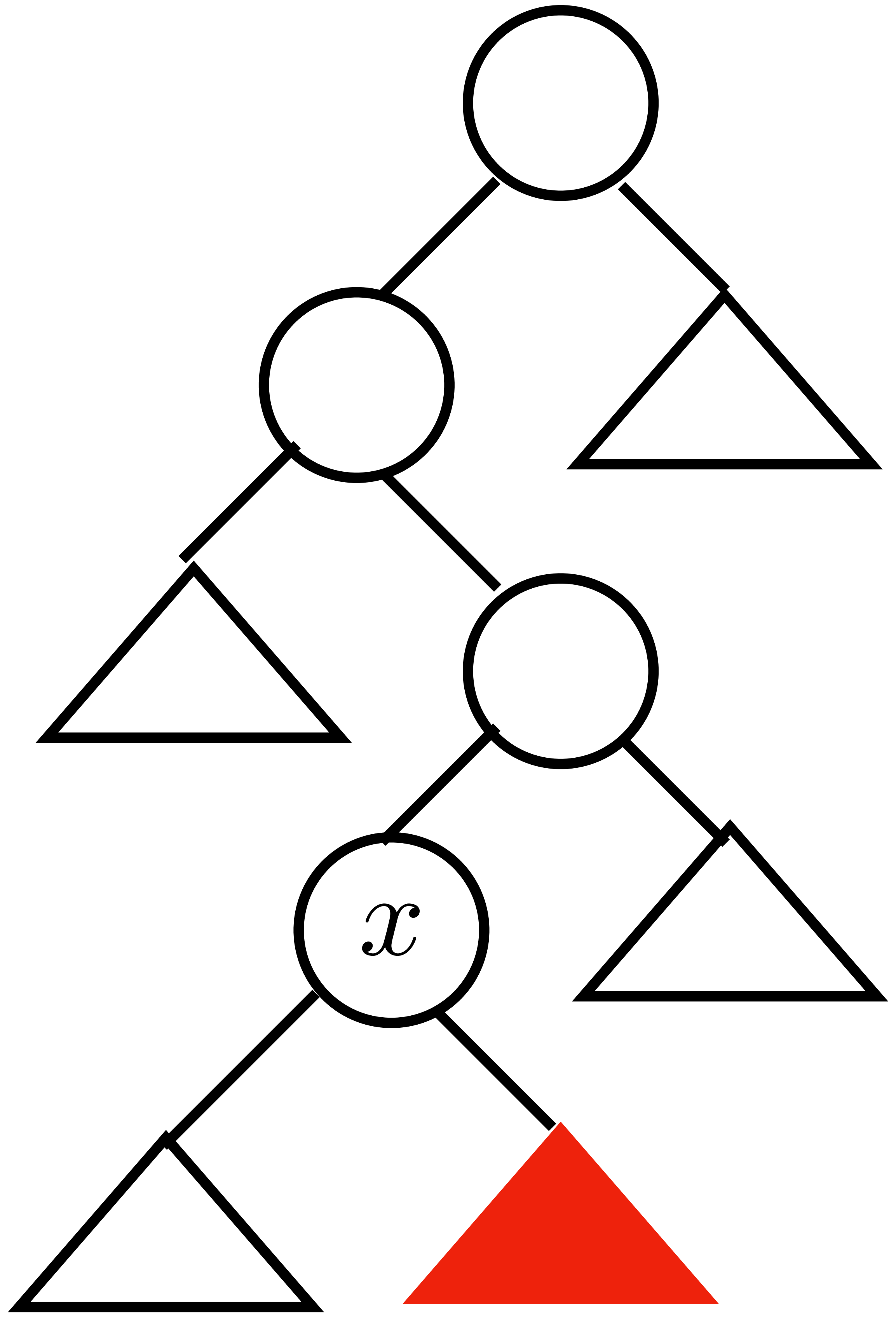
Before we get into successor let's think about the BST property some more.



Where are keys larger than x ?

Successor

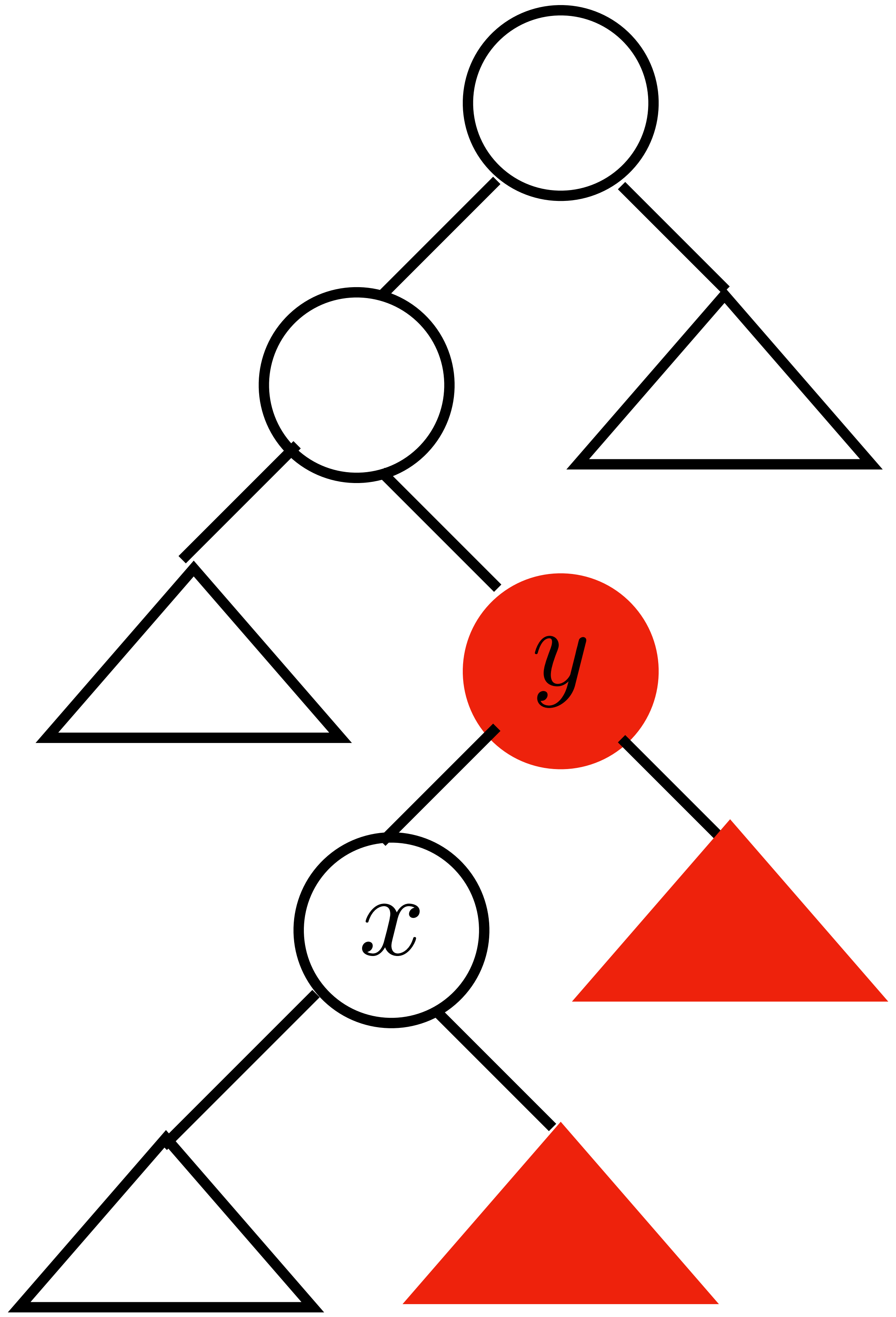
Before we get into successor let's think about the BST property some more.



Where are keys larger than x ?

Successor

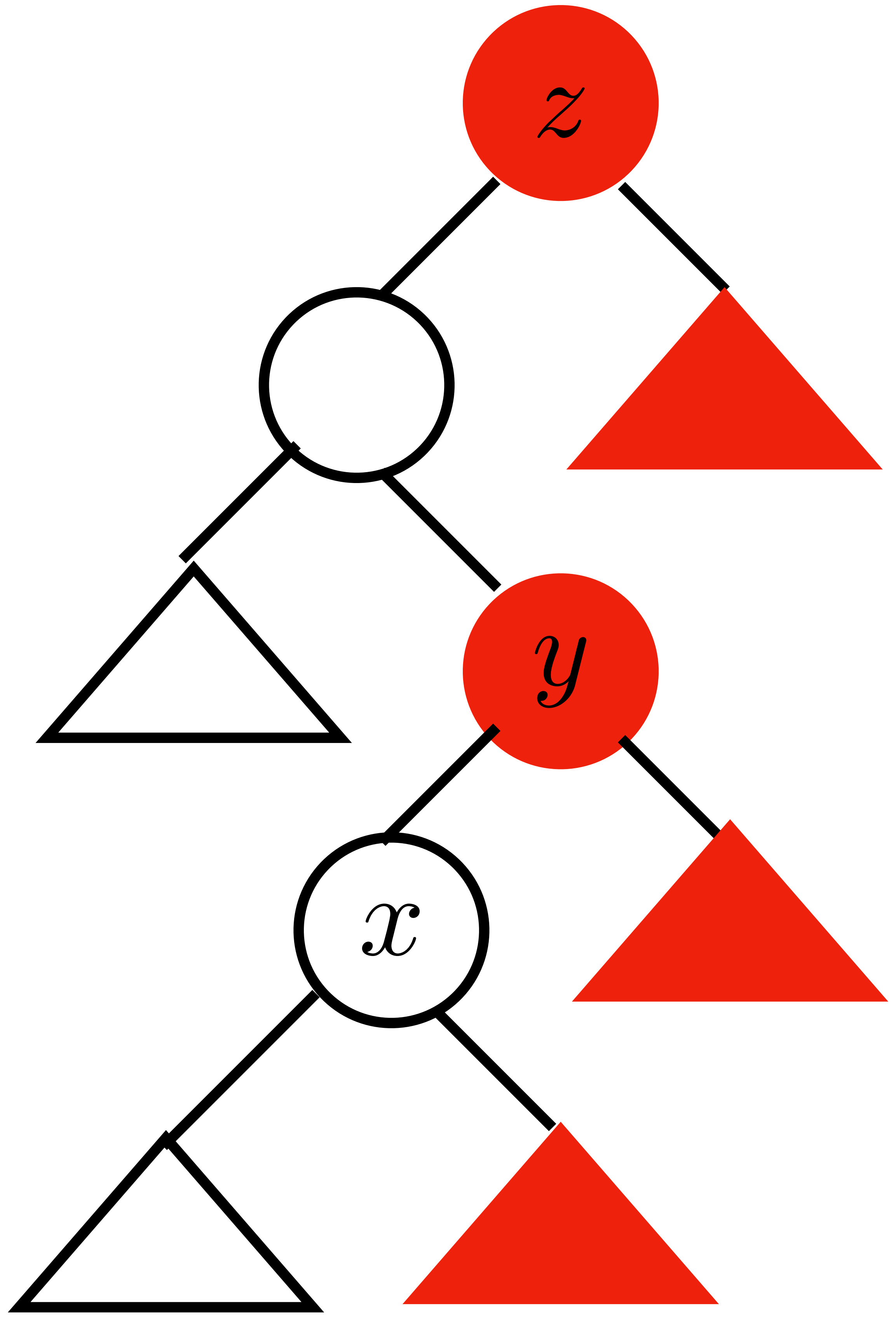
Before we get into successor let's think about the BST property some more.



Where are keys larger than x ?

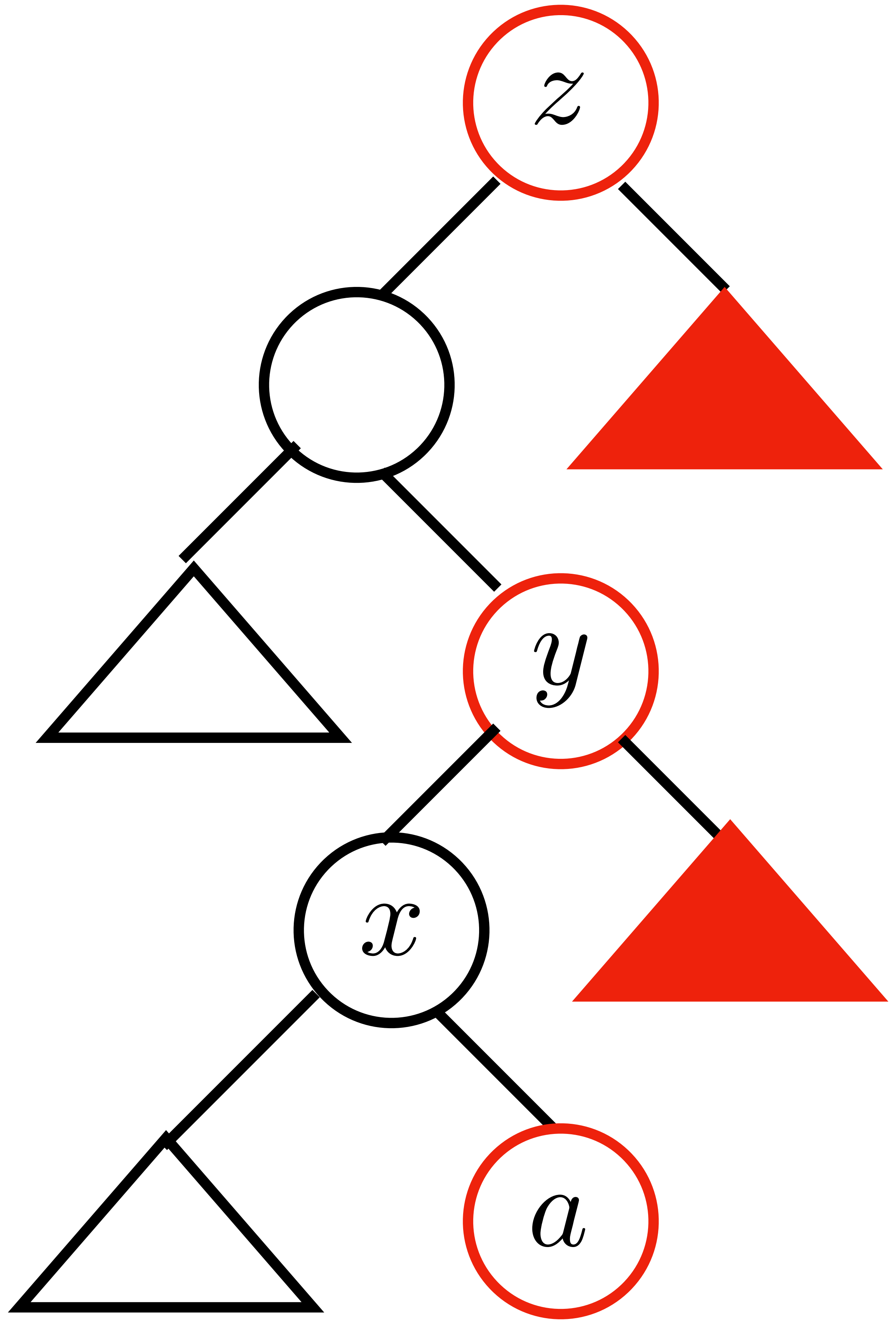
Successor

Before we get into successor let's think about the BST property some more.



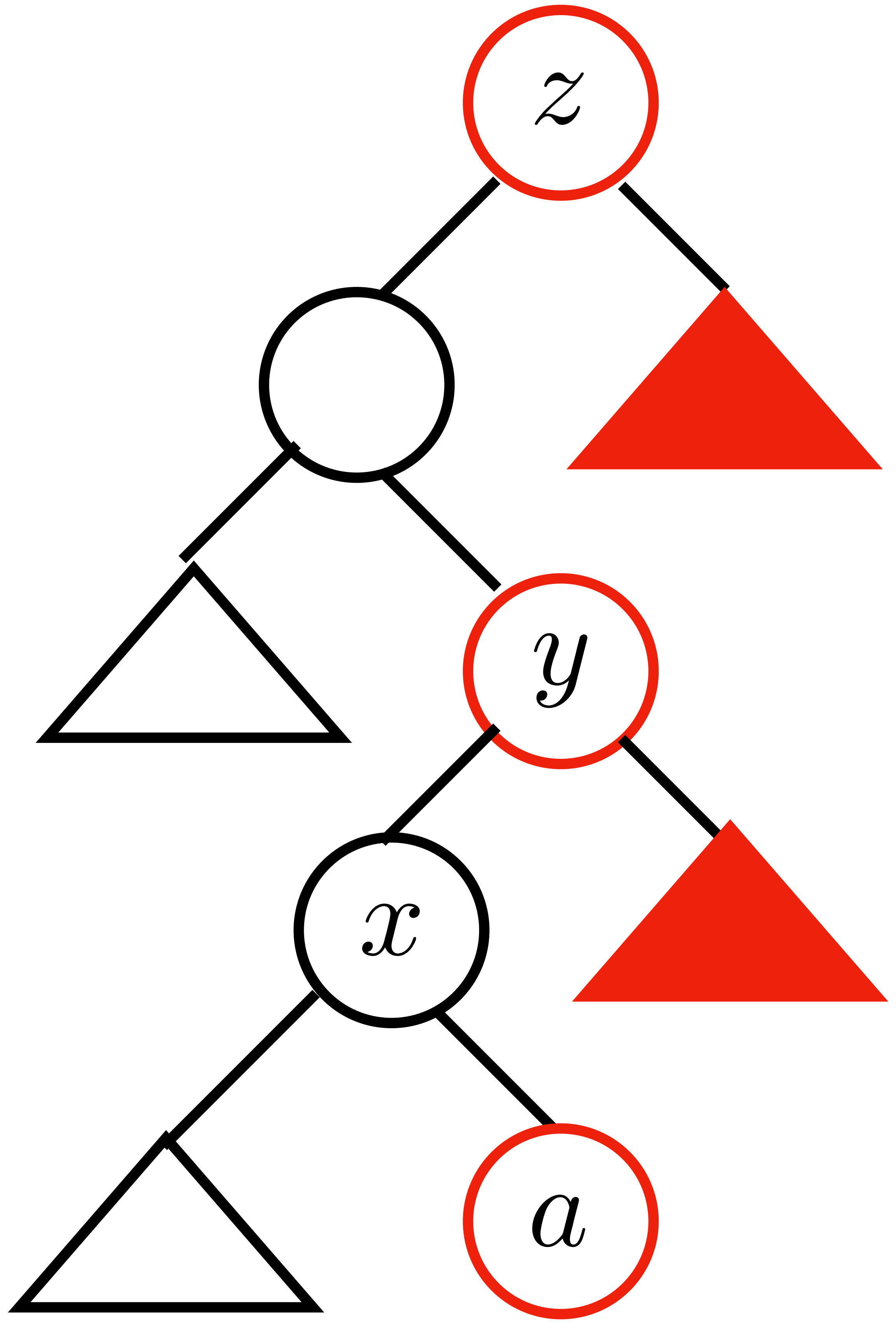
Where are keys larger than x ?

Now that we understand where the keys larger than x are, we move to find the successor of x.



What is the ordering of z, y, a ?

Now that we understand where the keys larger than x are, we move to find the successor of x .

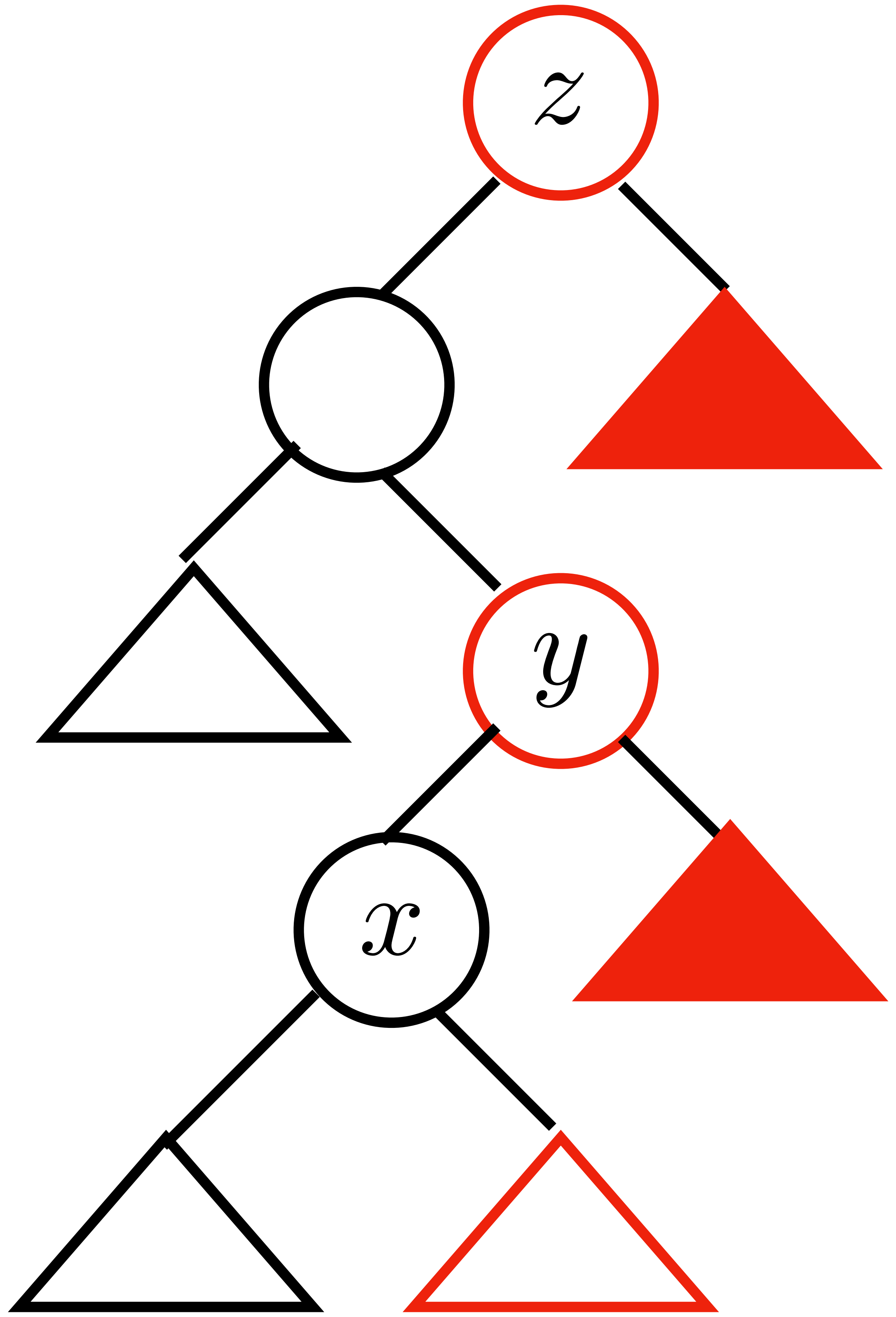


What is the ordering of z, y, a ?

We know that $a < y < z$.

To find the successor of x we should look in its right subtree first.

Successor: Case 1

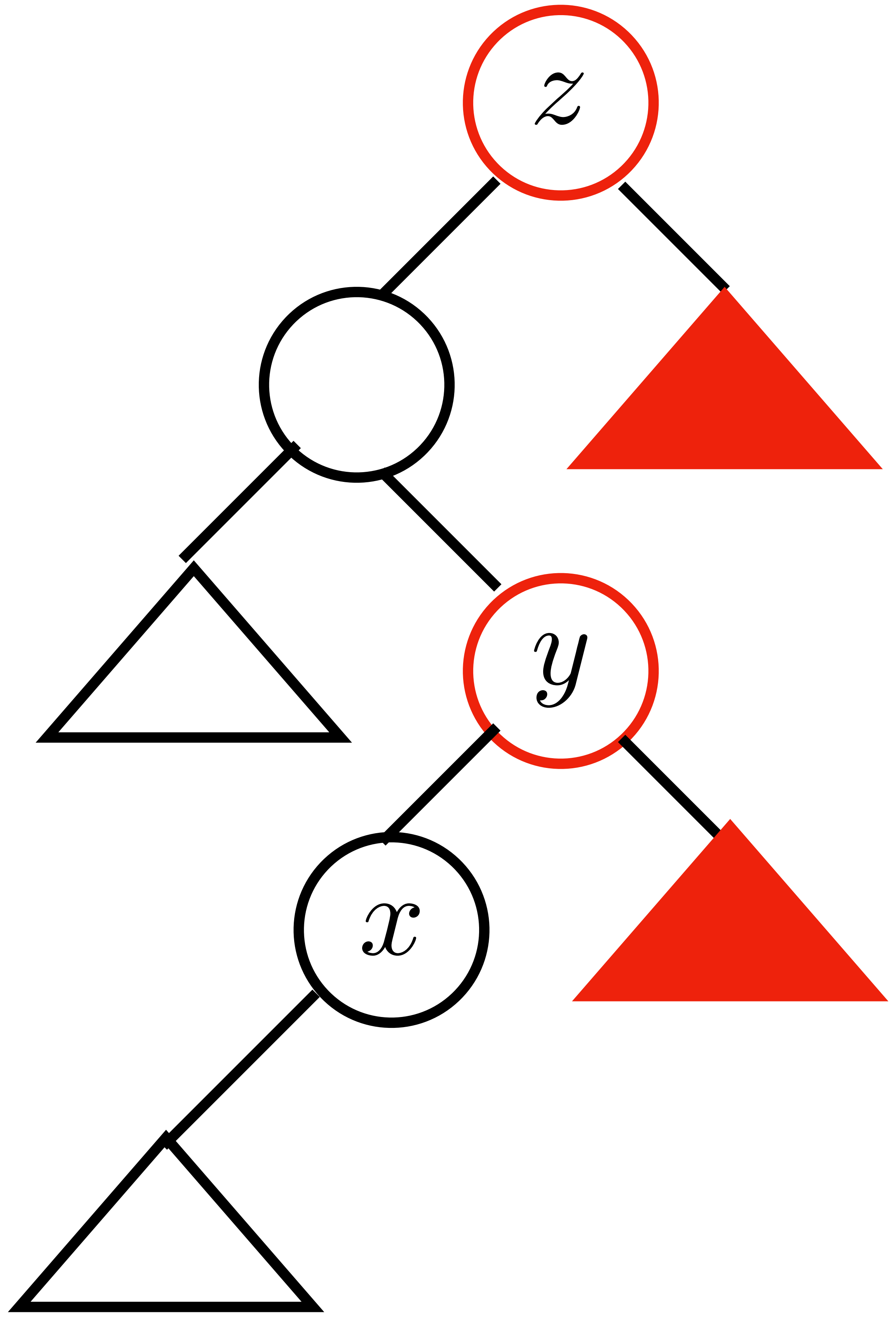


To find the successor of x we should look in its right subtree first.

Fact: If x has a right child, the successor of x is the minimum element of the right subtree of x .

The successor is found by always going left in the right subtree of x .

Case 2: no right child



What if x has no right child?

Fact: If x has no right child, the successor of x is the node where you last go left on the path from the root to x .

To implement successor we can remember the last left turn as we search for x in the tree.

Successor: complexity

To find the successor of x, we first find x in the tree.

The complexity is at least $\Omega(h)$, that of `contains`.

What work do we do beyond that of `contains`?

In the case x has a right child, we find the minimum in the right subtree.
This can also be done in $O(h)$ time.

`successor` also has complexity $\Theta(h)$.

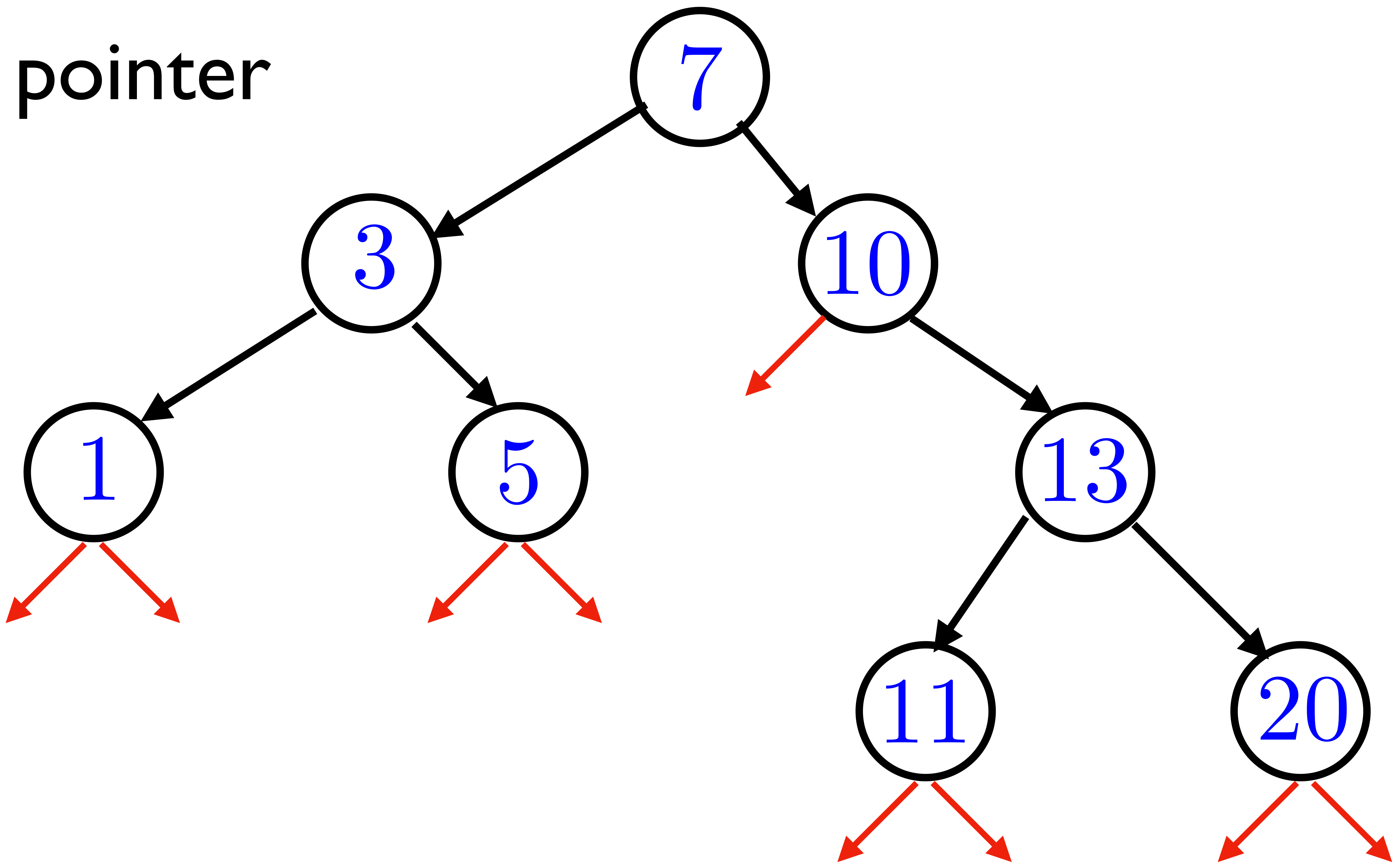
Remove

Let's now see how to remove a node. This is trickier than the others.

First some easy cases. Say the node to delete is a leaf.

We delete the leaf, and change the pointer from its parent to a `nullptr` .

Example: remove key 5 .

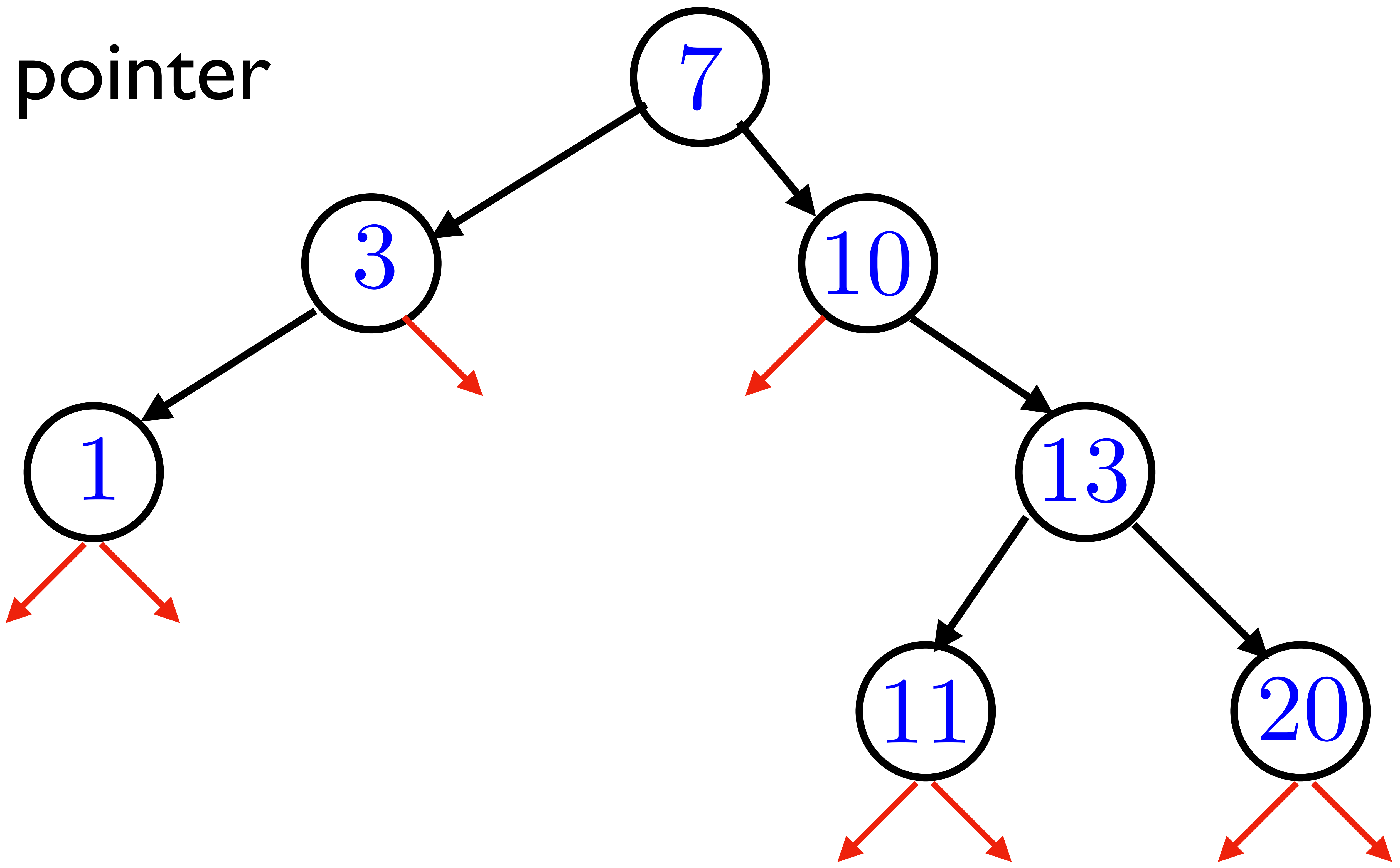


Remove: leaf

Let's now see how to remove a node. This is trickier than the others.

First some easy cases. Say the node to delete is a leaf.

We delete the leaf, and change the pointer from its parent to a `nullptr` .



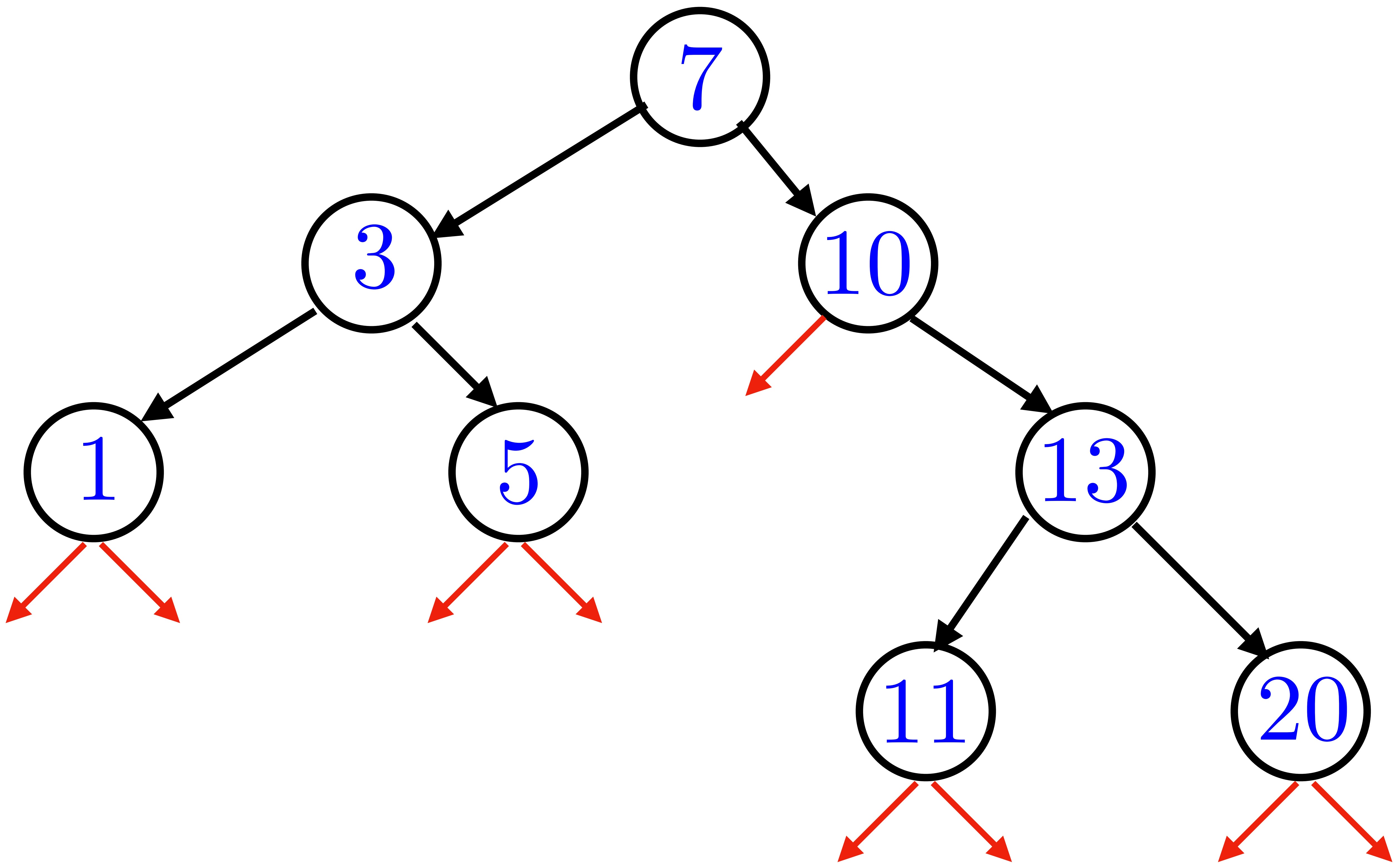
Example: delete key 5 .

Remove: one child

Similar easy case: Node to delete just has one child.

Then the child takes the place of the node to delete.

Example: remove key 10 .

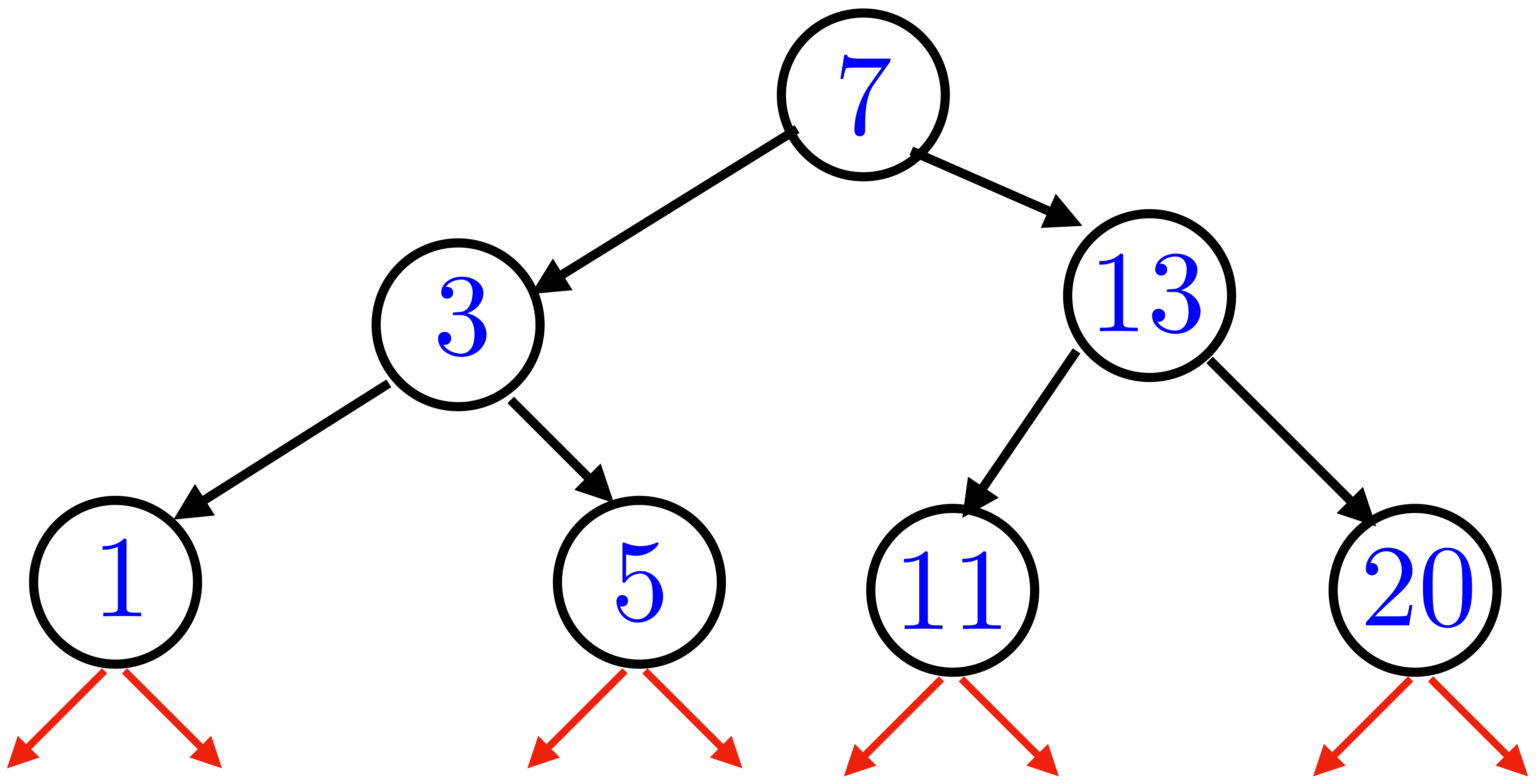


Remove: one child

Similar easy case: Node to delete just has one child.

Then the child takes the place of the node to delete.

Example: delete key 10.



Note this preserves the BST property.

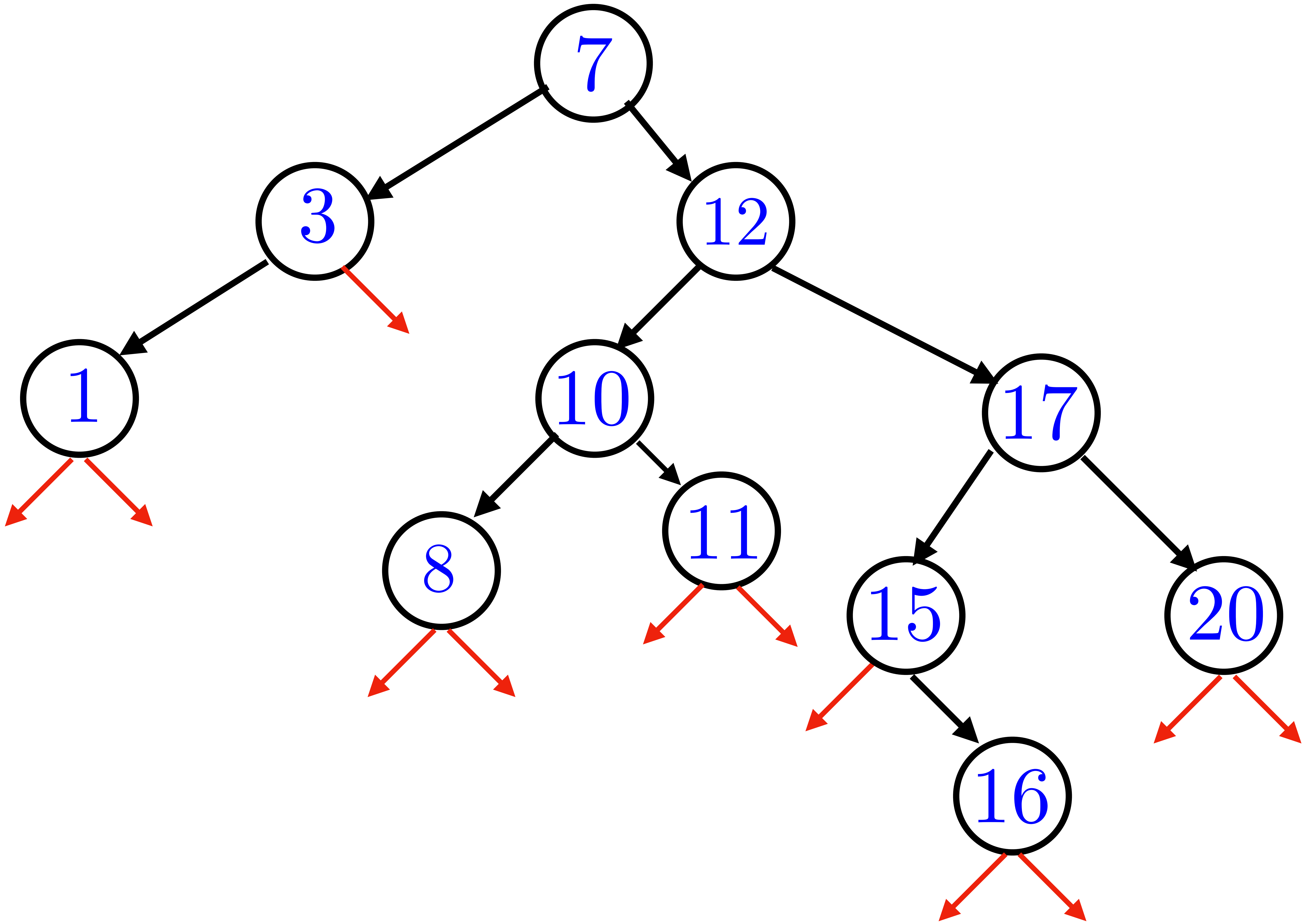
Remove: both children

The interesting case is where the node to delete has both children.

Let's say we want to delete key 12.

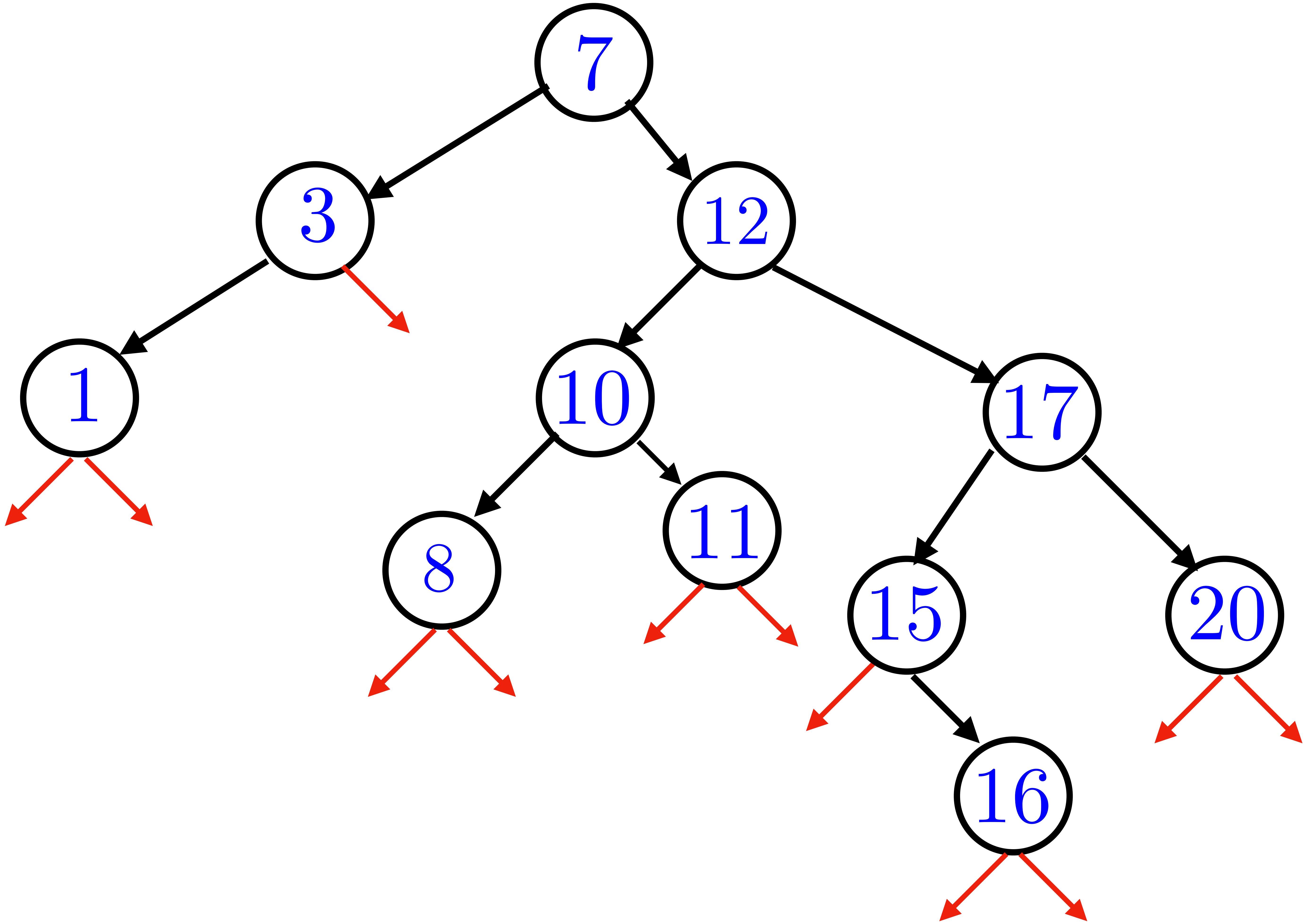
We want to replace 12 by some other key in the set.

What key can we replace it with that causes minimal change?



Remove: both children

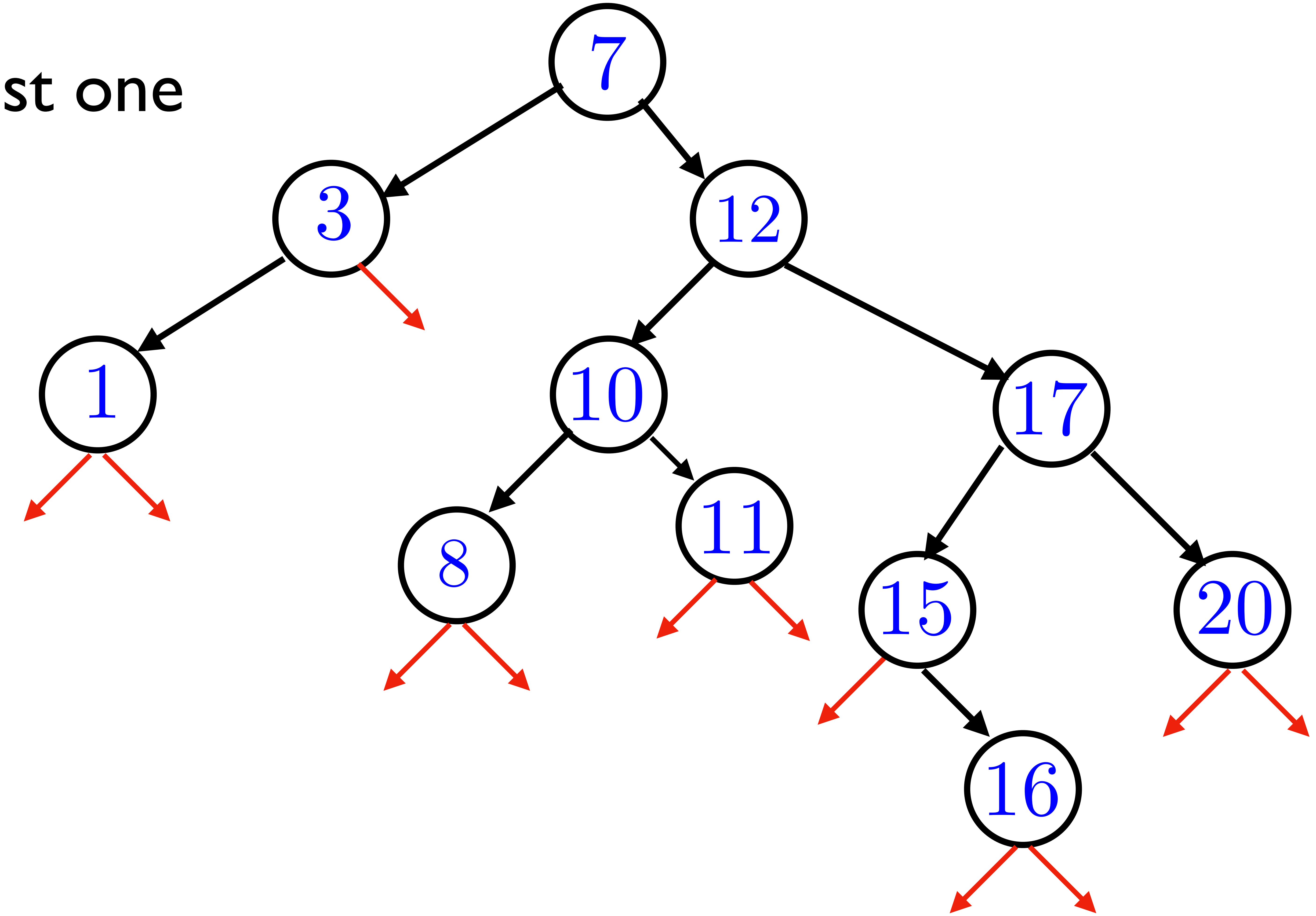
What key can replace 12 and cause minimal change?



Remove: both children

What key can replace 12 and cause minimal change?

Idea: Replace by a node with at most one child.

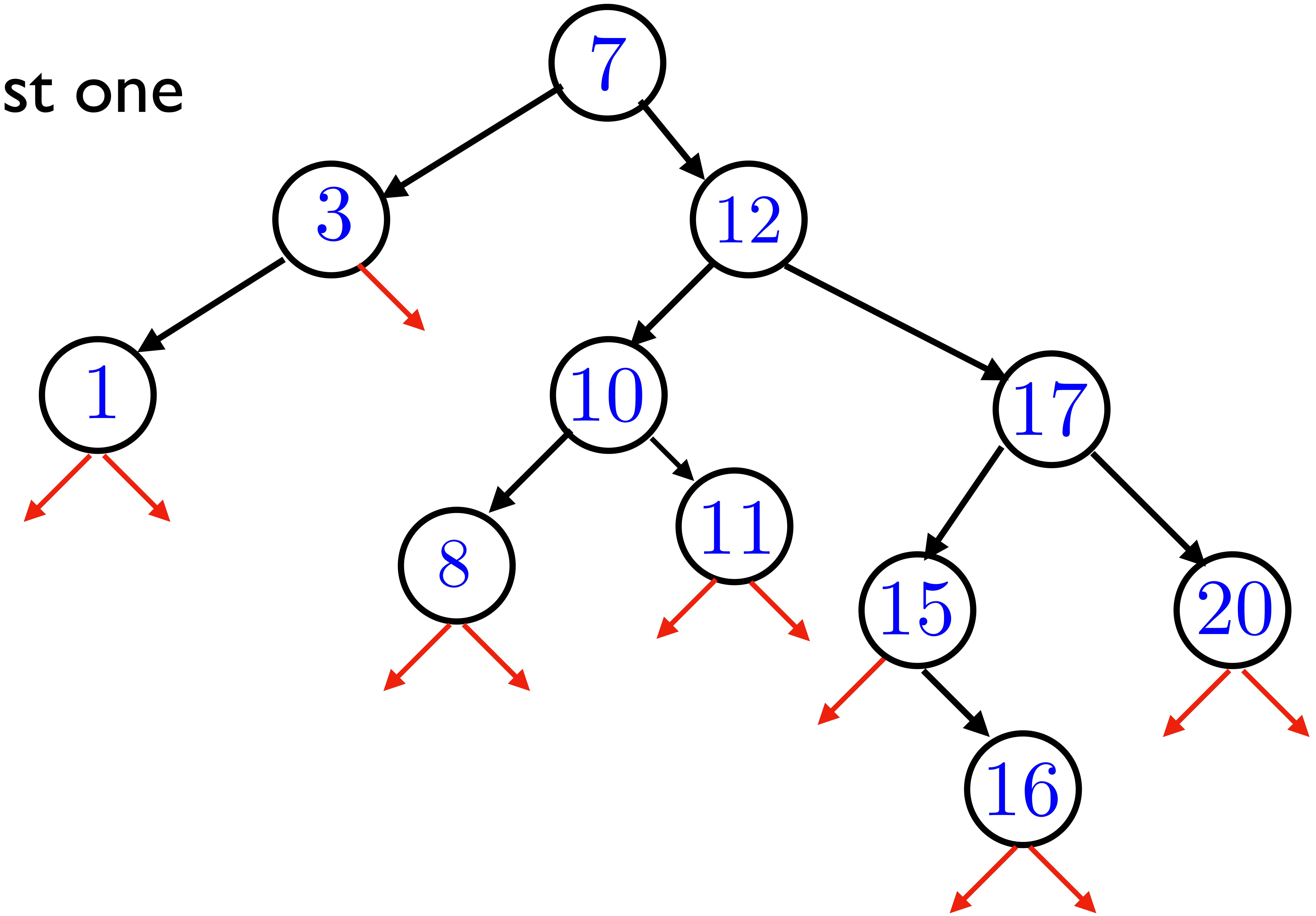


Remove: both children

What key can replace 12 and cause minimal change?

Idea: Replace by a node with at most one child.

What is a node with at most one child whose key fits in the position of 12?



Remove: both children

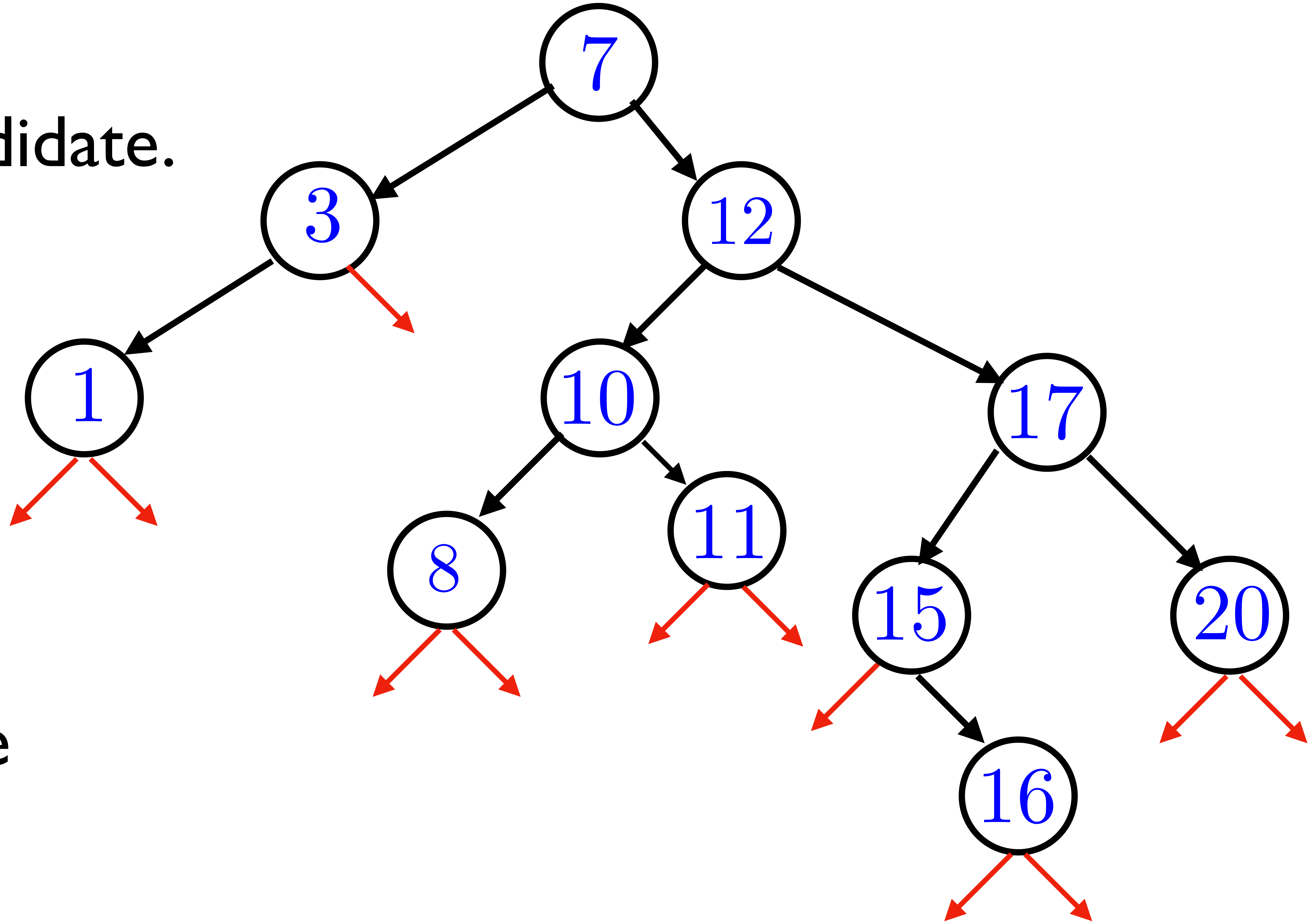
What is a node with at most one child whose key fits in the position of 12?

The successor of 12 is a good candidate.

It has no left child.

It is bigger than everything in the left subtree of 12.

It is less than everything else in the right subtree of 12.



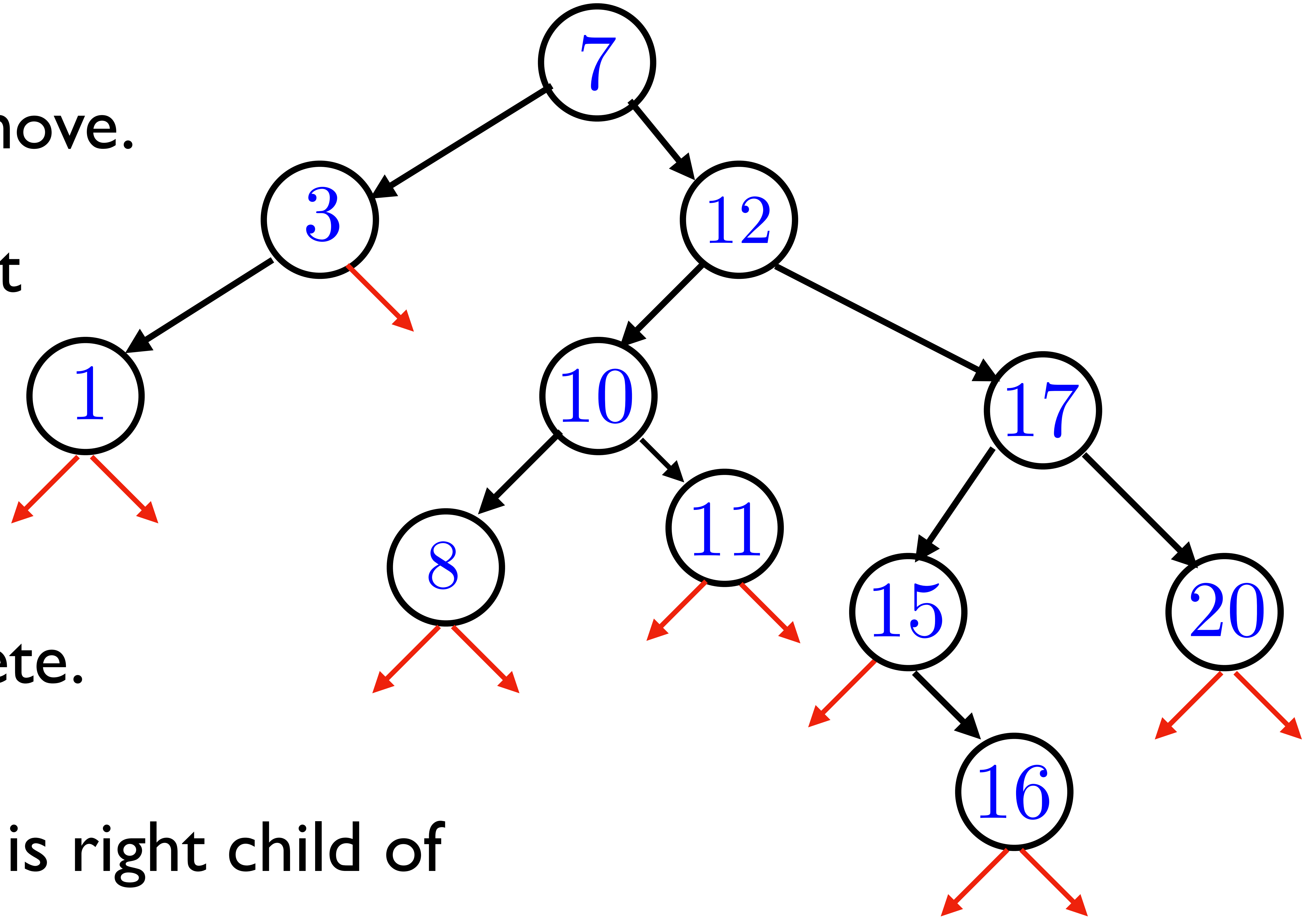
Remove: both children

Algorithm idea:

1) Find successor s of key to remove.

2) Make parent of s point to right child of s .

3) Make parent of node to delete point to s , and update children of s with children of node to delete.



Careful with special case: successor is right child of node to delete.

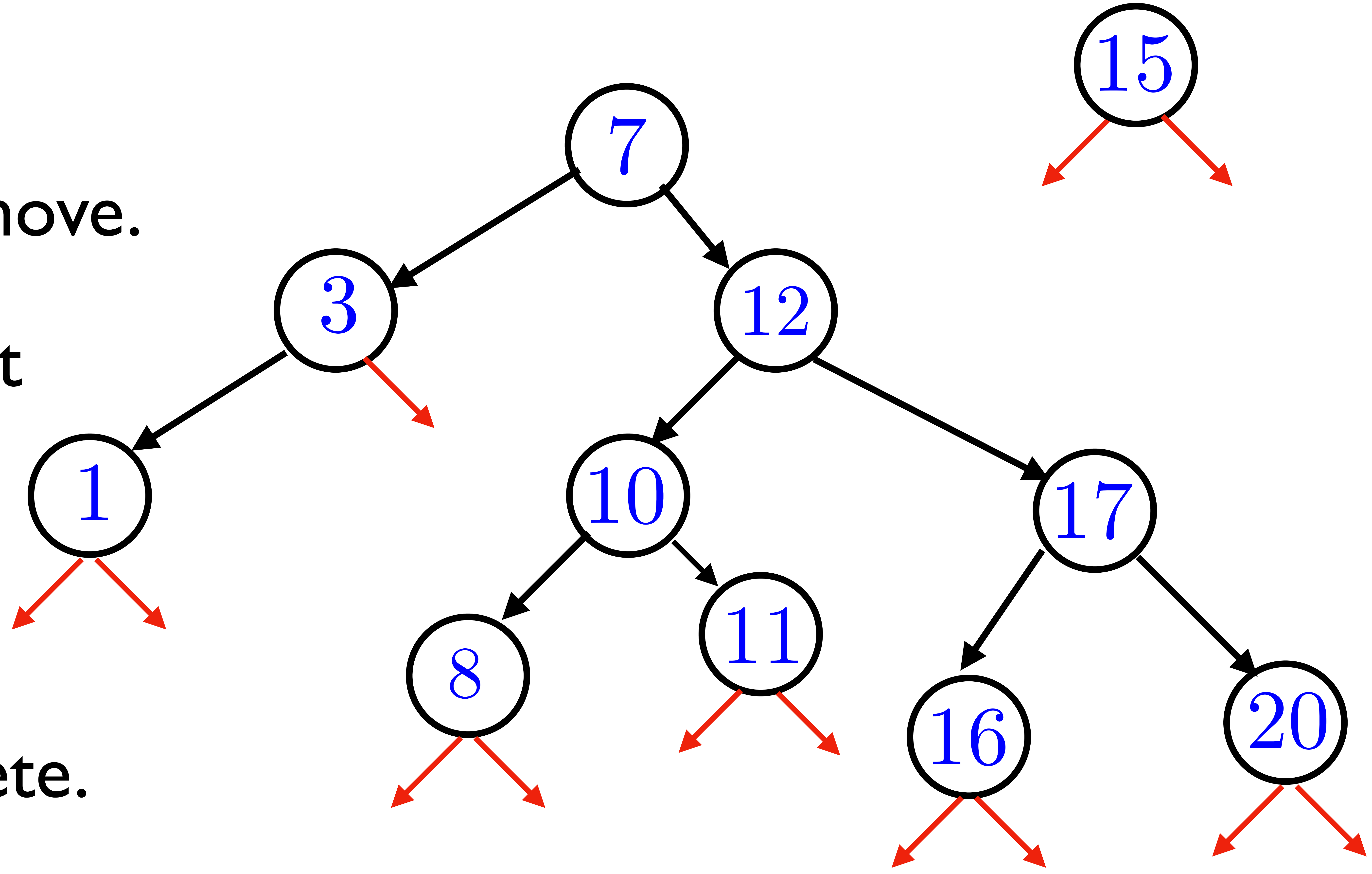
Remove: both children

Algorithm idea:

1) Find successor s of key to remove.

2) Make parent of s point to right child of s .

3) Make parent of node to delete point to s , and update children of s with children of node to delete.



Careful with special case: successor is right child of node to delete.

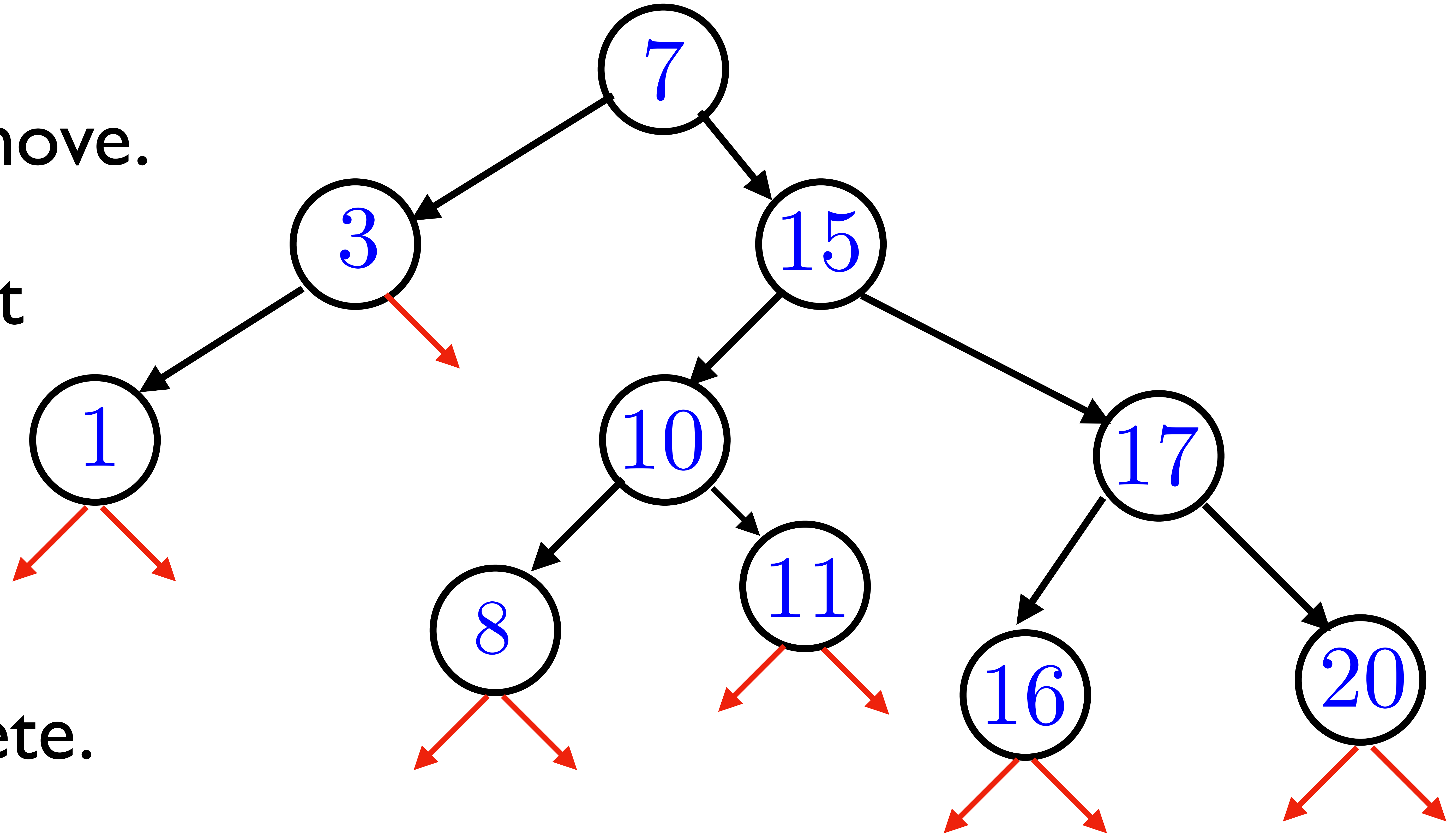
Remove: both children

Algorithm idea:

1) Find successor s of key to remove.

2) Make parent of s point to right child of s .

3) Make parent of node to delete point to s , and update children of s with children of node to delete.



Careful with special case: successor is right child of node to delete.

Complexity of remove

Complexity of remove is essentially:

1) Finding node to delete (`contains`).

2) Successor operation.

3) Constant number of pointer changes.

Complexity is again $\Theta(h)$, where h is the height of the tree.

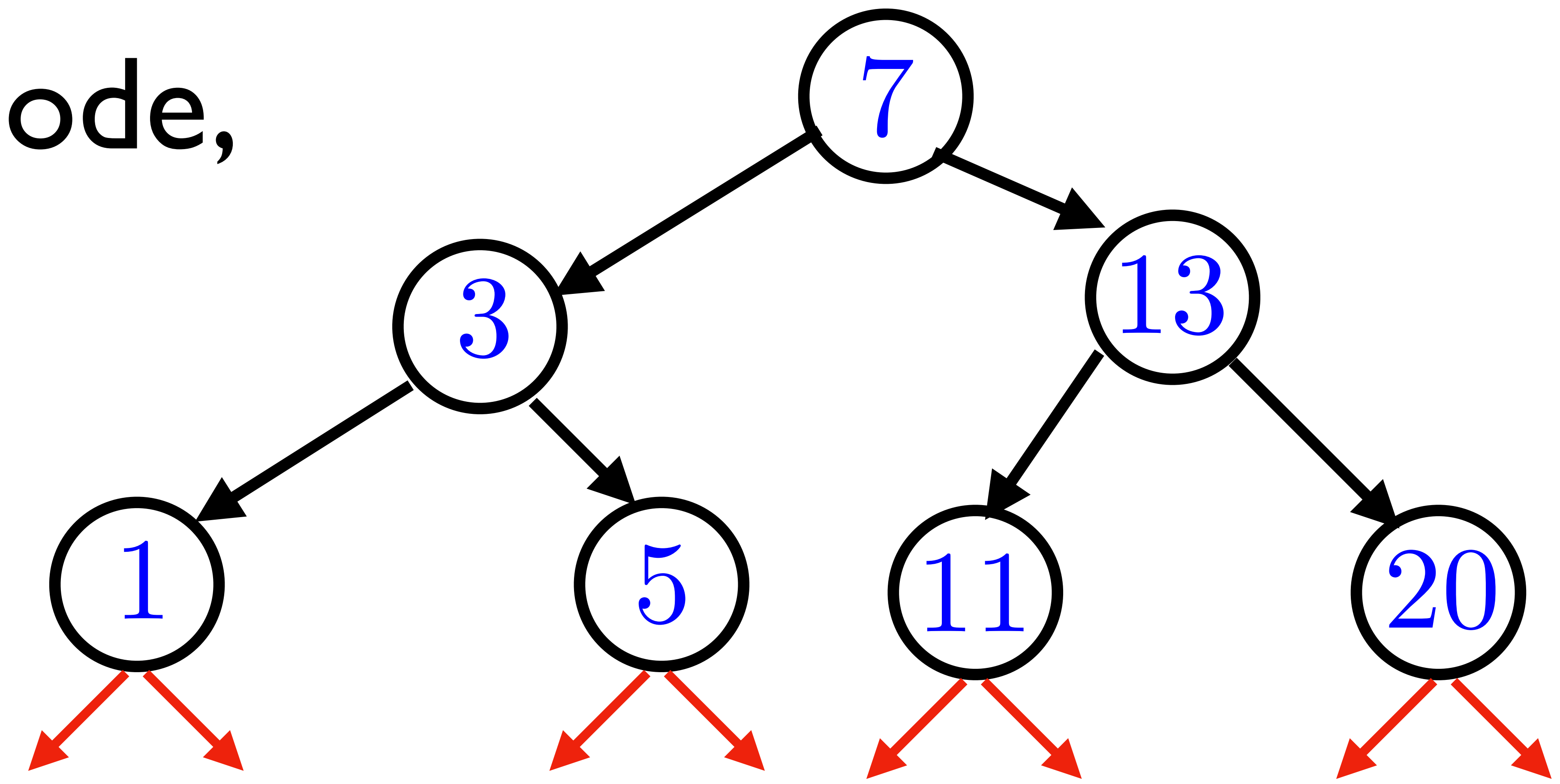
In-order traversal

In-order traversal

Another natural operation we may want from a BST is to extract all the keys in order.

This can be done by an in-order traversal of the tree.

To print keys in order, we want to first print all keys in a node's left subtree, then print the node, then print all keys in the trees right subtree.



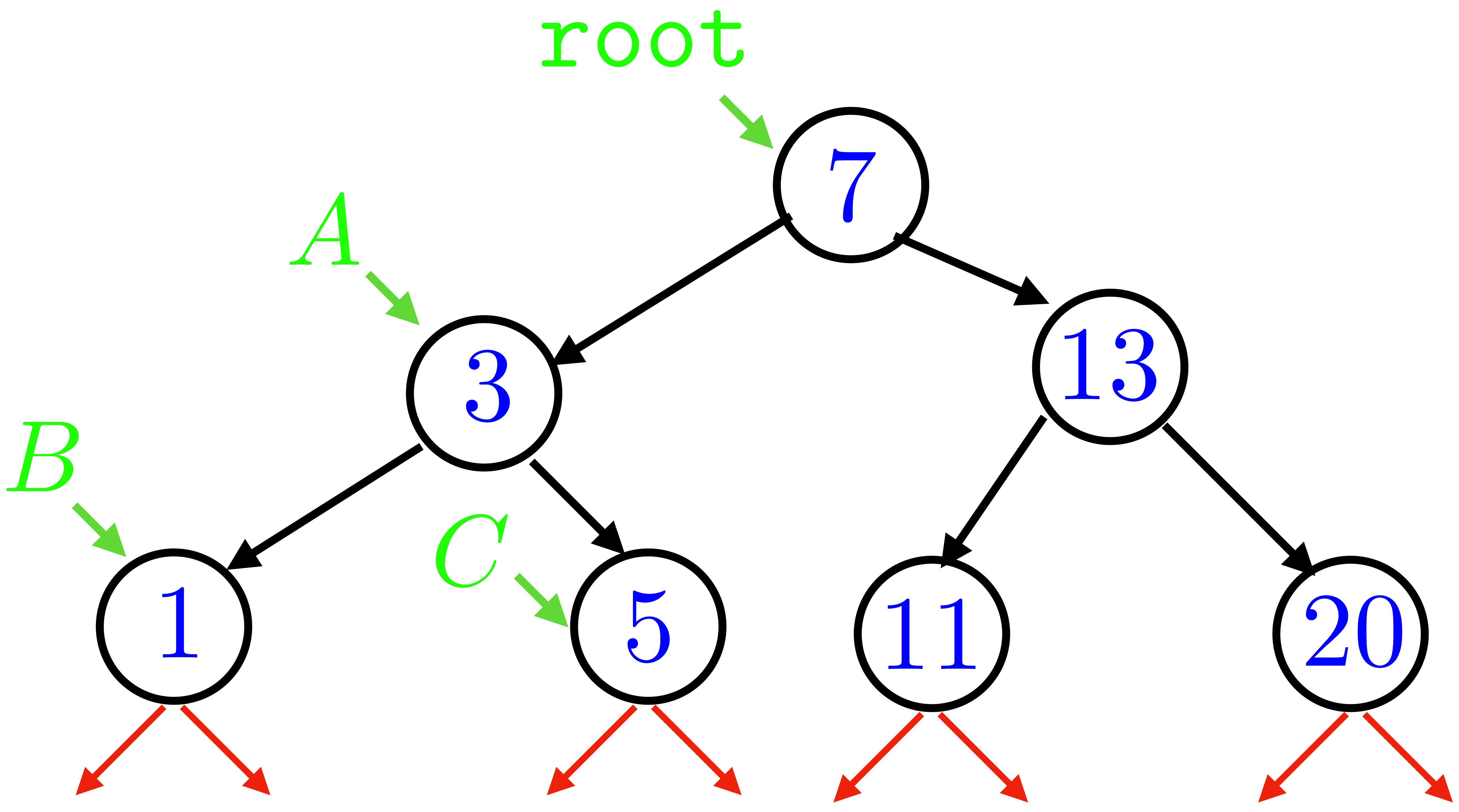
In-order traversal

To print keys in order, we want to first print all keys in a node's left subtree, then print the node, then print all keys in the trees right subtree.

This gives a simple recursive implementation of in-order traversal.

```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```



calls

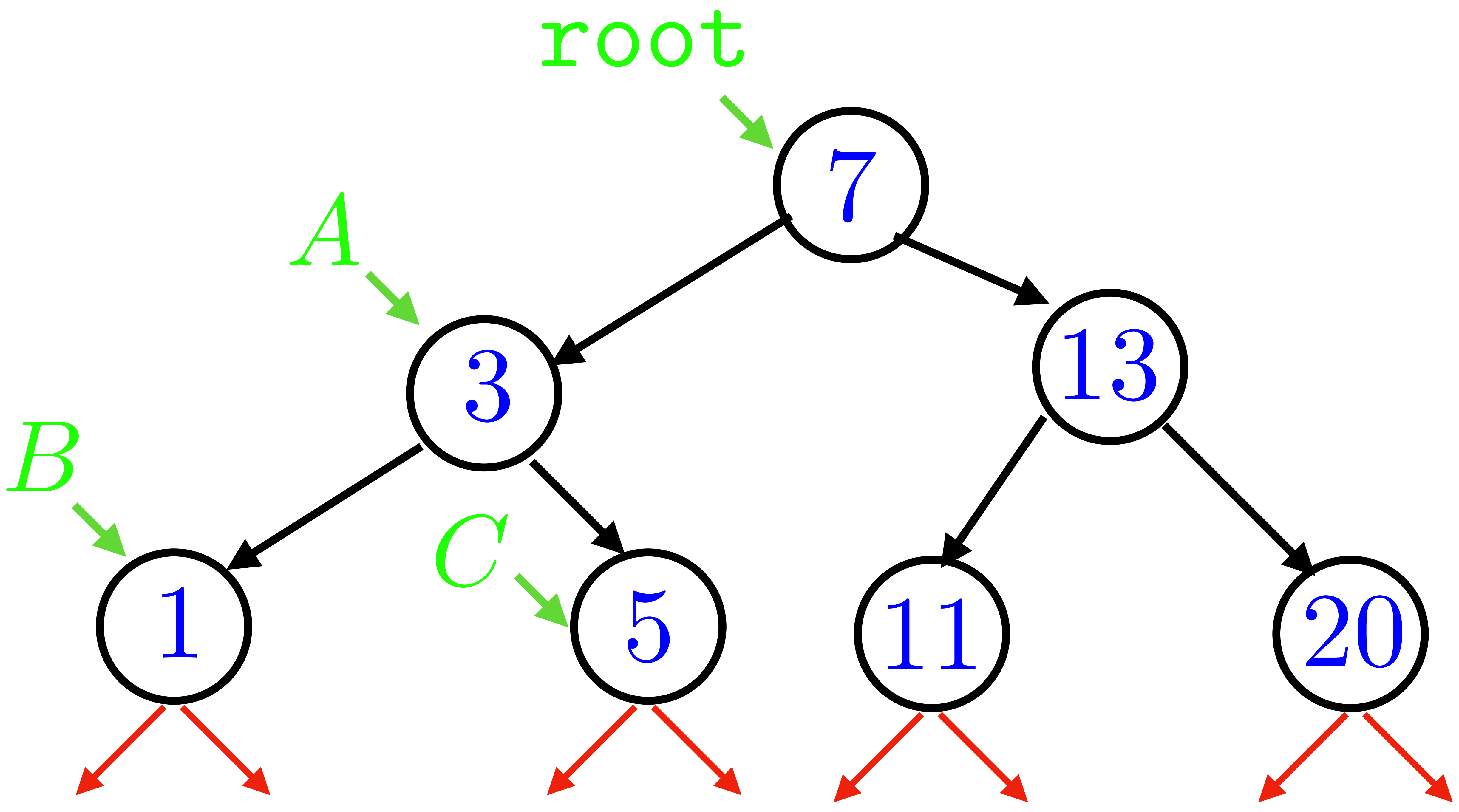
output

```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls

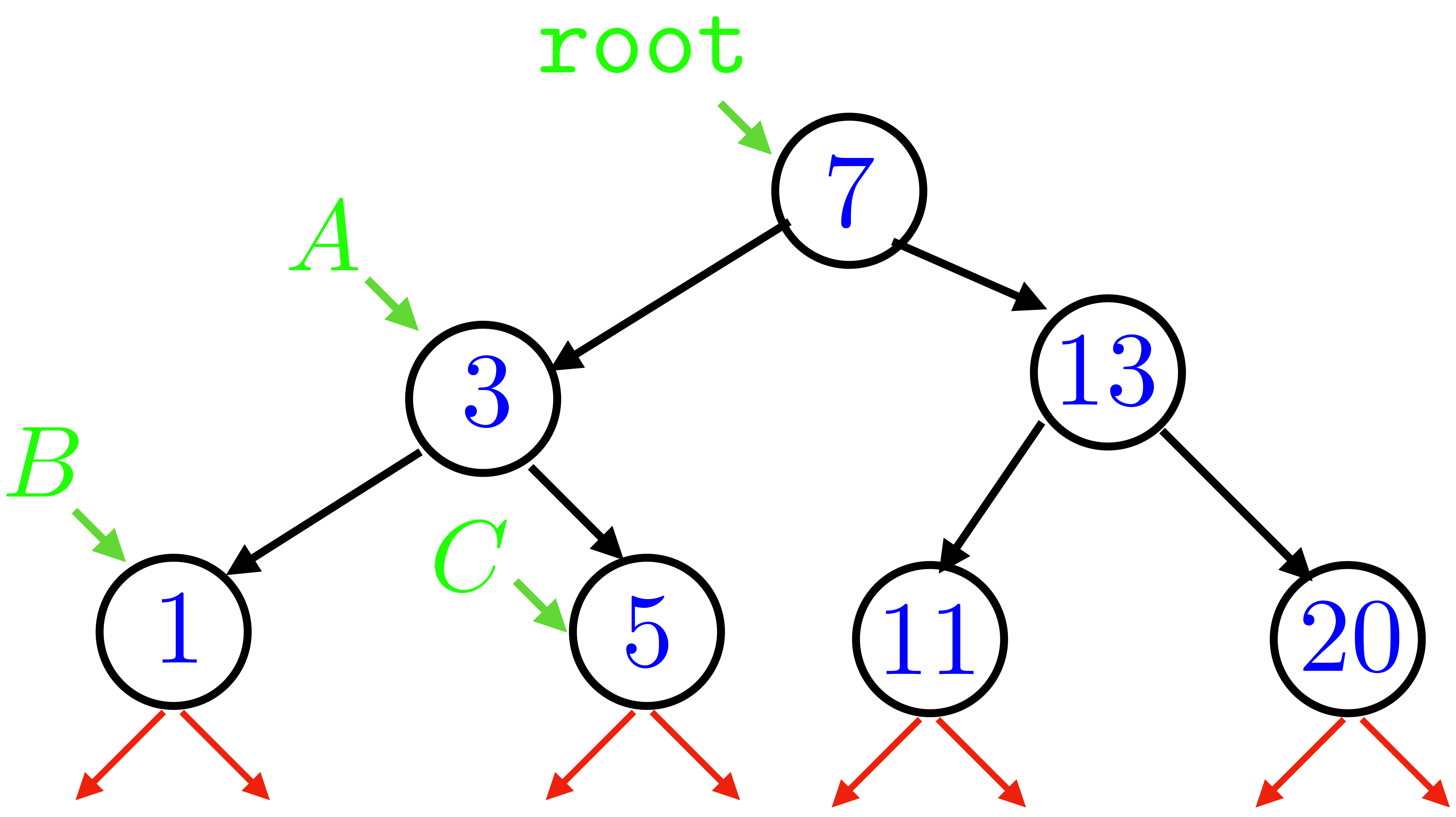
print(root)

output



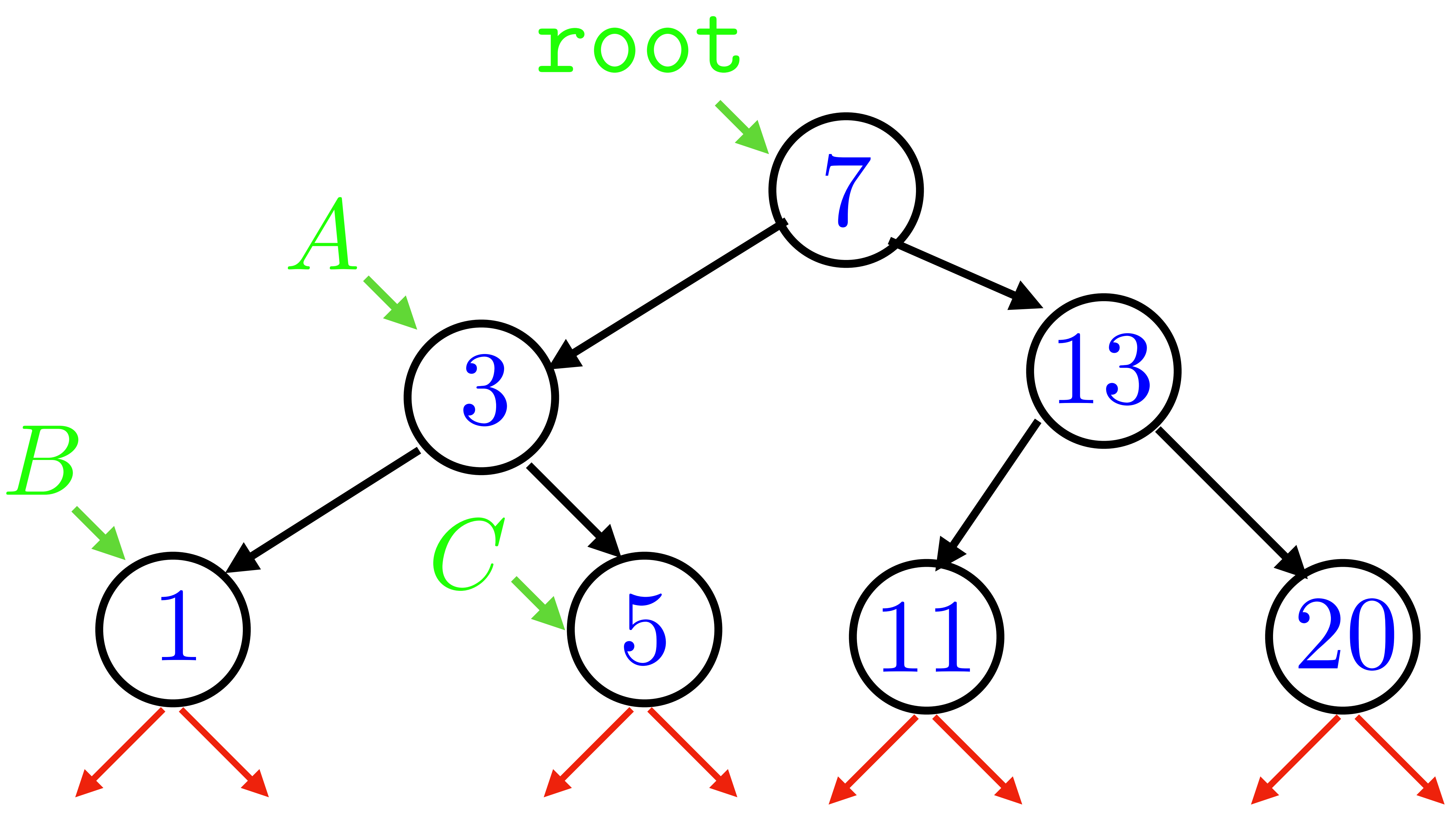
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	
print(A)	



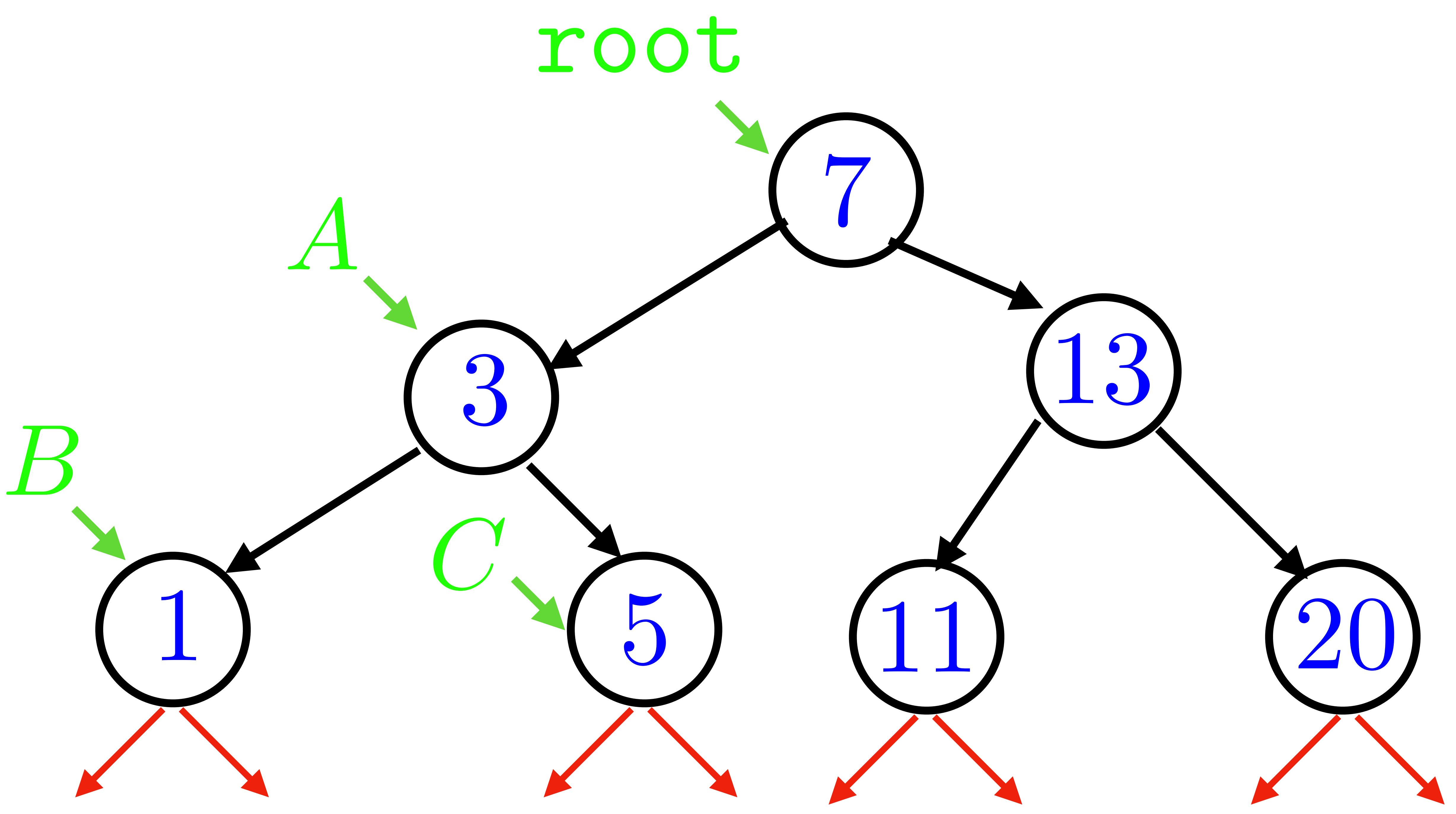
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	
print(A)	
print(B)	



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	
print(A)	
print(B)	
print(nullptr)	



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

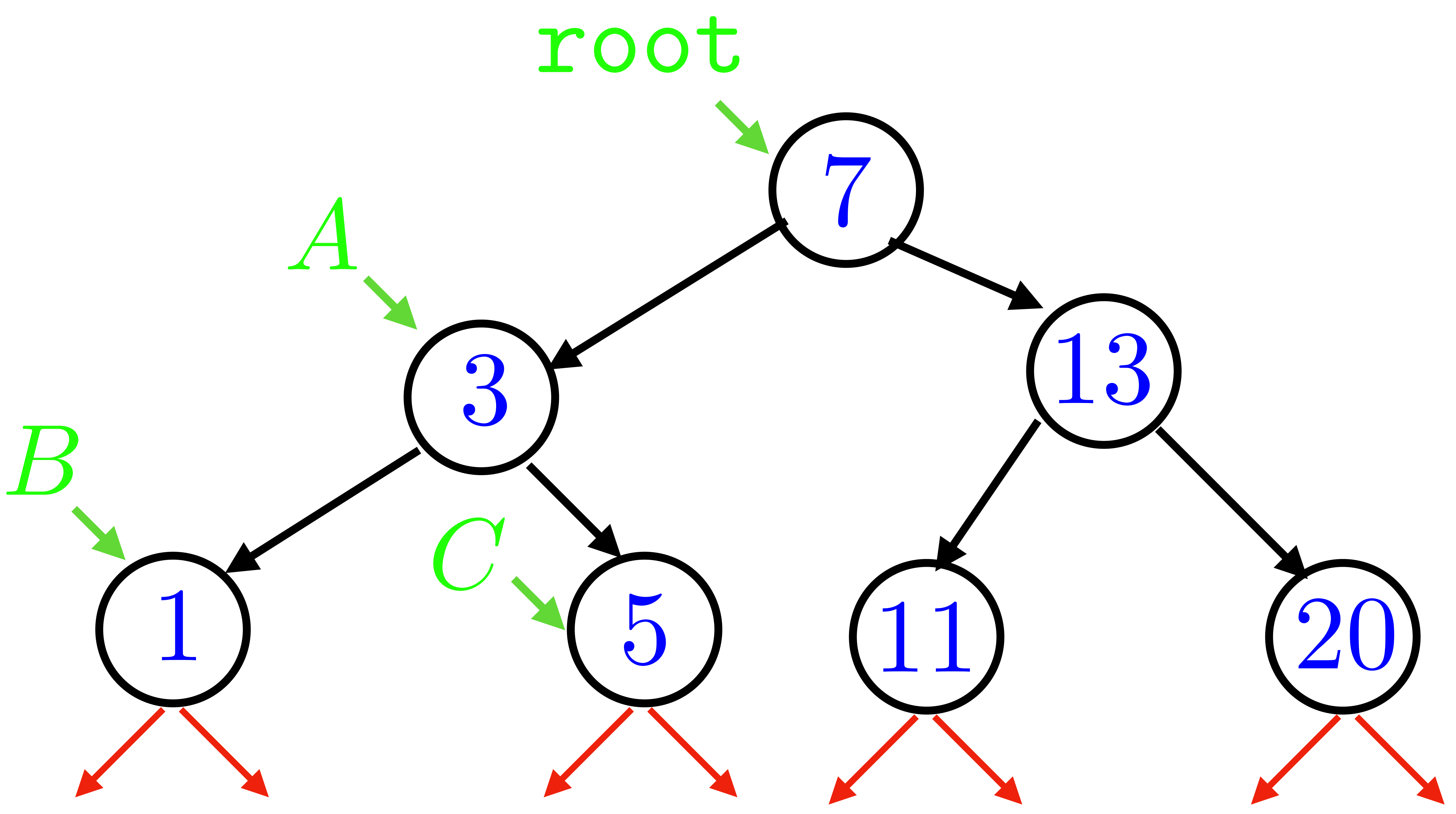
calls

print(root)

print(A)

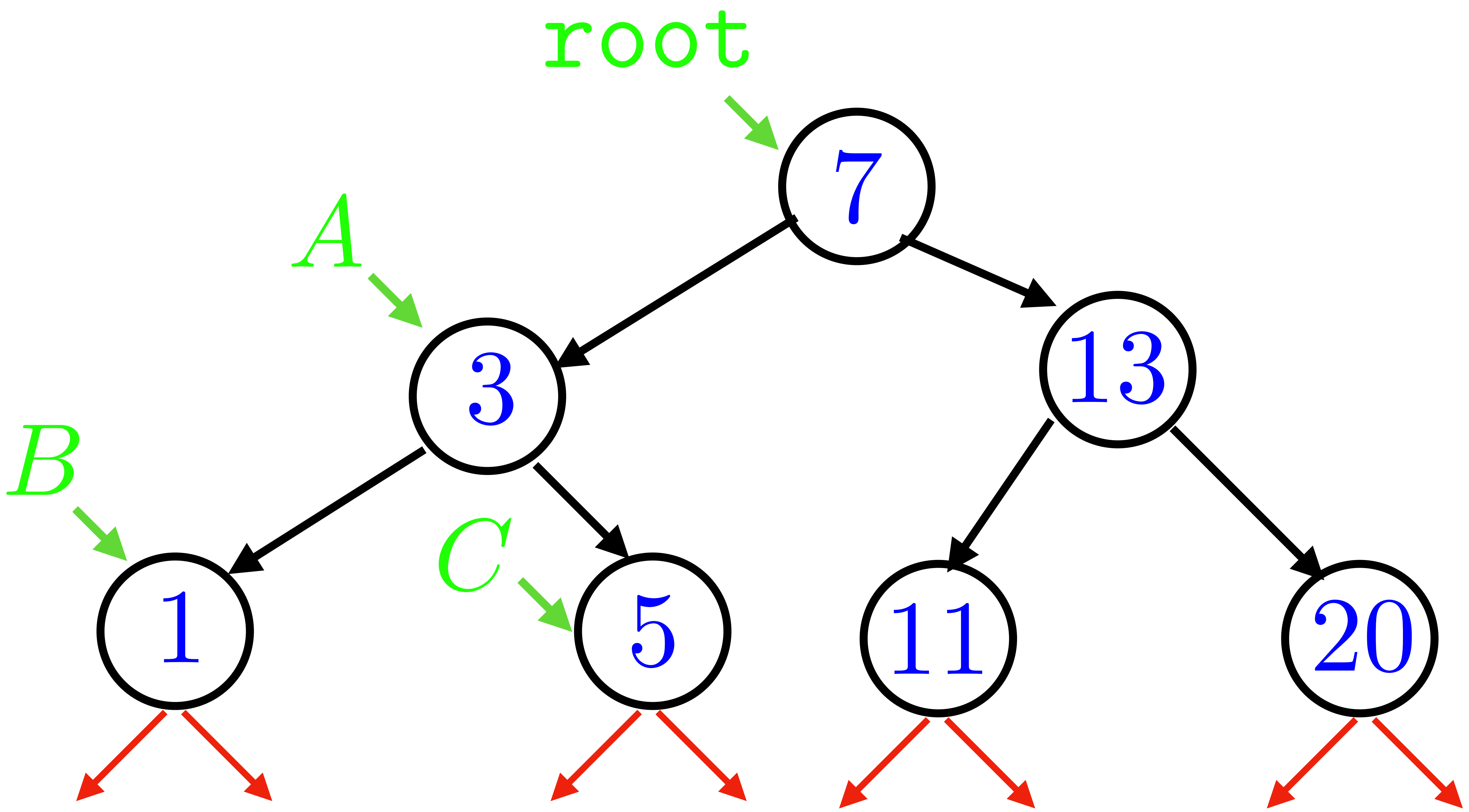
print(B)

output



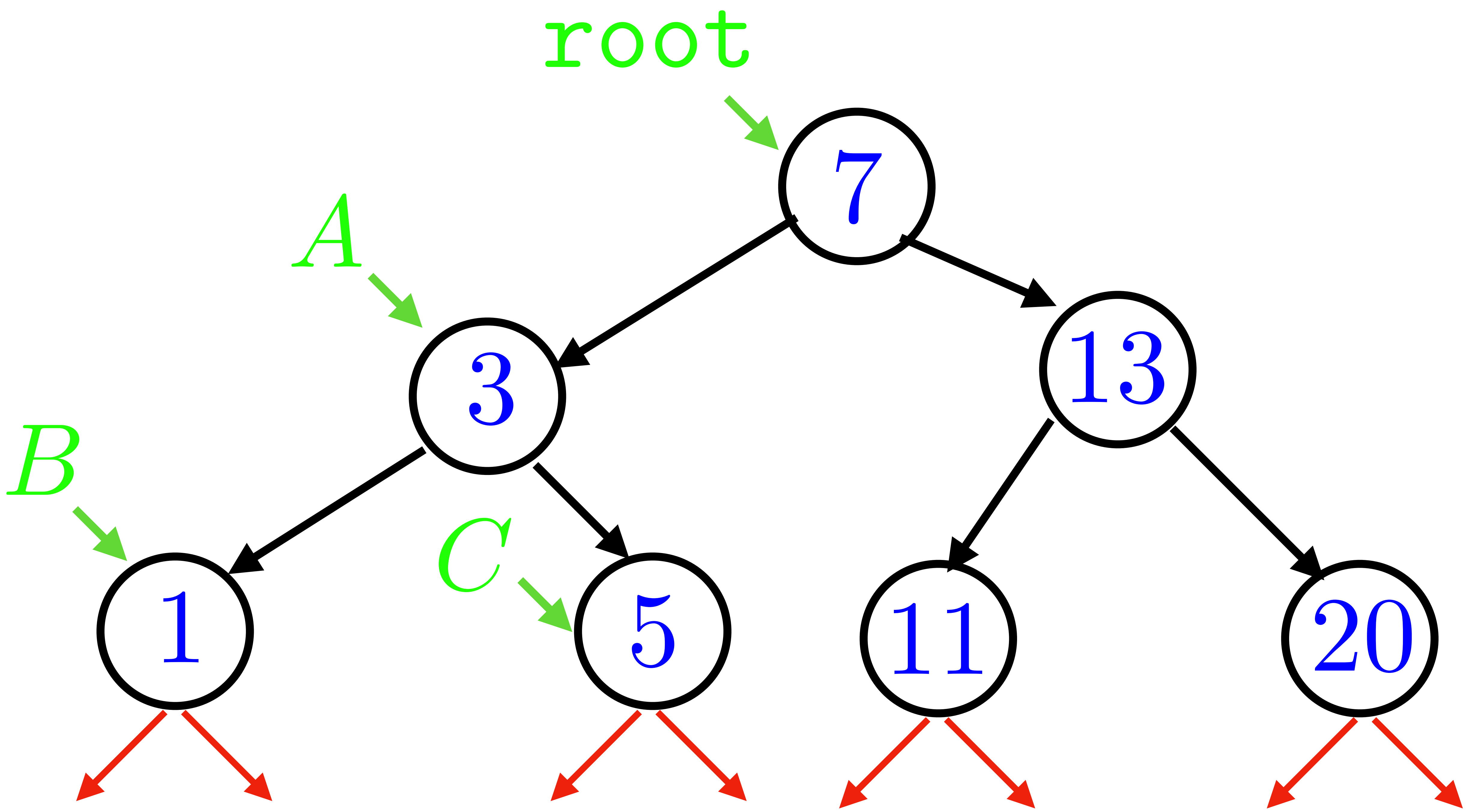
```cpp
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
→   std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	
print(A)	
print(B)	



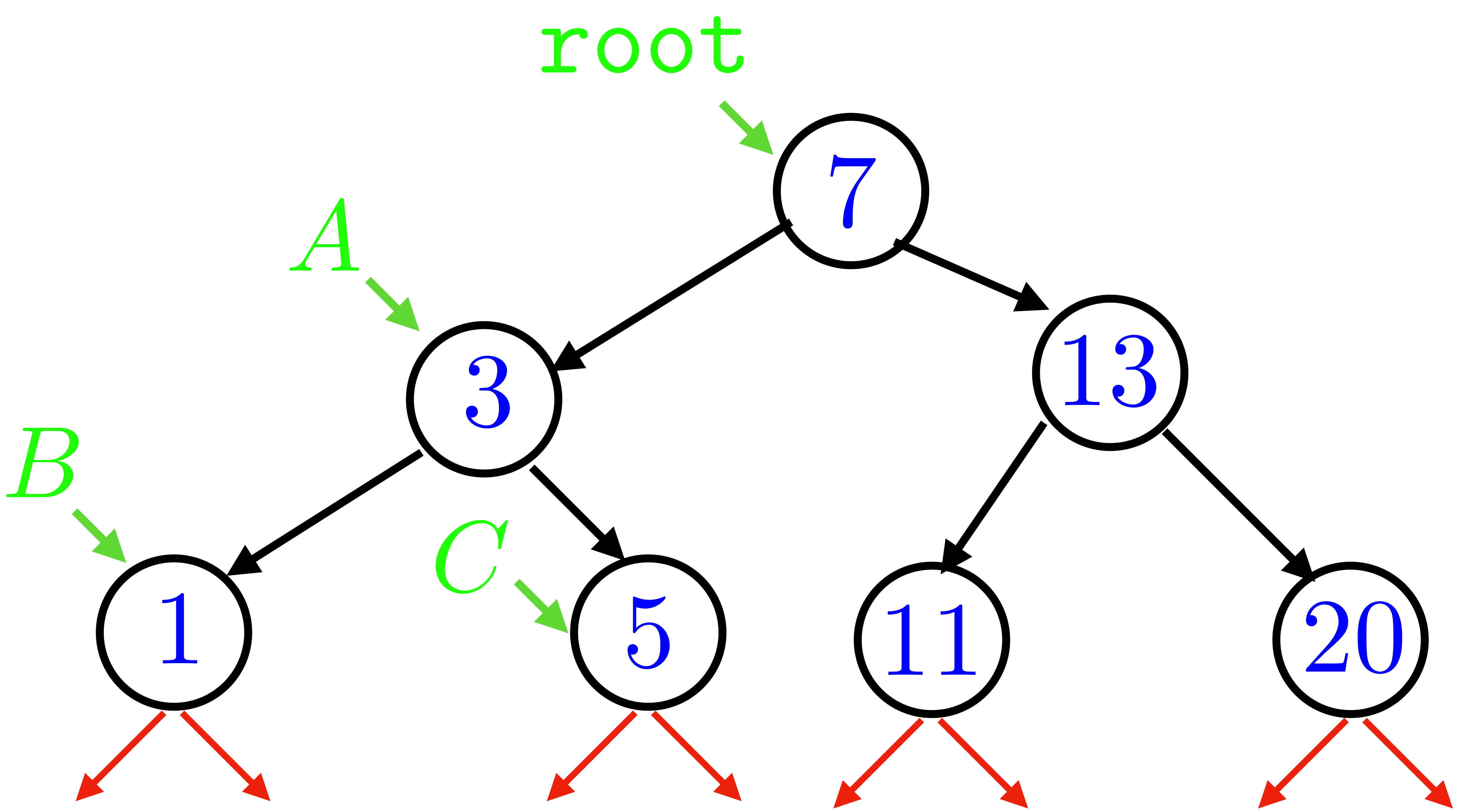
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
→   std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	
print(B)	



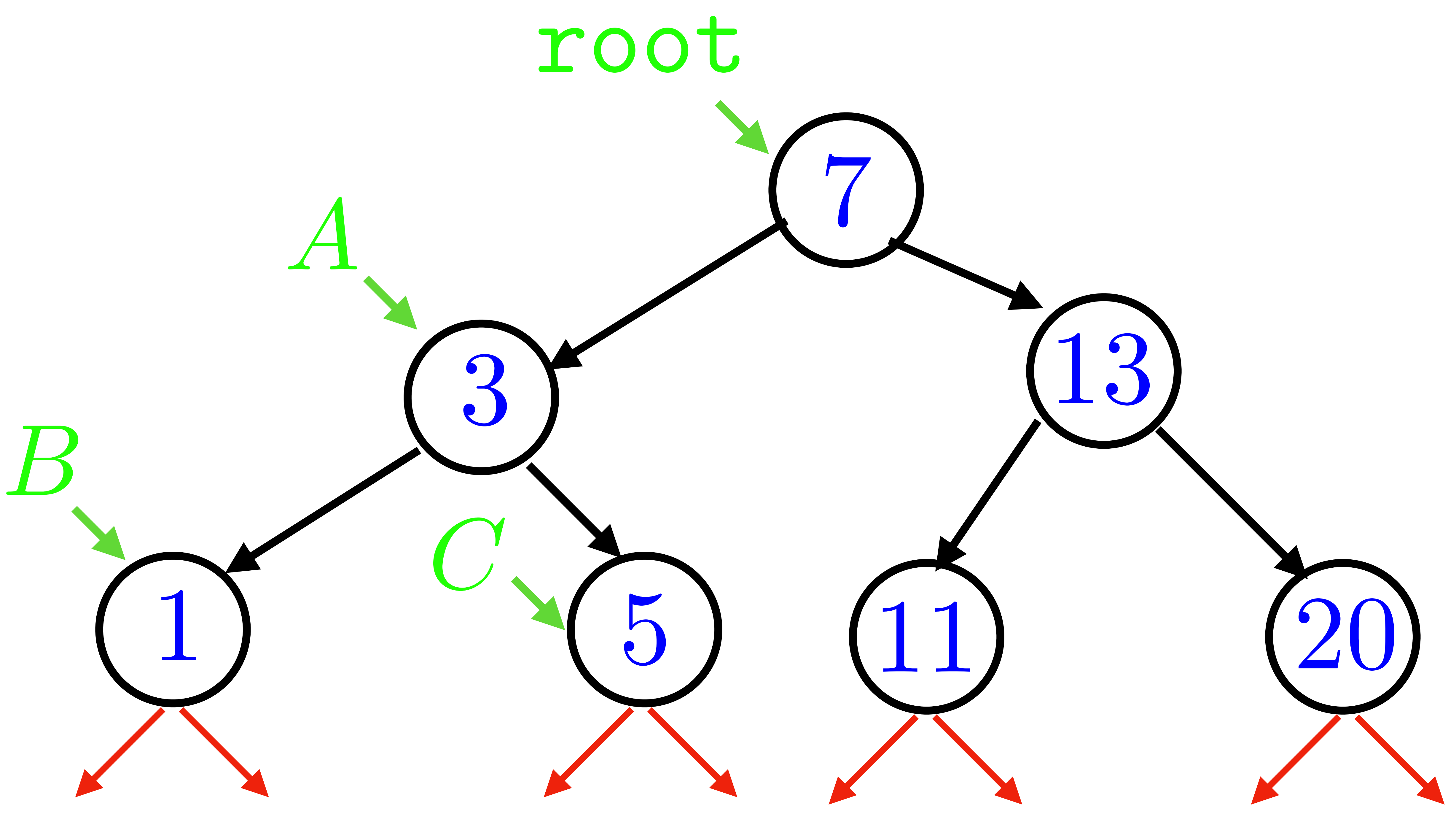
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
→   print(node->right);
}
```

calls	output
print(root)	1
print(A)	
print(B)	



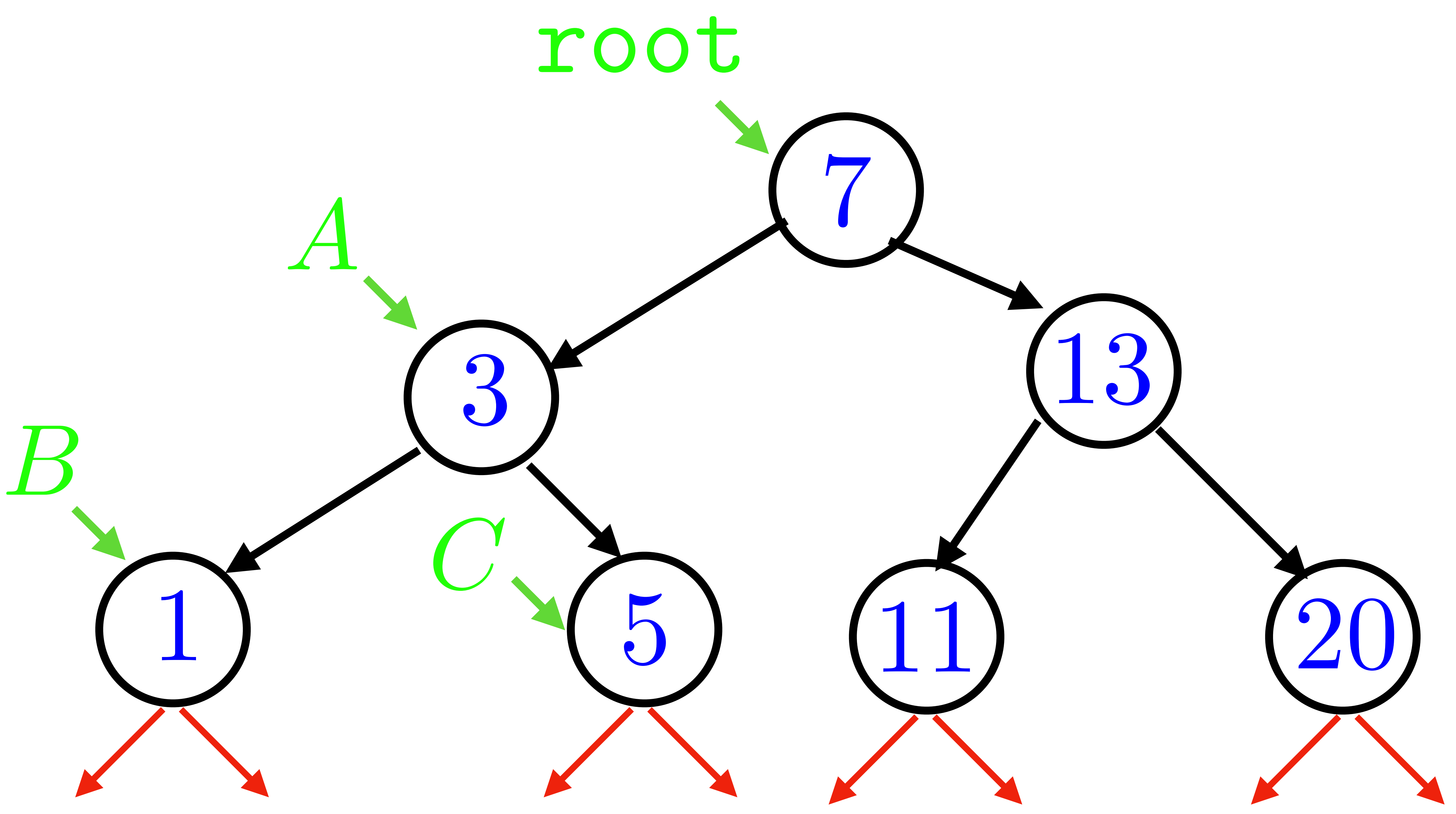
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	
print(B)	
print(nullptr)	



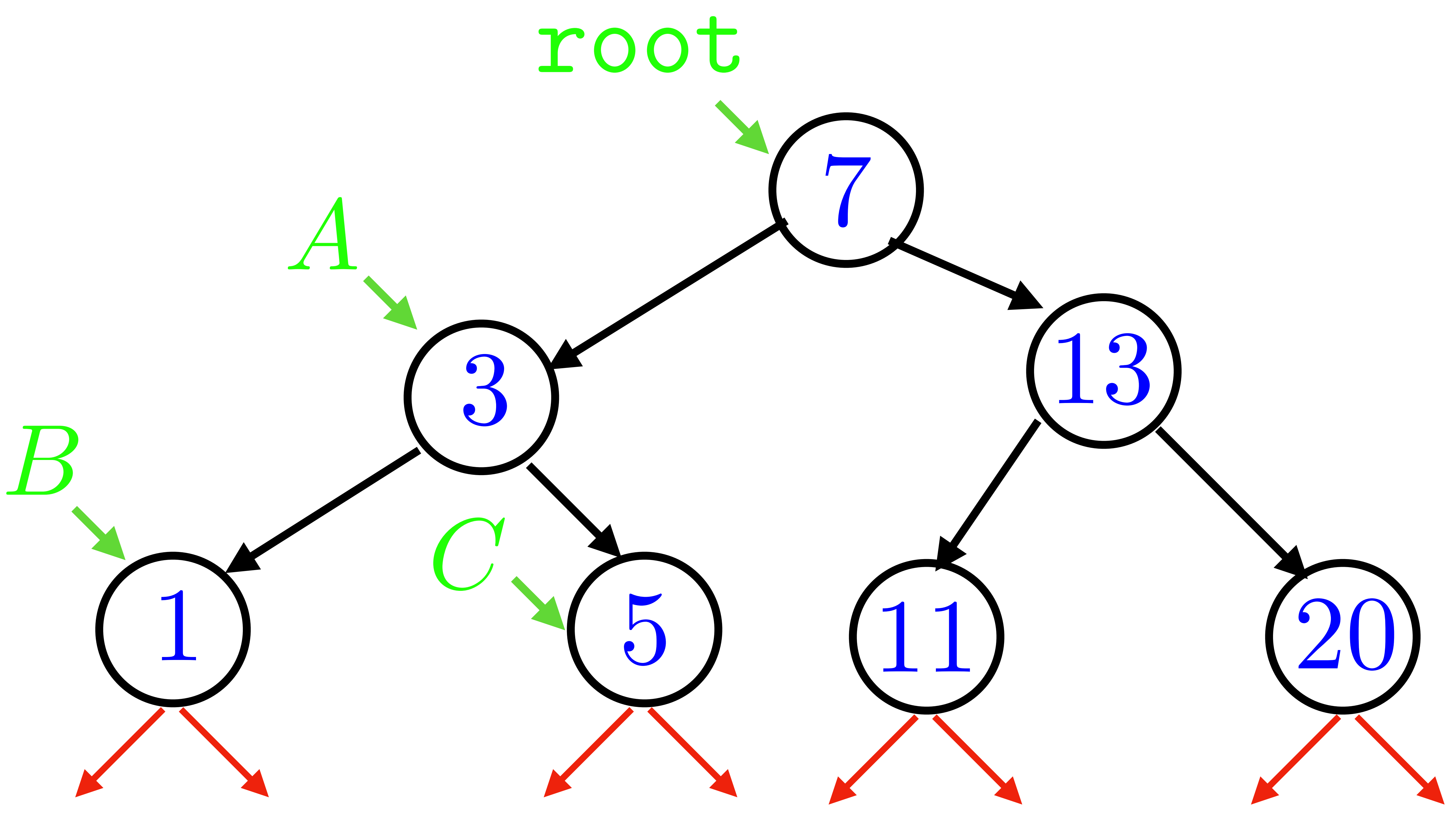
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	
print(B)	



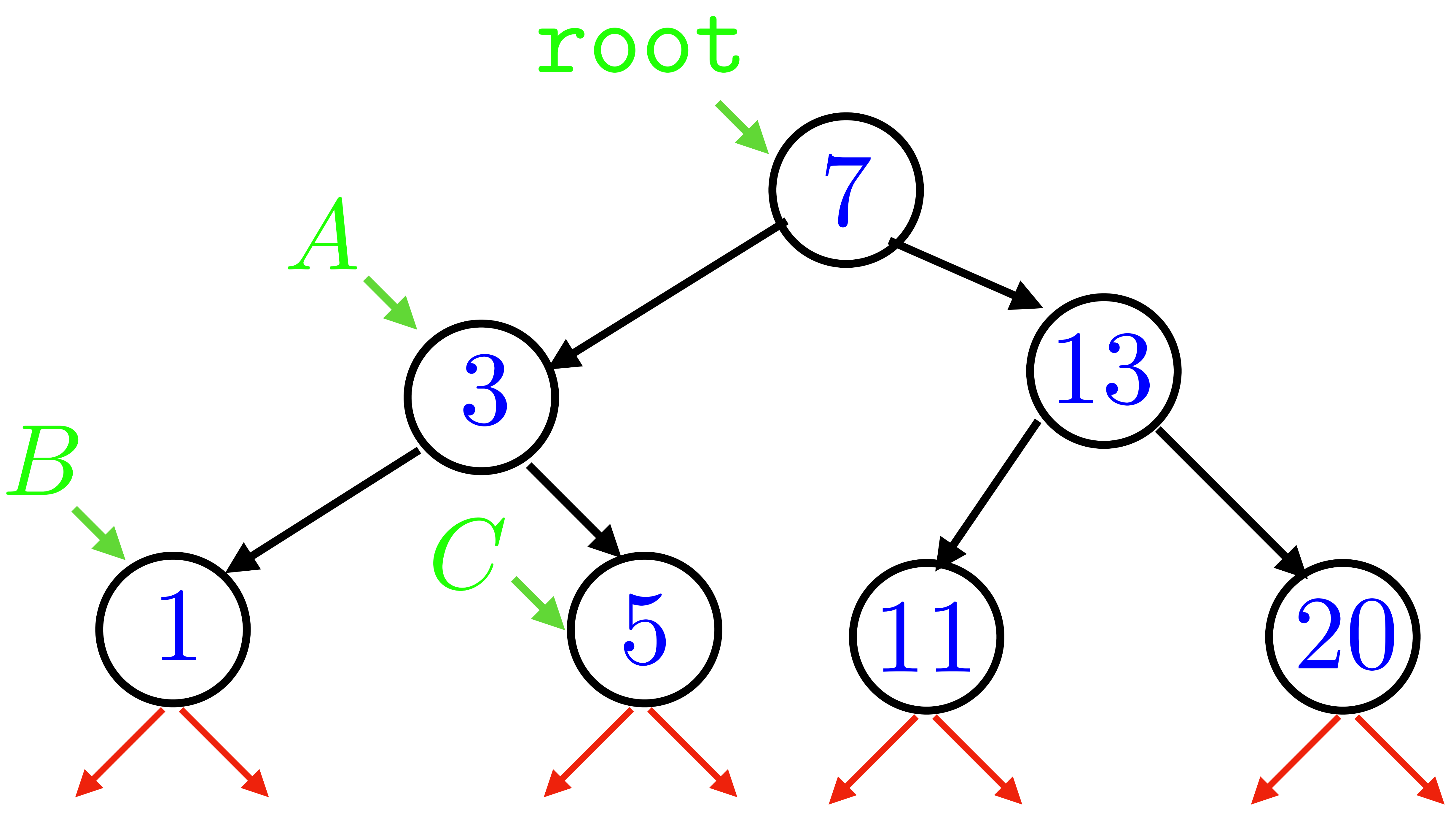
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	



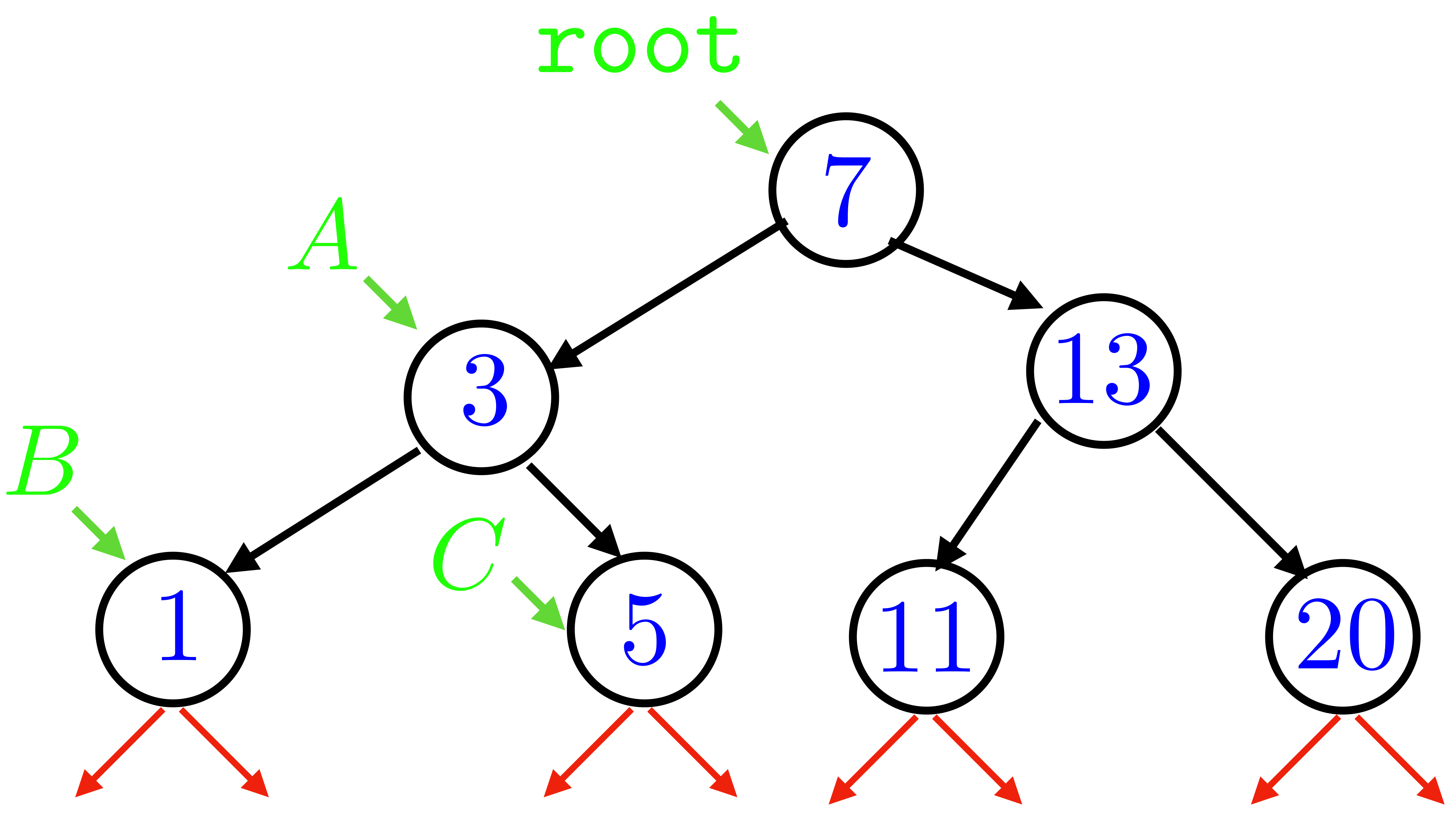
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
→   std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
→   std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	3



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
  → print(node->right);
}
```

calls

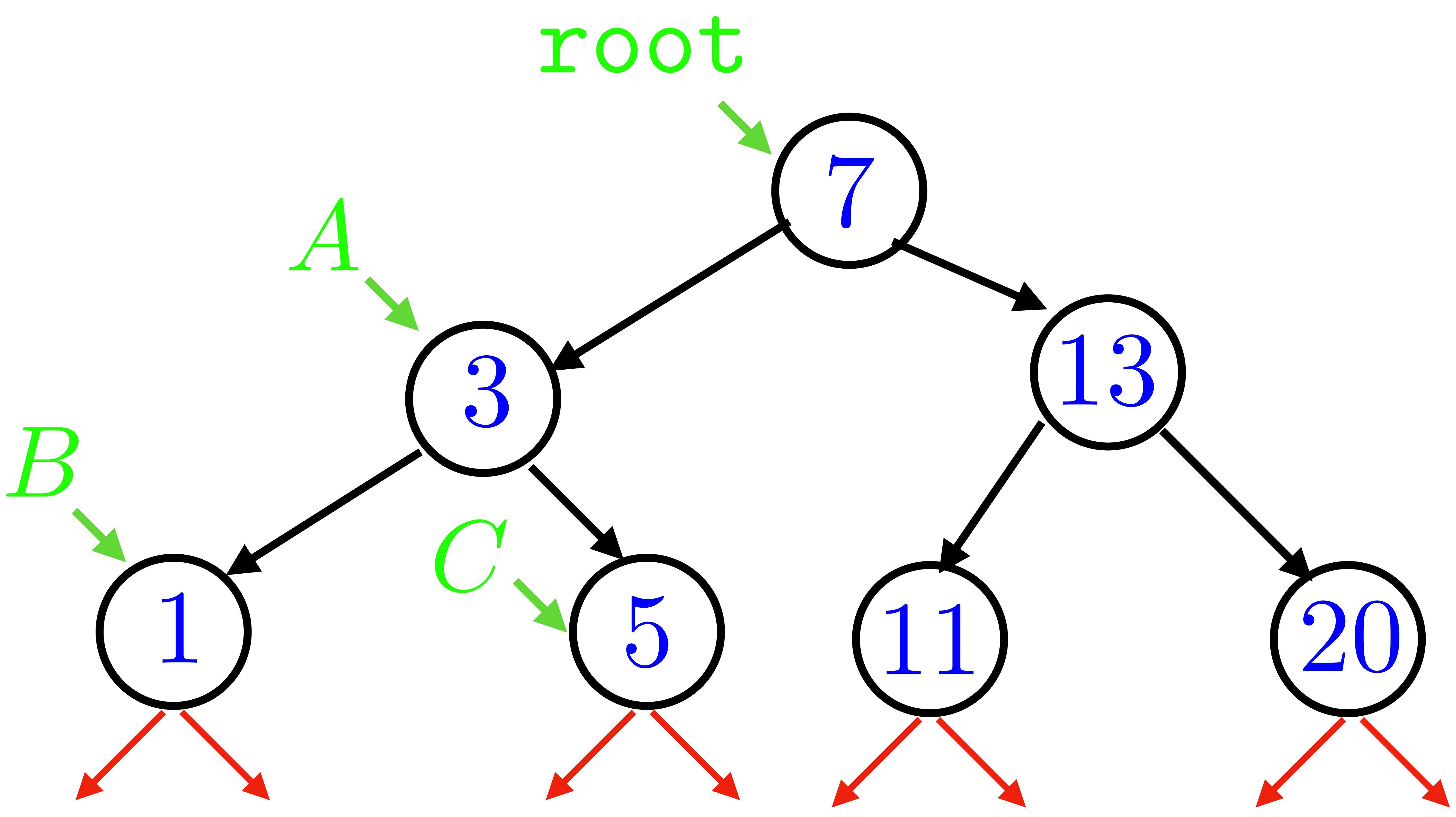
print(root)

print(A)

output

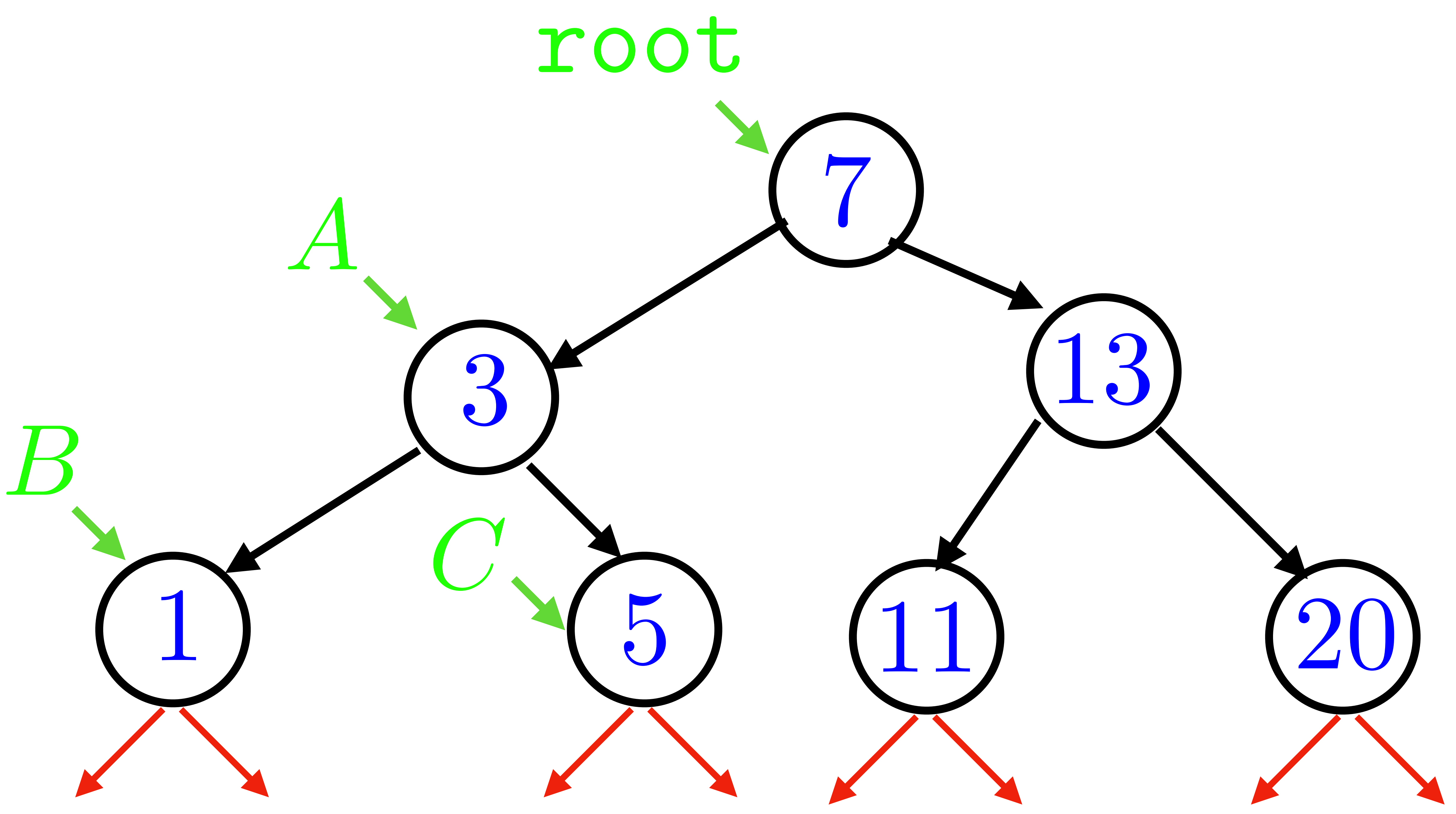
1

3



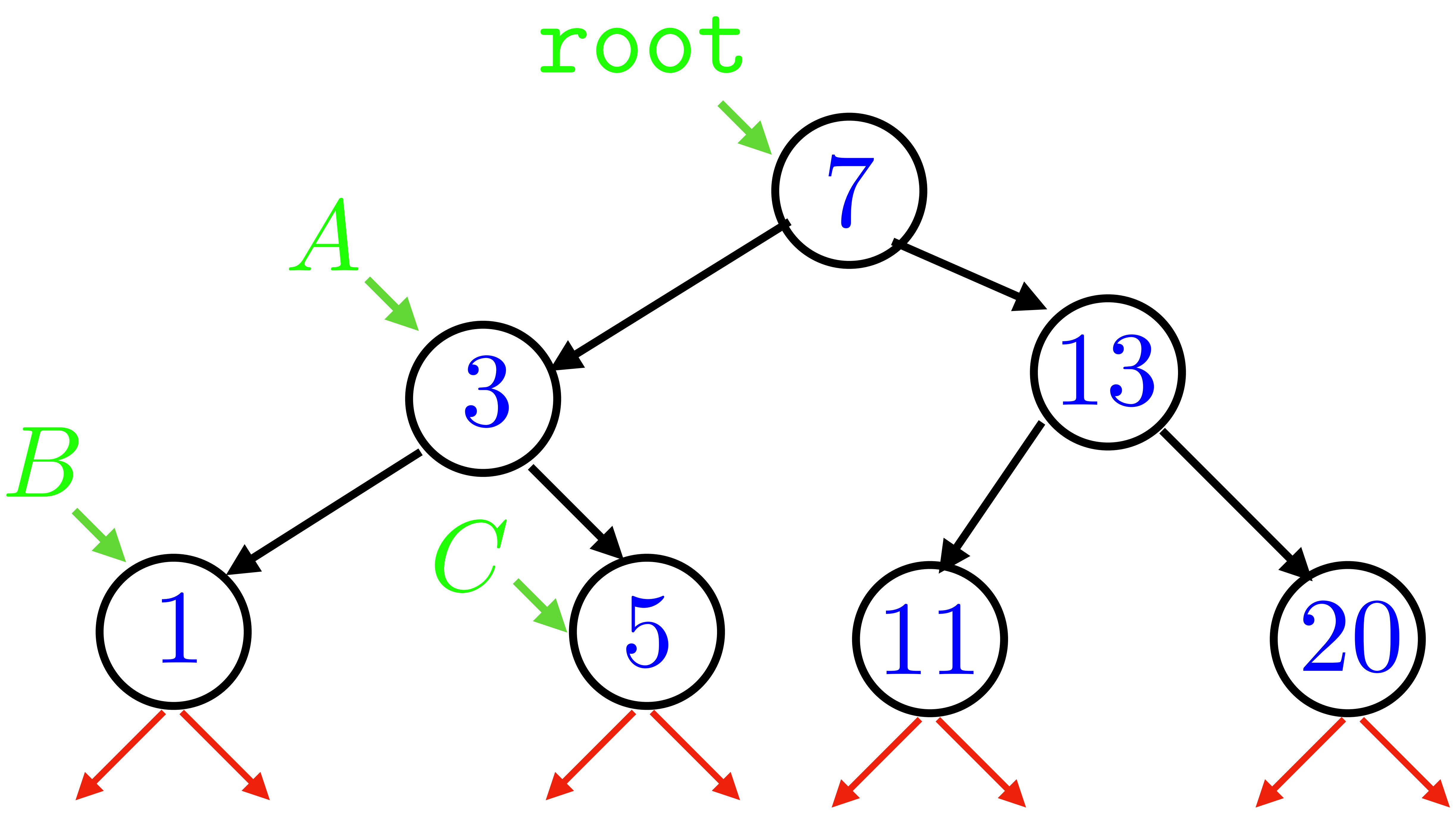
```cpp
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	3
print(C)	



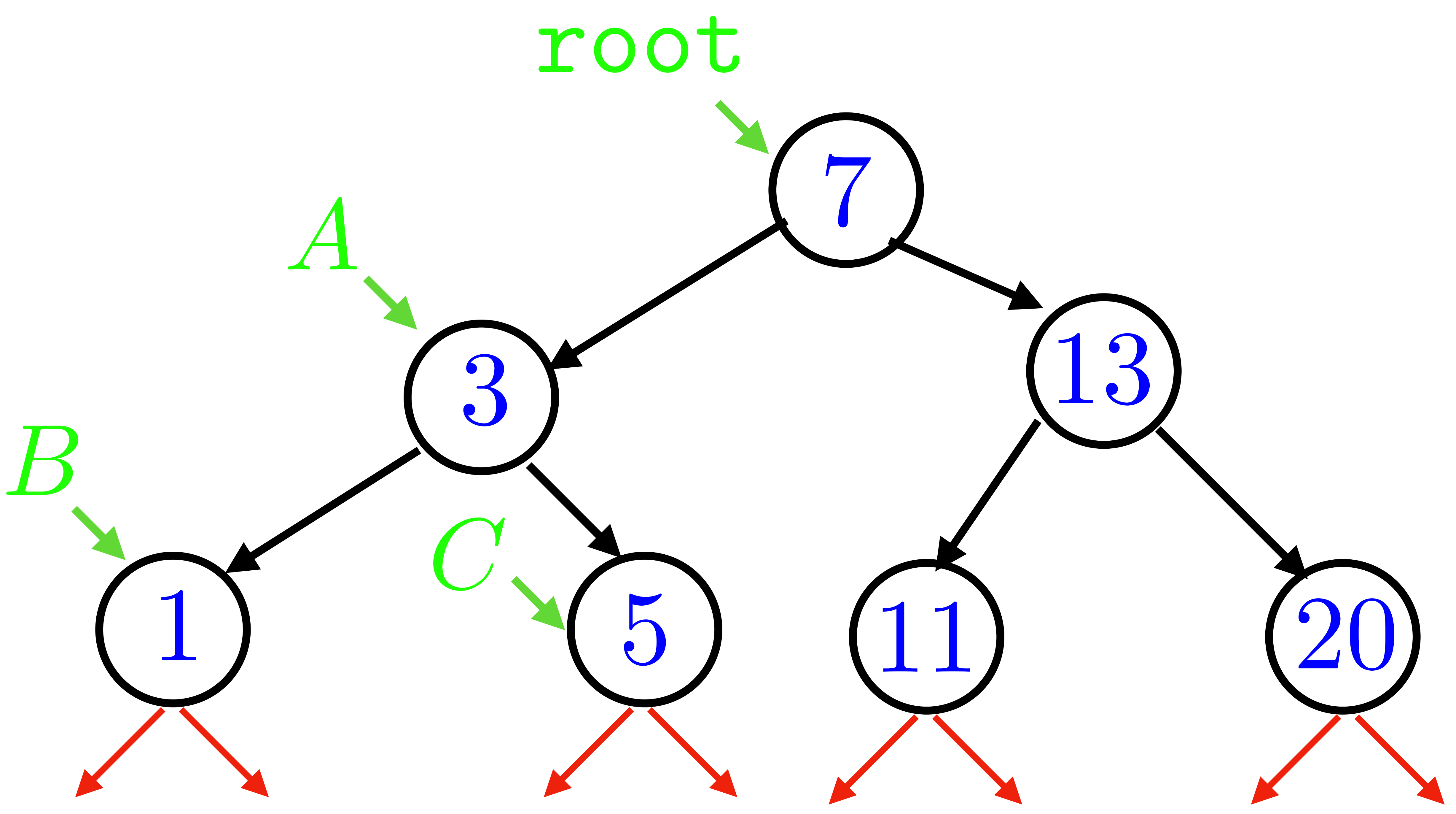
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	3
print(C)	5



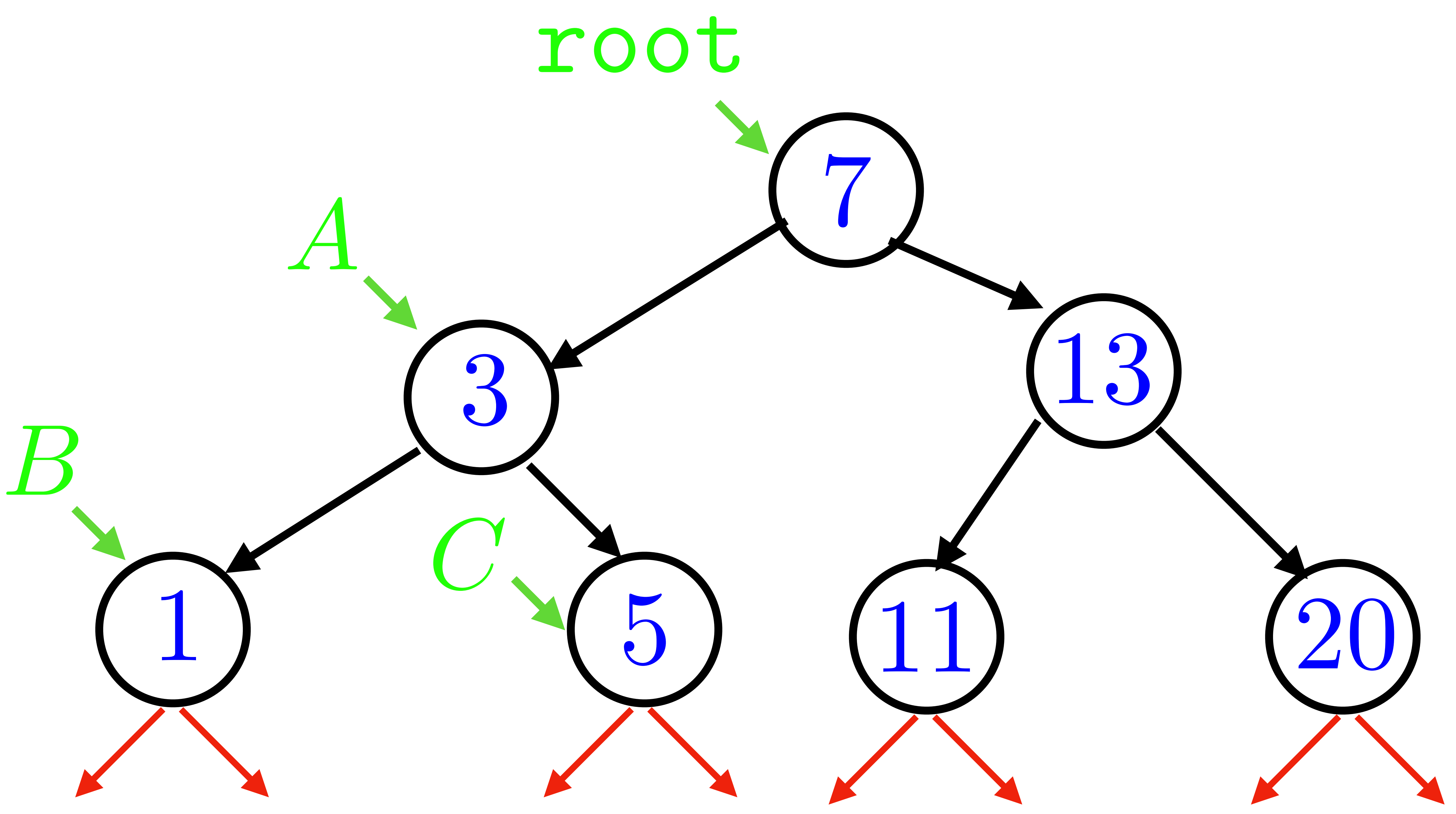
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
print(A)	3
	5



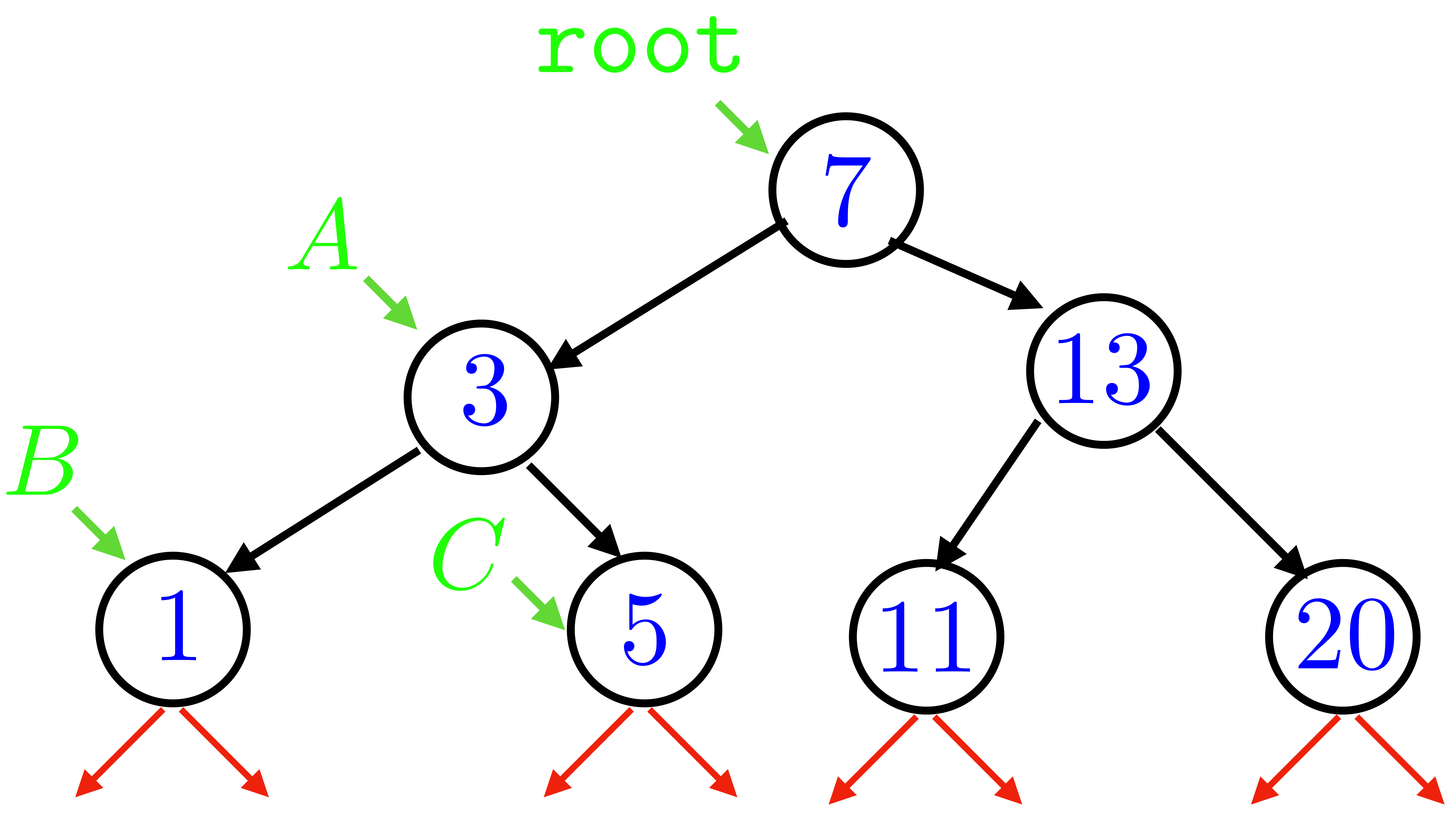
```cpp
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
	3
	5



```cpp
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
 →  std::cout << node->key << '\n';
    print(node->right);
}
```

calls	output
print(root)	1
	3
	5



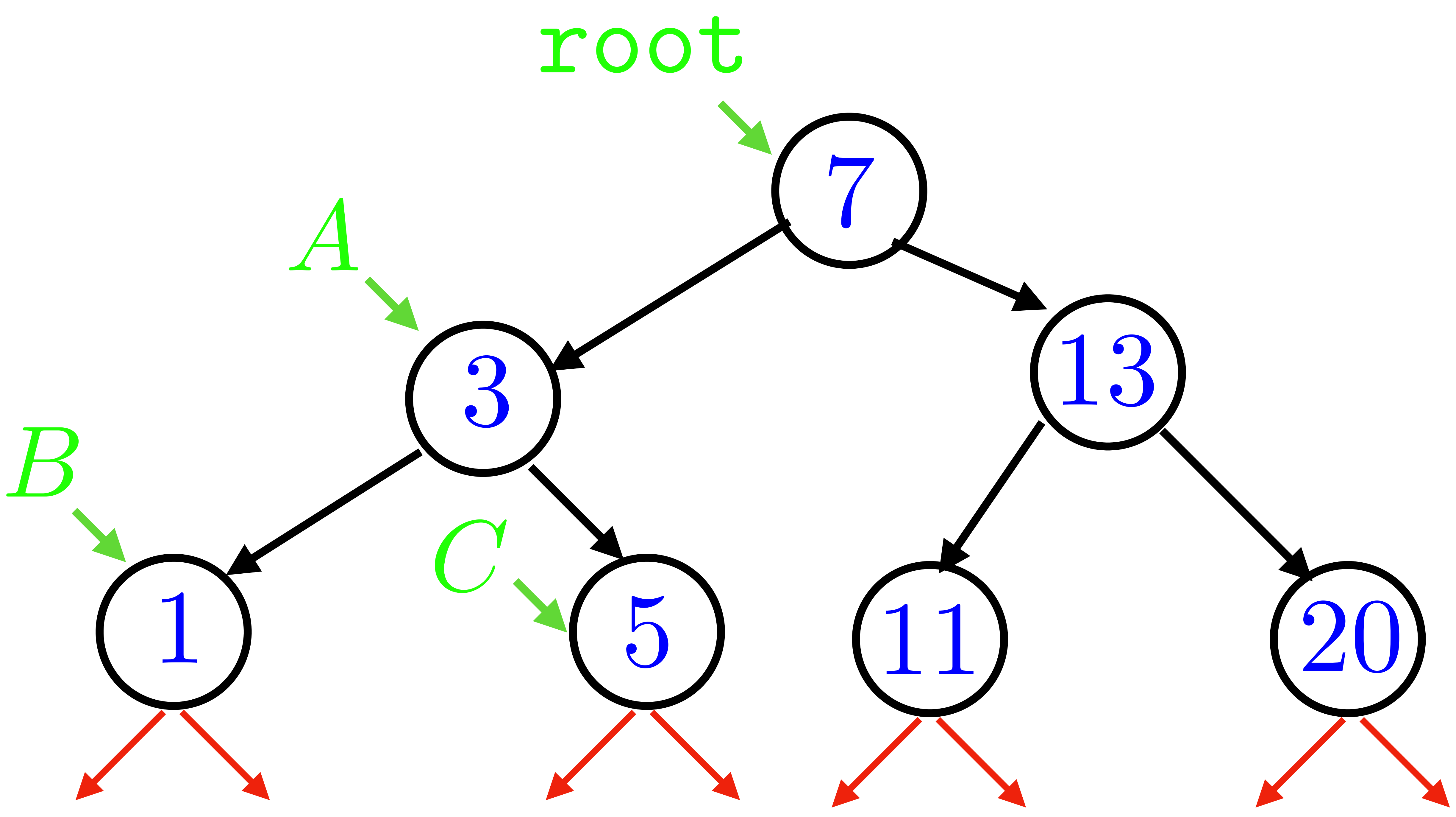
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
 →  std::cout << node->key << '\n';
    print(node->right);
}
```

calls

print(root)

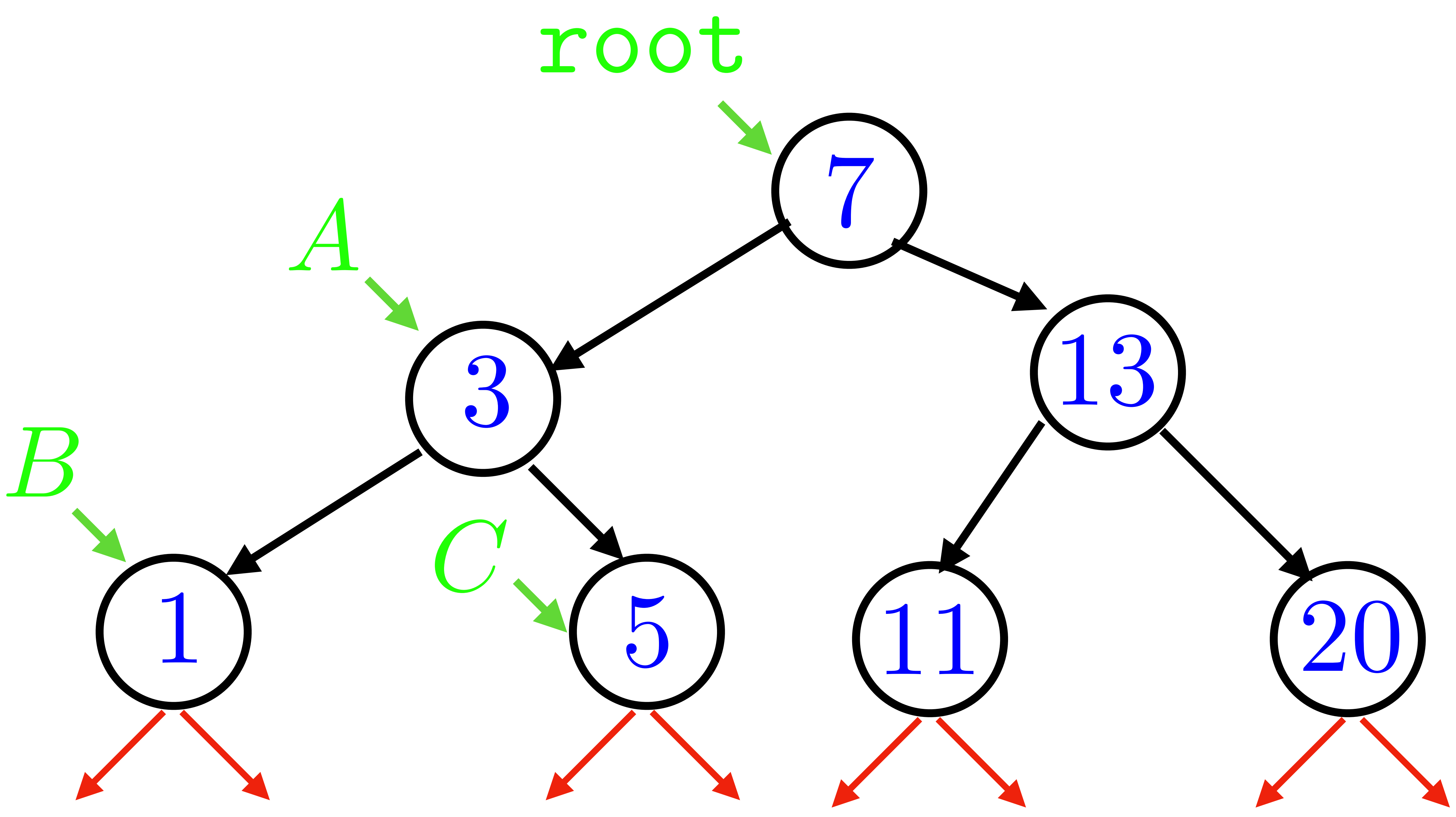
output

1
3
5
7



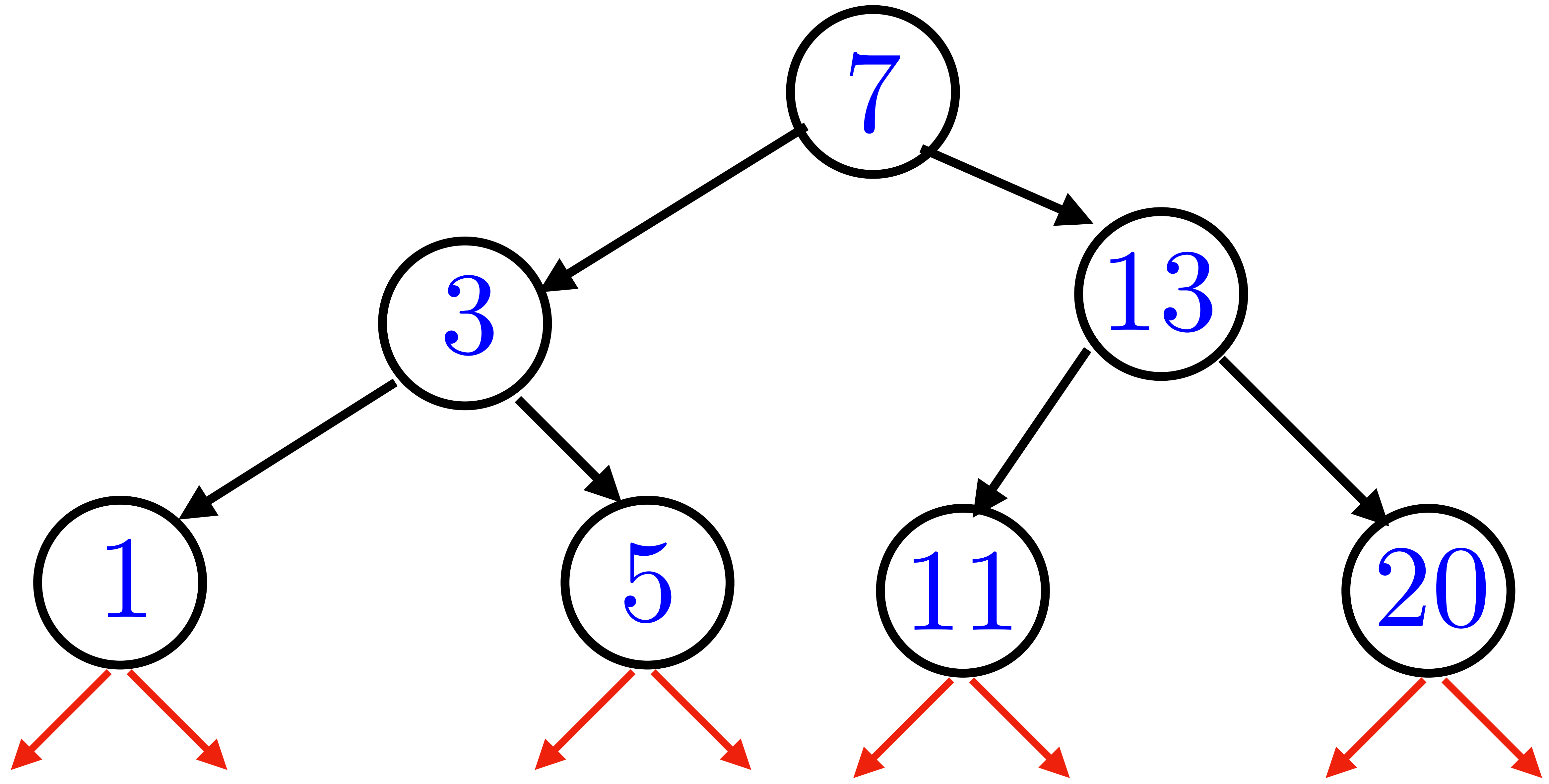
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
→   print(node->right);
}
```

calls	output
print(root)	1
	3
	5
	7



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

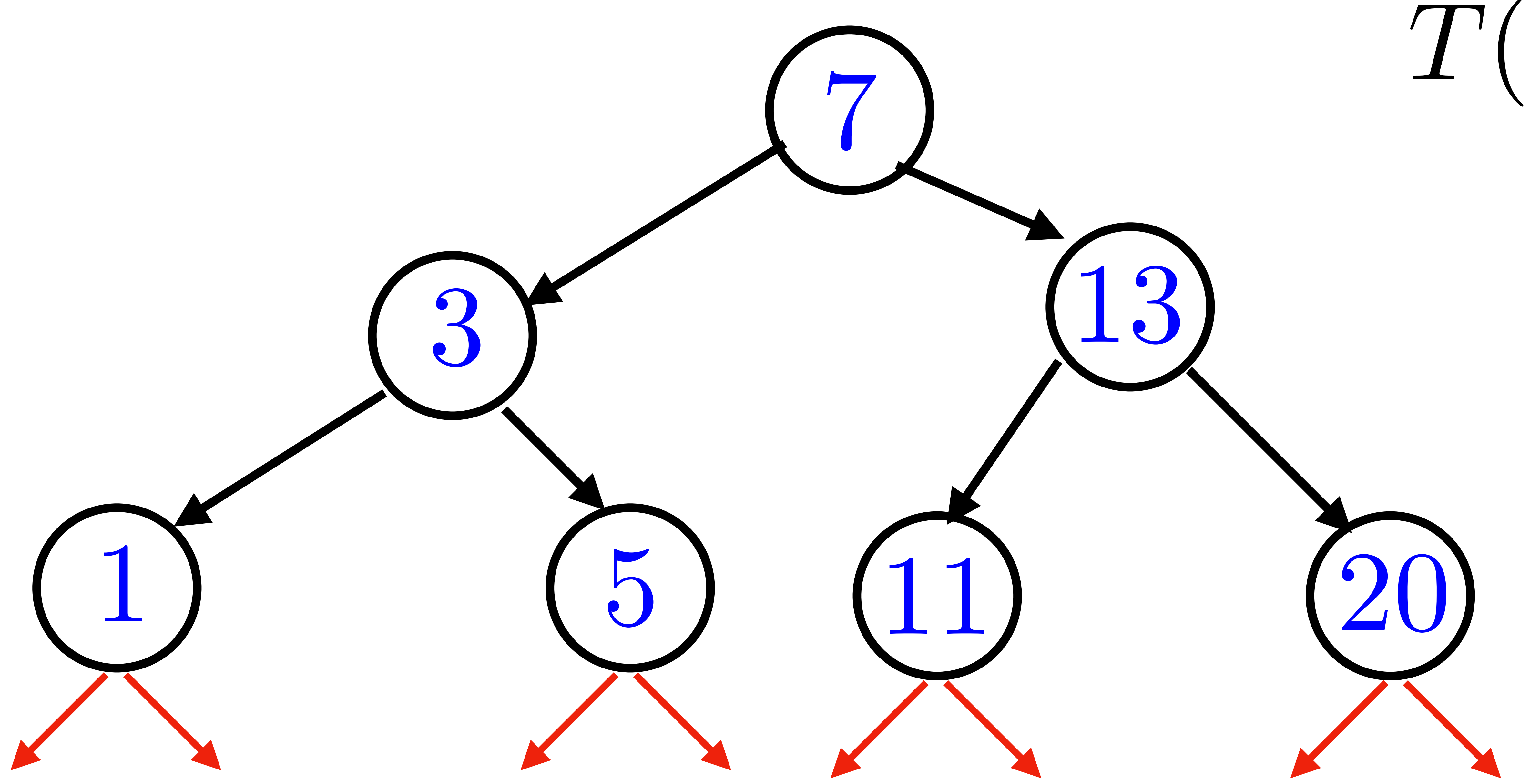
The complexity is $\Theta(n)$.



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

The complexity is $\Theta(n)$.

$T(n) \leq$

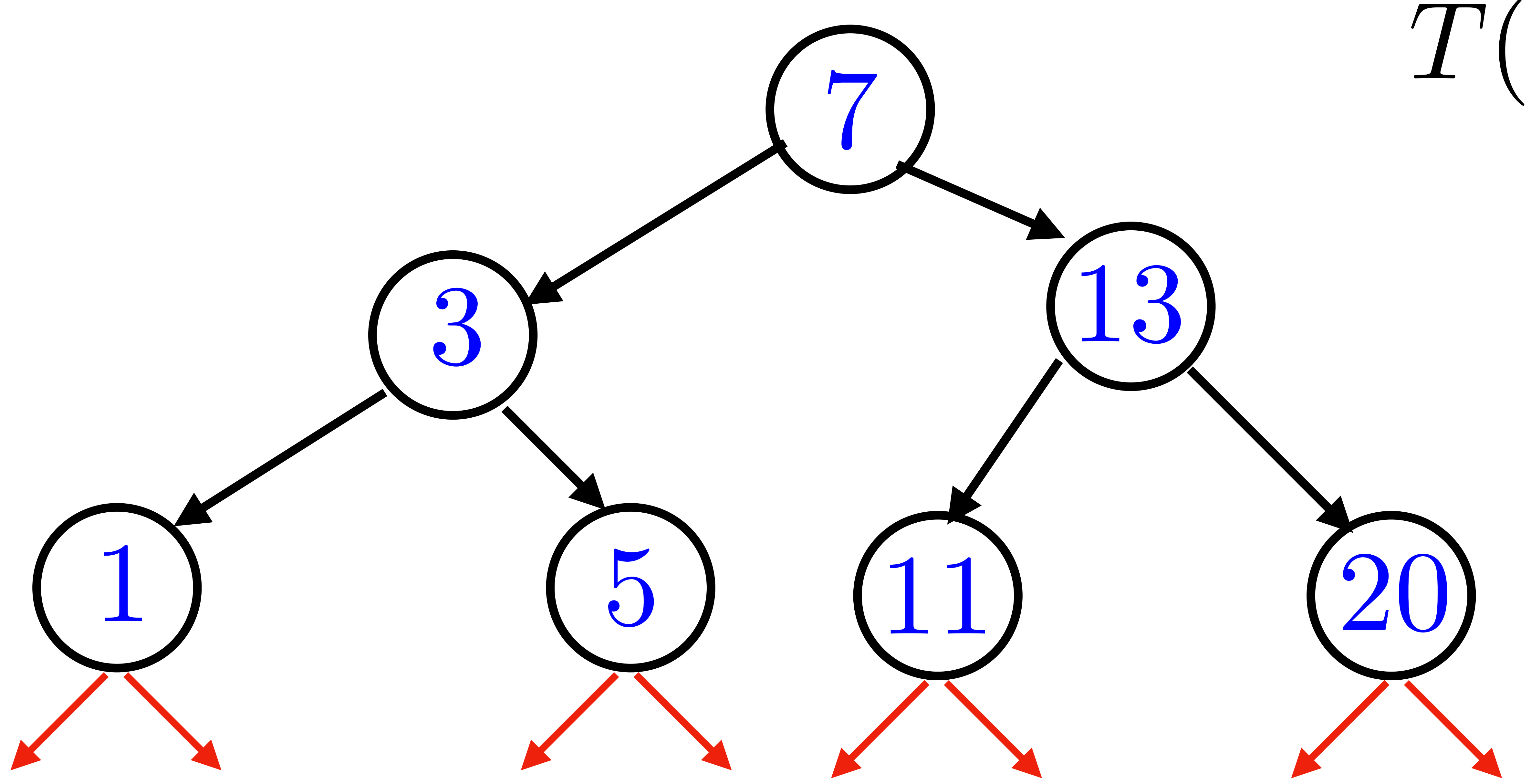


```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

$O(1)$

The complexity is $\Theta(n)$.

$T(n) \leq$



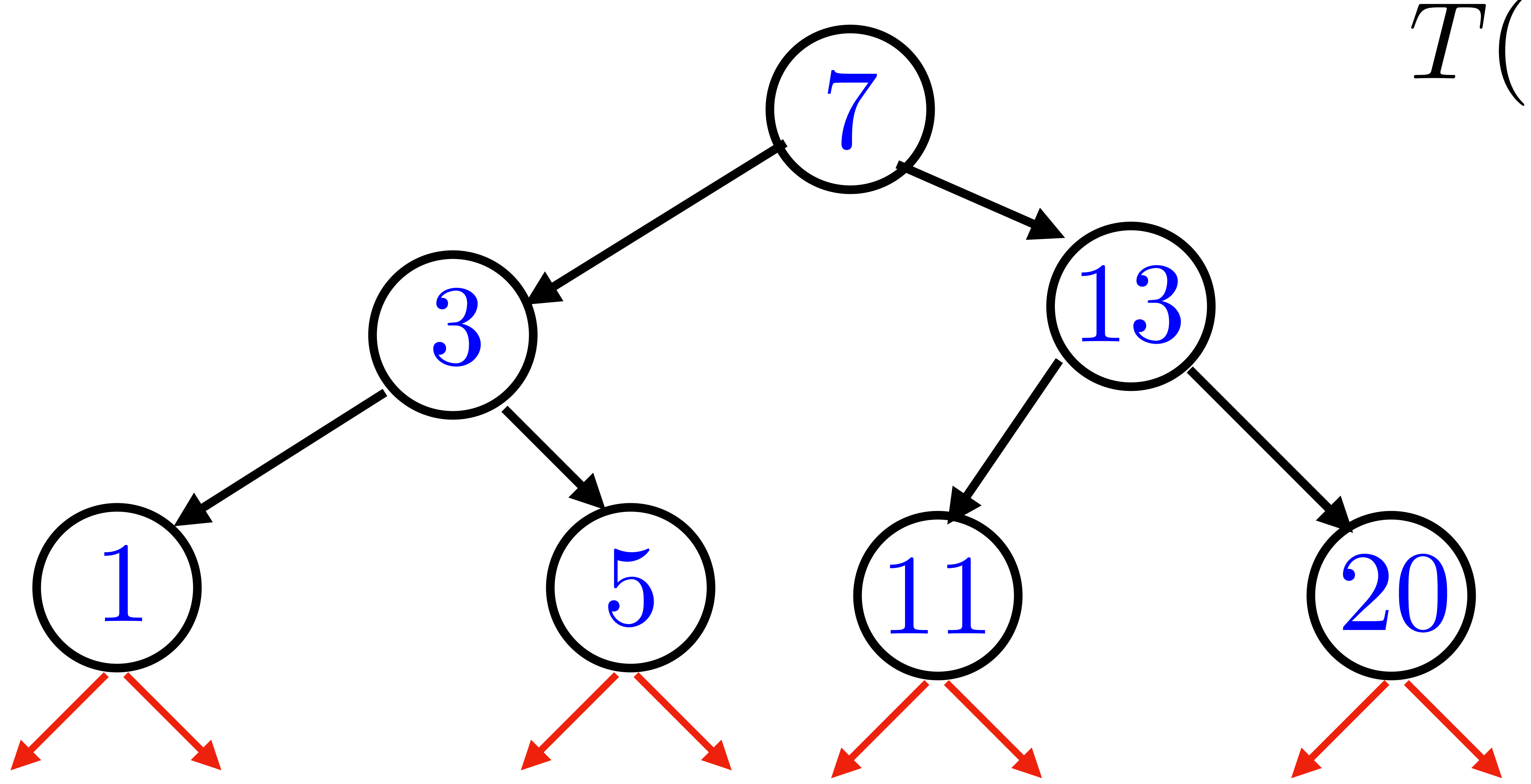
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

$O(1)$ (for the `if`/`return`)

$O(1)$ (for the `std::cout` line)

The complexity is $\Theta(n)$.

$T(n) \leq$



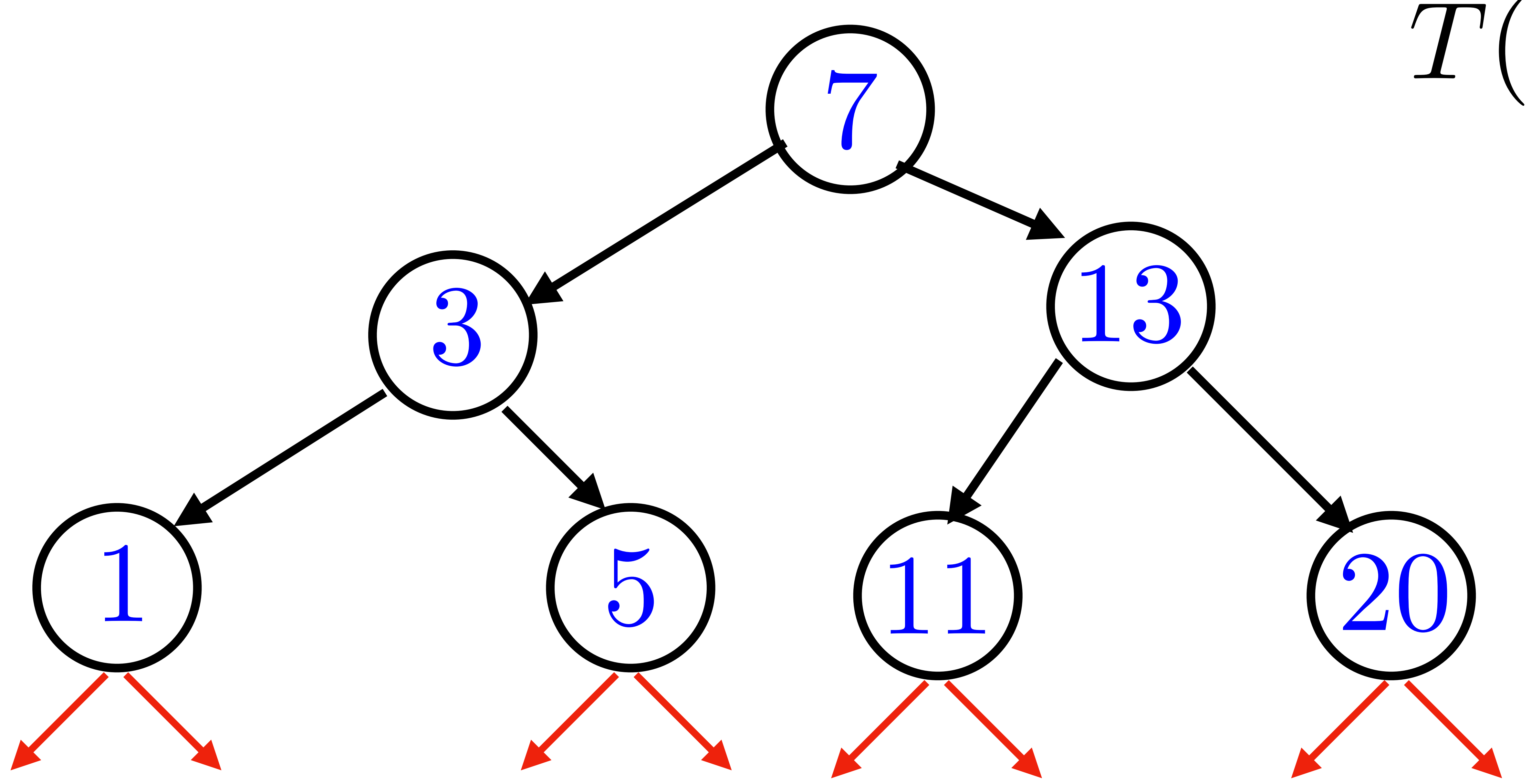
```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

$O(1)$

$T(k)$

$O(1)$

The complexity is $\Theta(n)$.

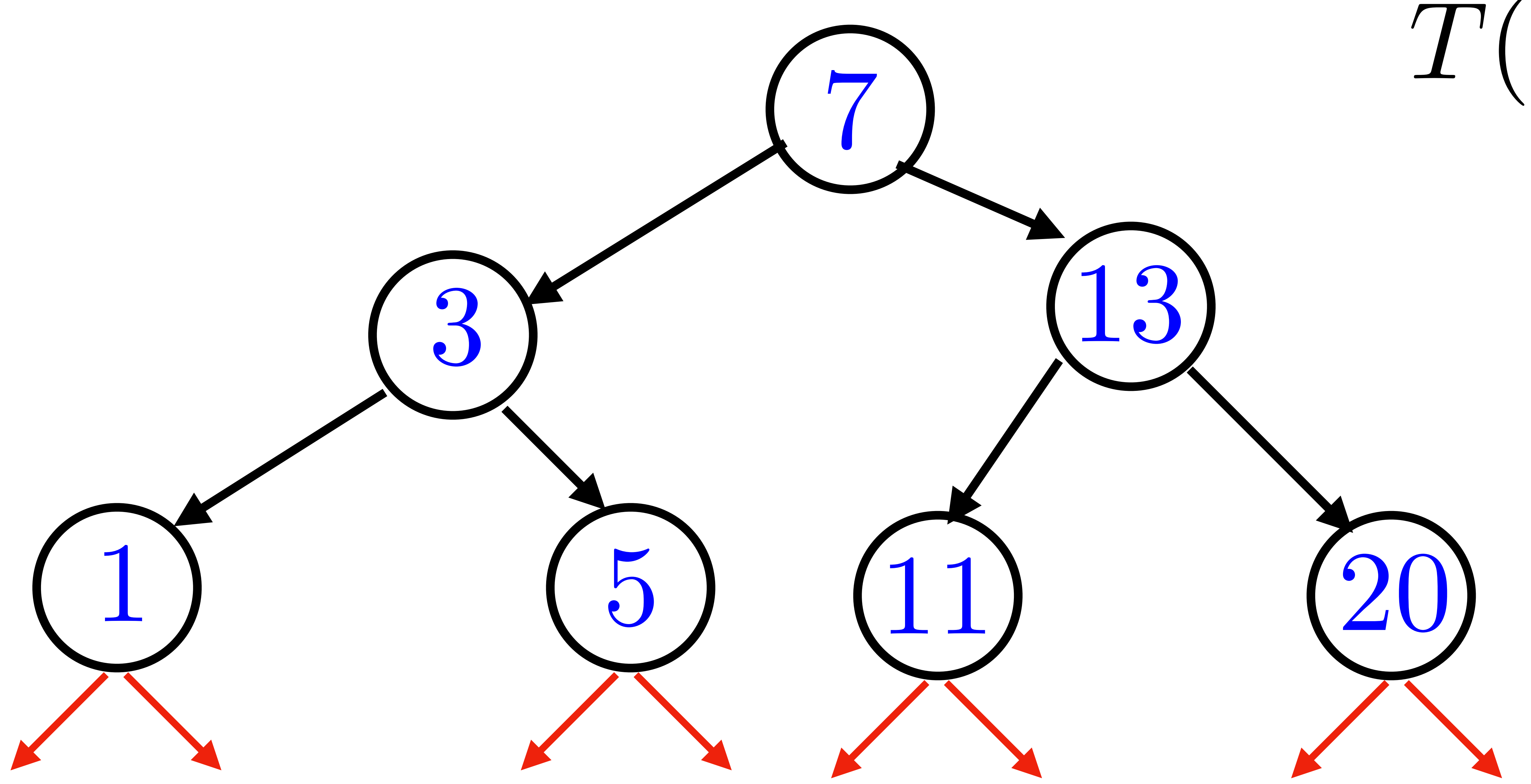


$T(n) \leq$

```
void print(Node* node){
    if(node == nullptr)          O(1)
        return;
    print(node->left);           T(k)
    std::cout << node->key << '\n';   O(1)
    print(node->right);          T(n-k-1)
}
```

The complexity is $\Theta(n)$.

$T(n) \leq$



```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

$O(1)$

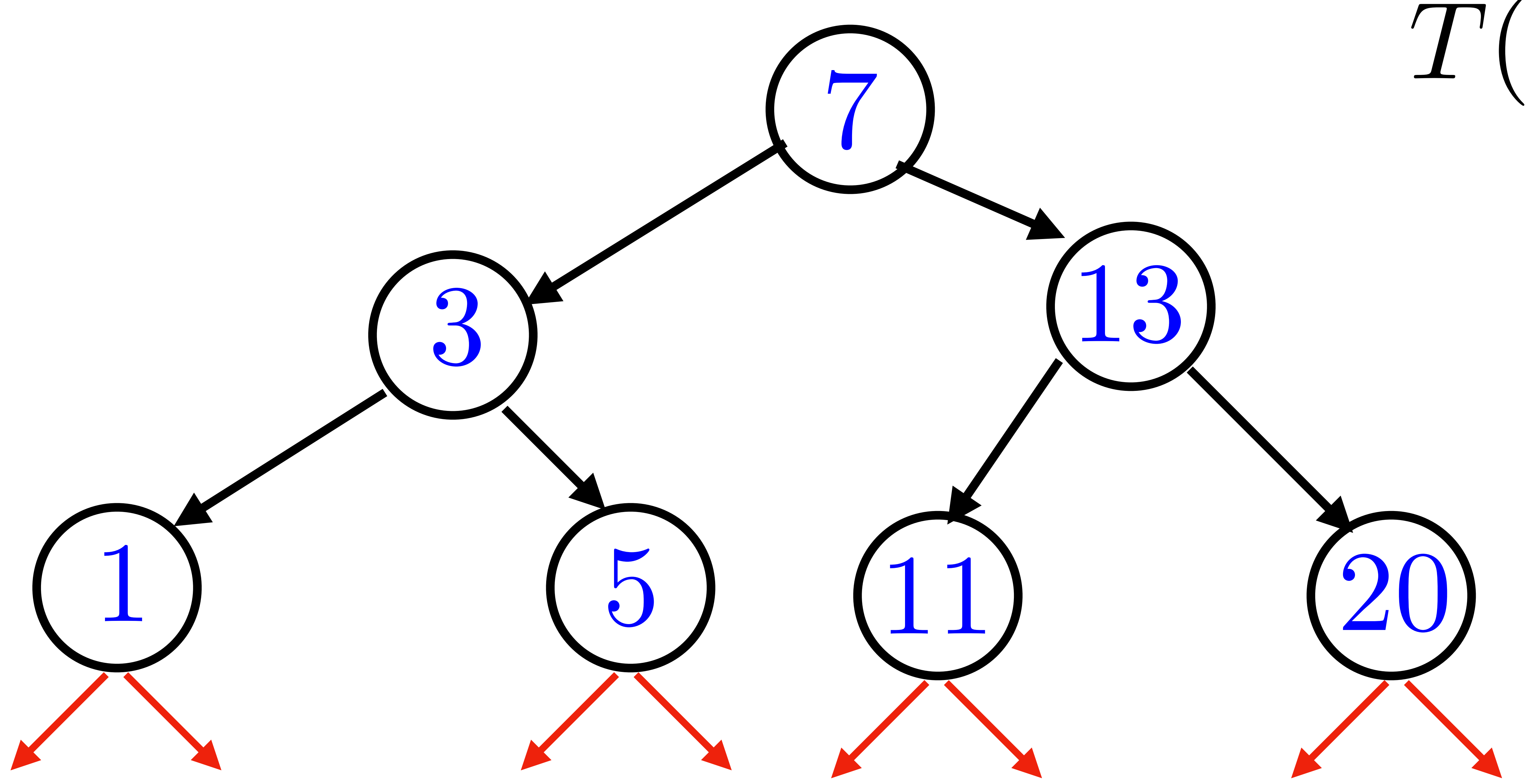
$T(k)$

$O(1)$

$T(n-k-1)$

The complexity is $\Theta(n)$.

$$T(n) \leq T(k) + T(n-k-1) + O(1)$$

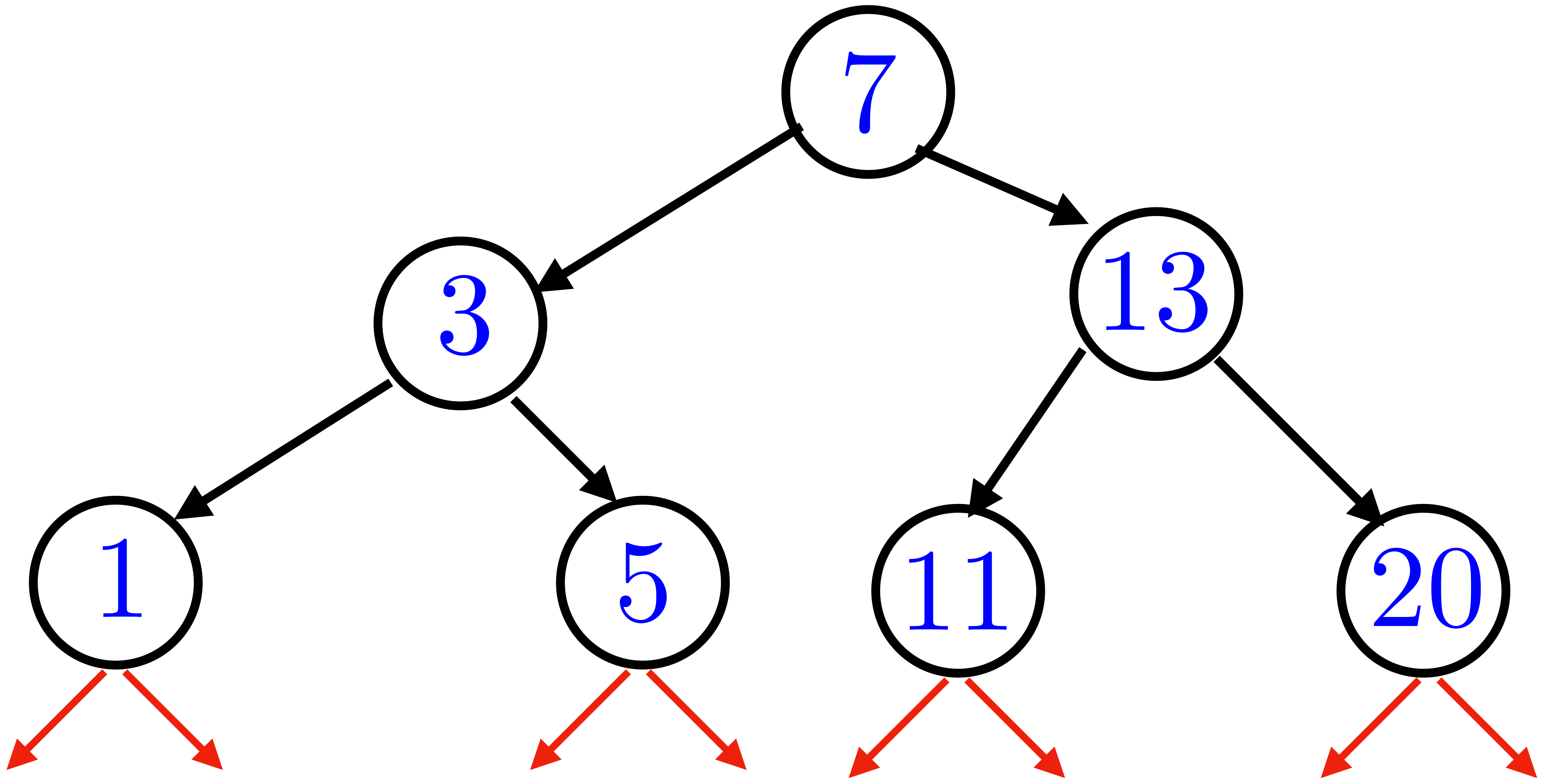


$$T(n) \le T(k) + T(n-k-1) + O(1)$$

Let's simplify the constants and say that $T(1) = 1$ and

$$T(n) = T(k) + T(n-k-1) + 1$$

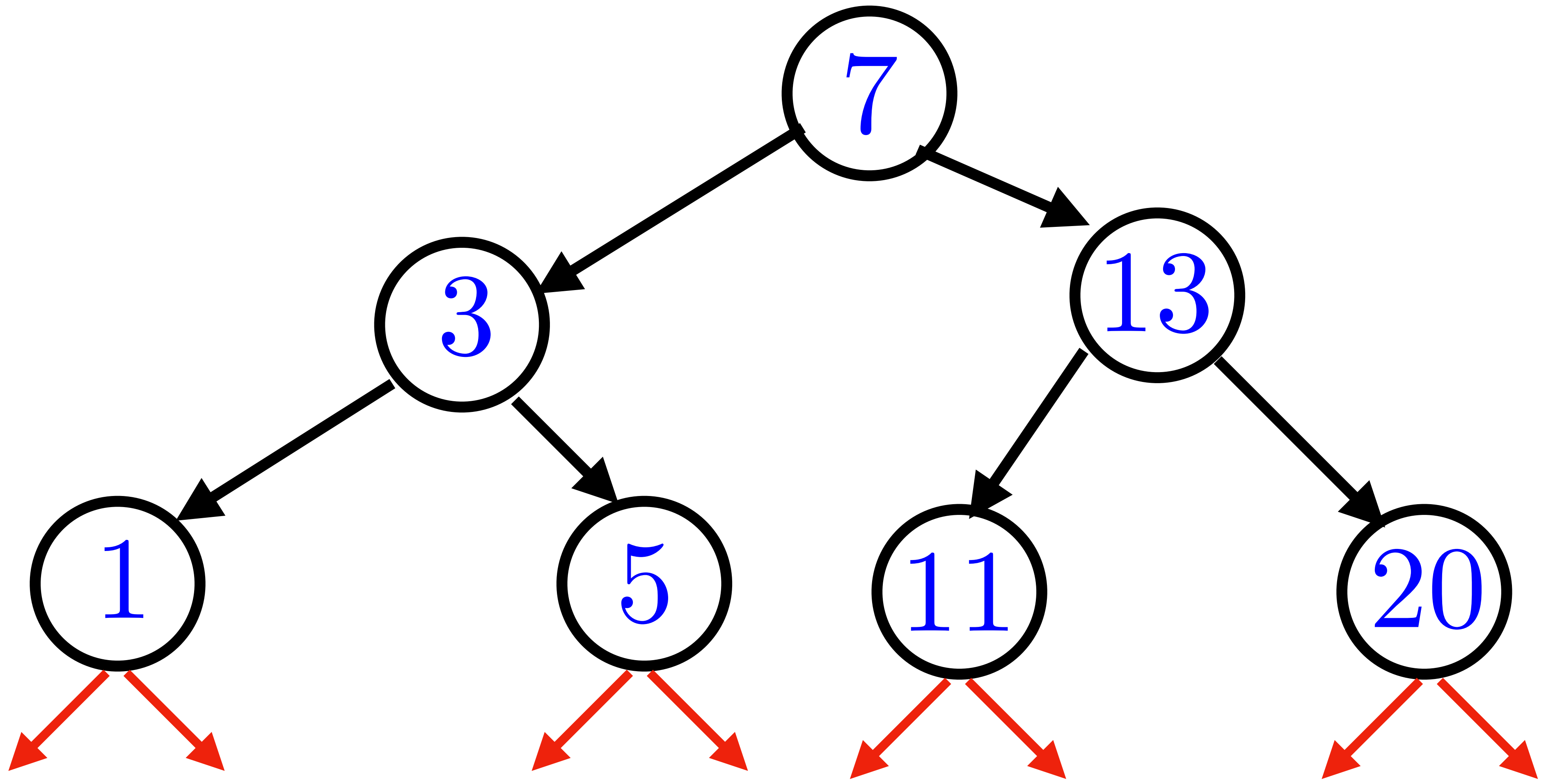
Then you can directly verify that $T(n) = n$.



Pre-order and Post-order traversal

```
void preorder(Node* node){
    if(node == nullptr)
        return;
    std::cout << node->key << '\n';
    preorder(node->left);
    preorder(node->right);
}
```

```
void postorder(Node* node){
    if(node == nullptr)
        return;
    postorder(node->left);
    postorder(node->right);
    std::cout << node->key << '\n';
}
```



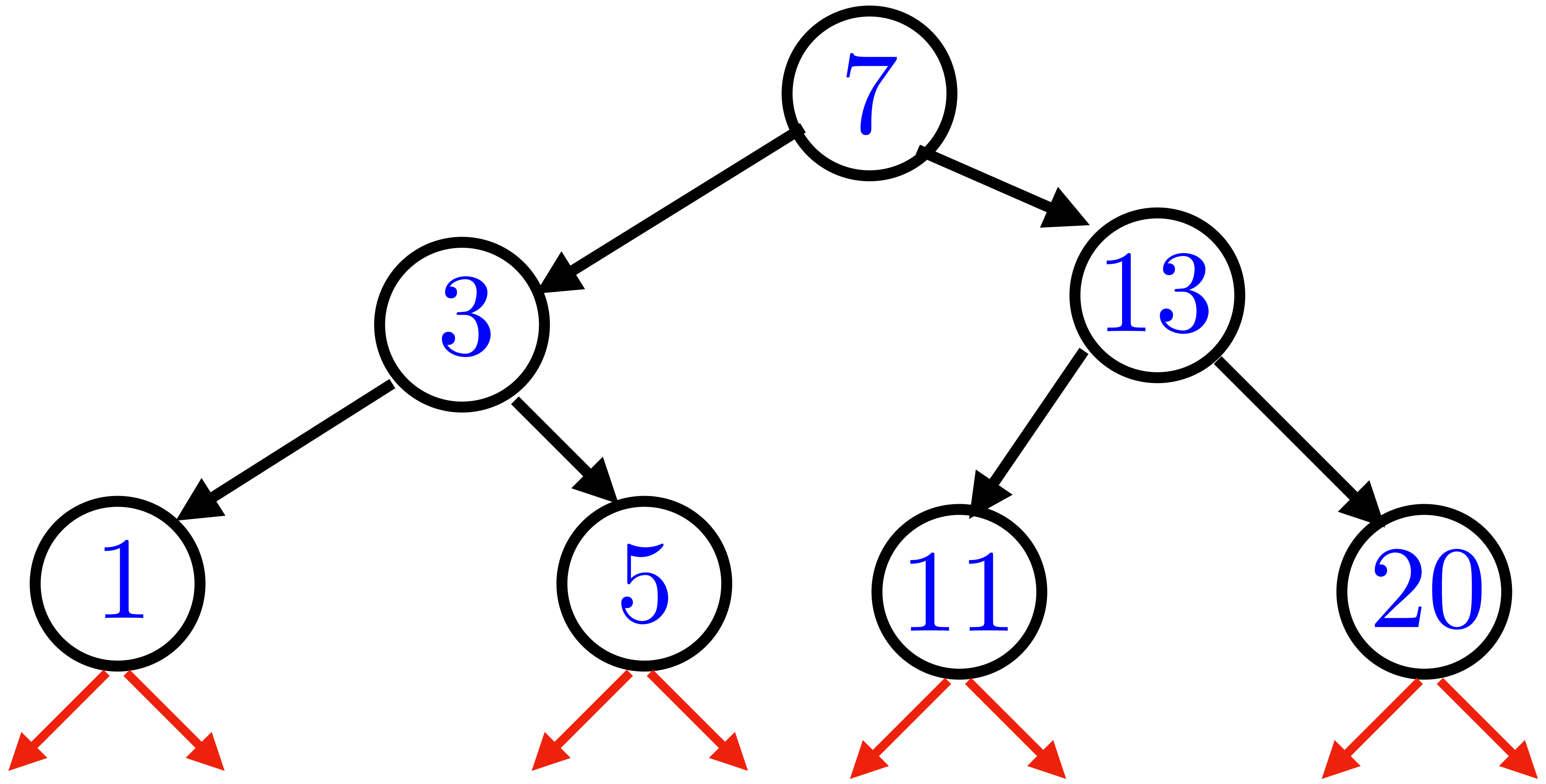
pre-order: $7, 3, 1, 5, 13, 11, 20$

post-order: $1, 5, 3, 11, 20, 13, 7$

BST Destructor

In the destructor for a BST we want to free the memory allocated to each node.

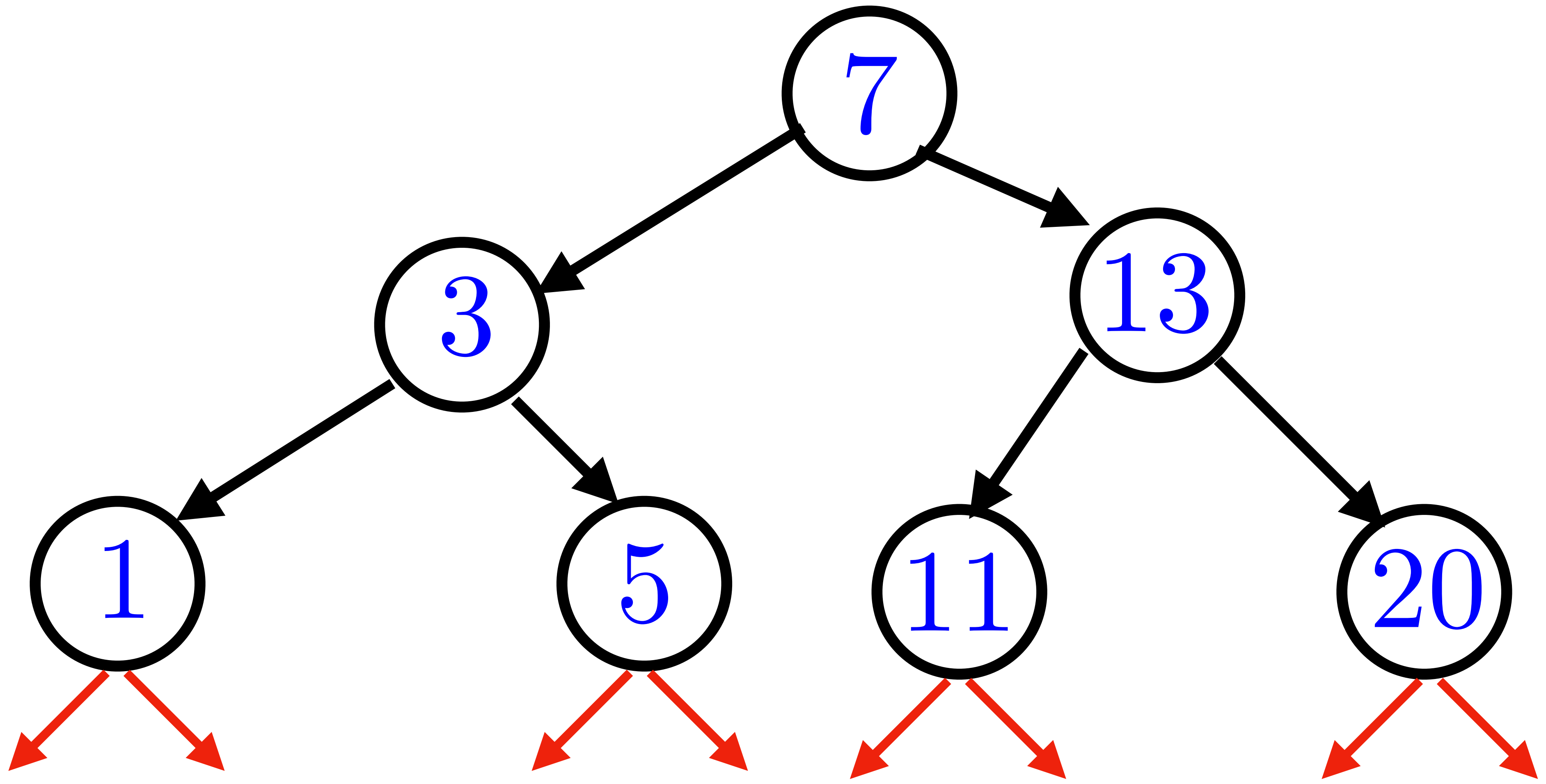
We traverse through the tree deleting pointers to the nodes.



BST Destructor

In the destructor for a BST we want to free the memory allocated to each node.

We traverse through the tree deleting pointers to the nodes.

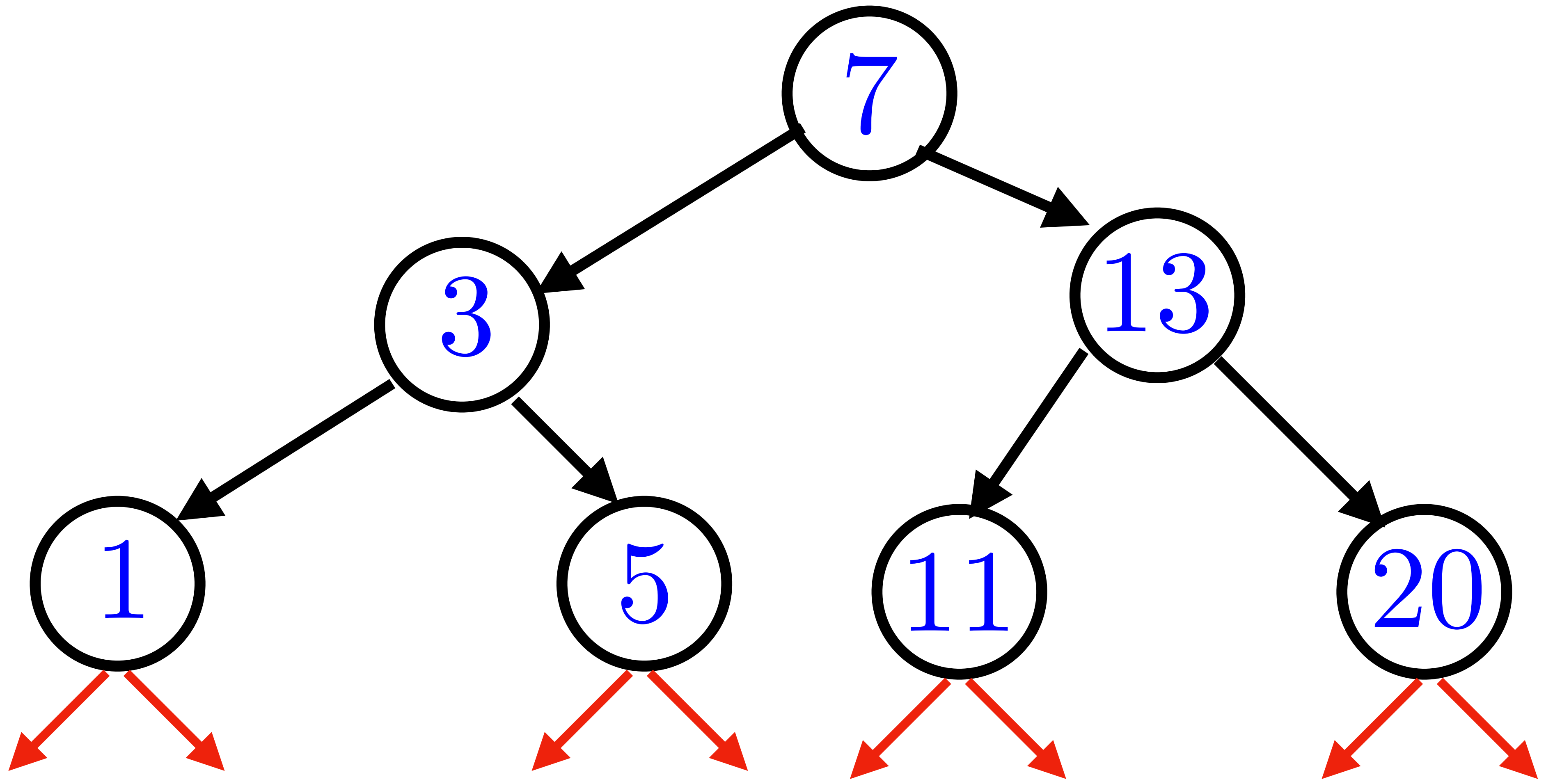


What kind of traversal should we use for this?

BST Destructor

In the destructor for a BST we want to free the memory allocated to each node.

We traverse through the tree deleting pointers to the nodes.



We don't want to delete the node with key 13 before deleting its children.

What kind of traversal should we use for this?

Height of a BST

Height of a BST

All of our operations have complexity $\Theta(h)$, where h is the height of the tree.

To understand the complexity of these algorithms we have to understand the height of a BST.

When we insert n elements what is the maximum height of the tree?

Worst case

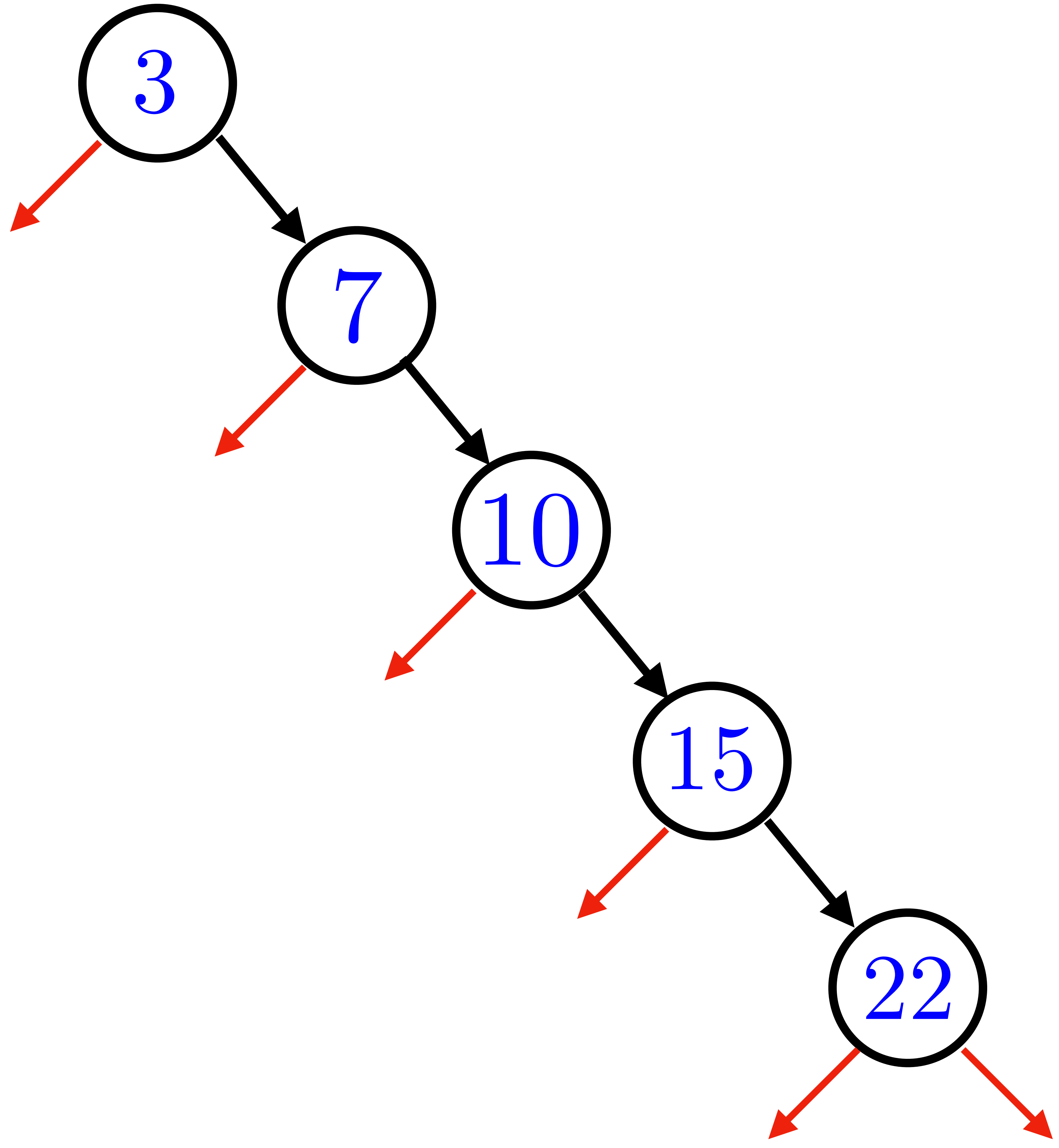
Say that we insert elements in the order 3, 7, 10, 15, 22.

In this case every vertex has at most one child.

The height is $n - 1$ which is as large as possible.

The worst case is when the elements are inserted in sorted order!

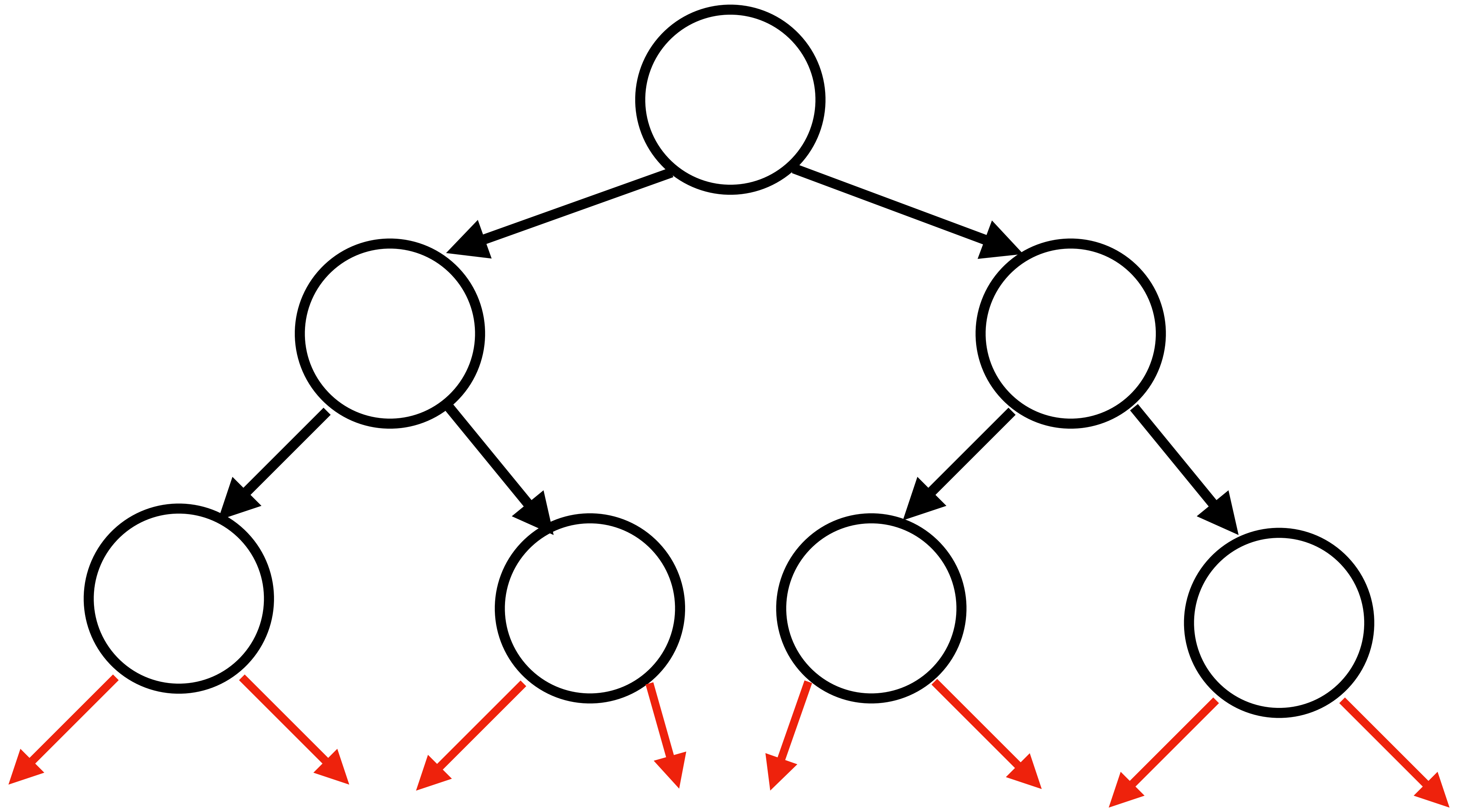
BSTs do not perform well in this scenario.



Best case

What is the best case height of the tree?

In the best case, for every vertex the height of its left and right subtrees is the same.



This is known as a full binary tree and is only possible if $n = 2^d - 1$.

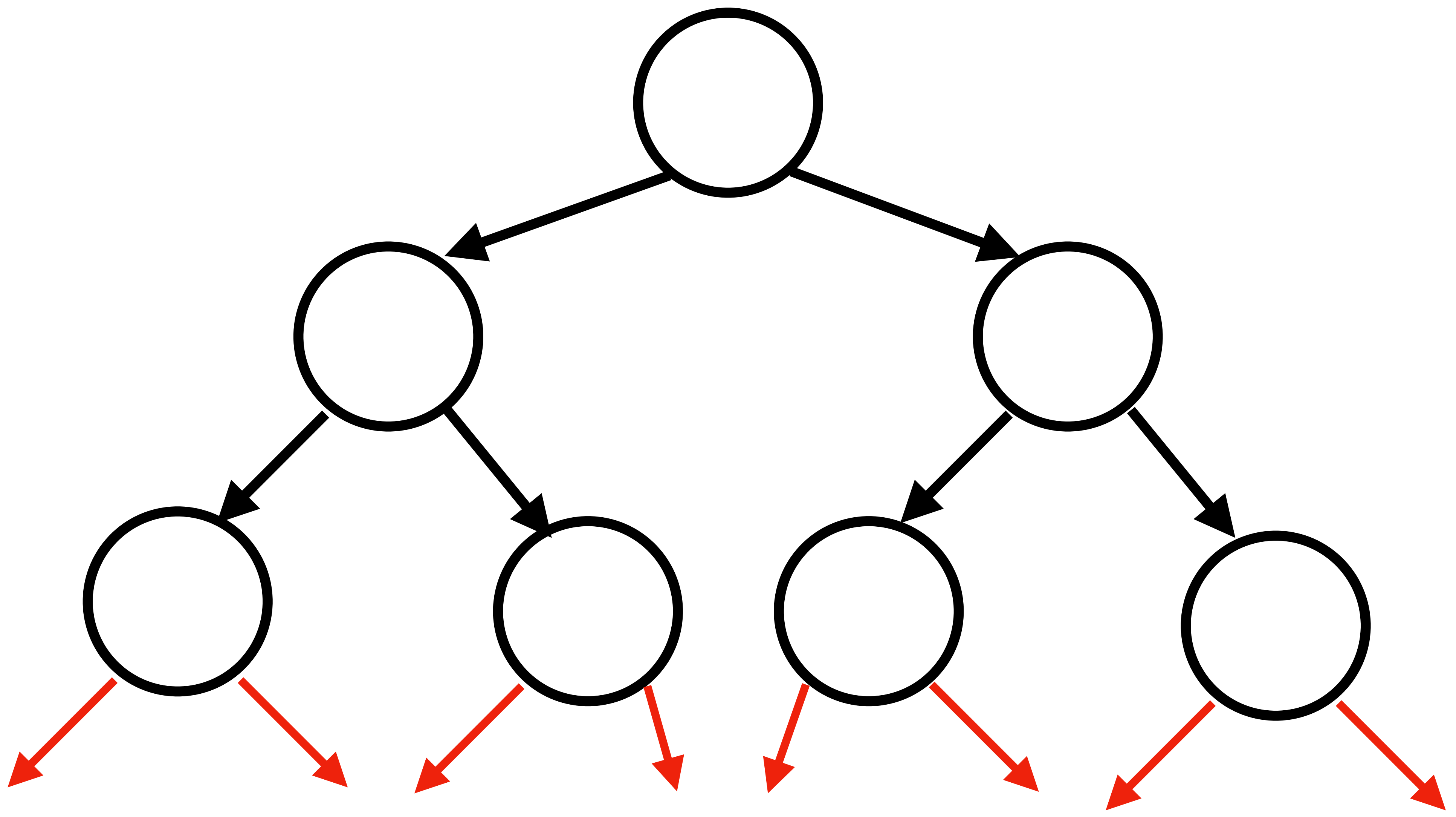
We always have $n \leq 2^{h+1} - 1$ and so $h \geq \log(n+1) - 1$.

Our operations will take time at least $\Omega(\log n)$.

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.

In what order should we insert these keys in to obtain a full binary tree?

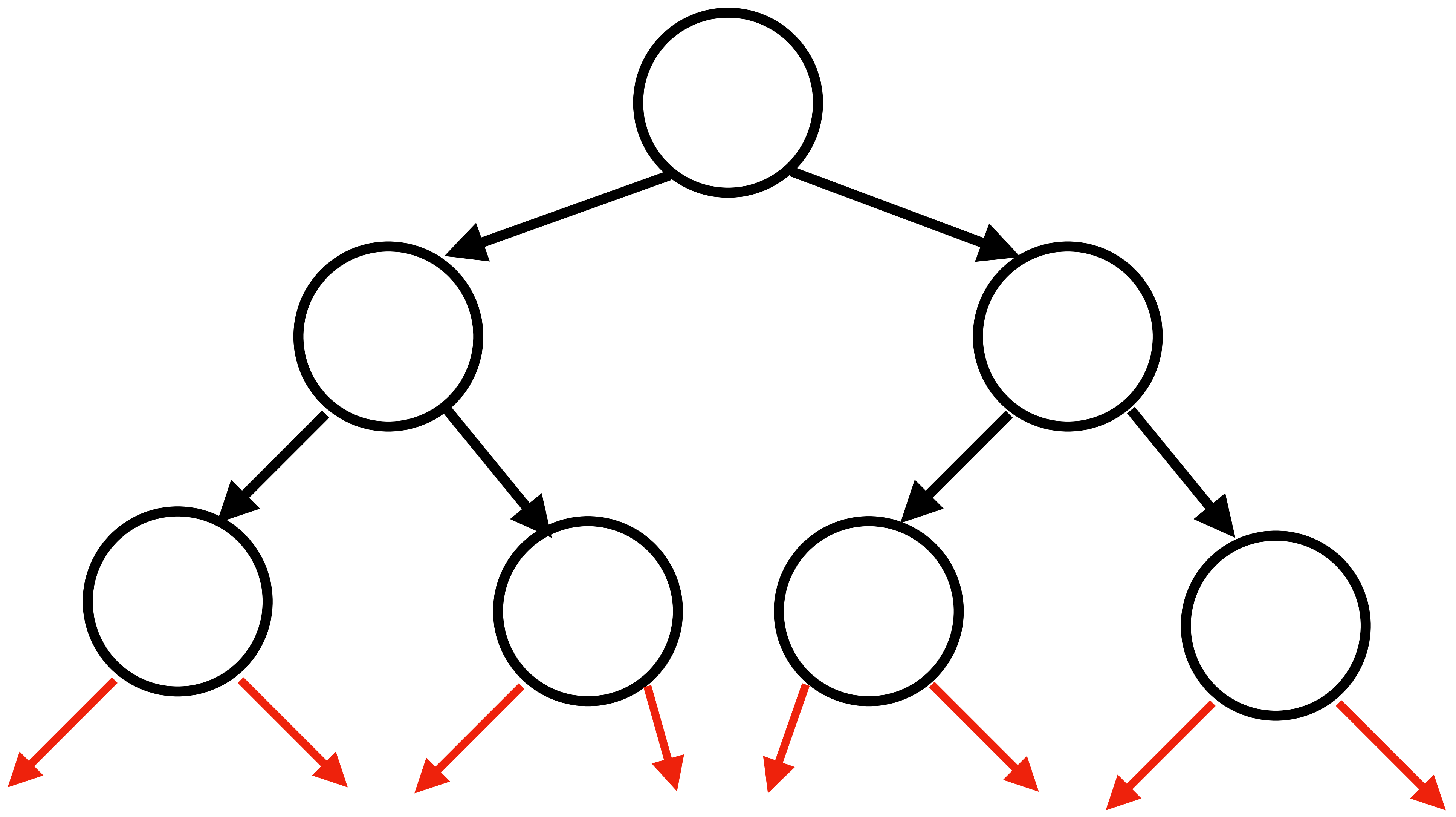


Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.

In what order should we insert these keys in to obtain a full binary tree?

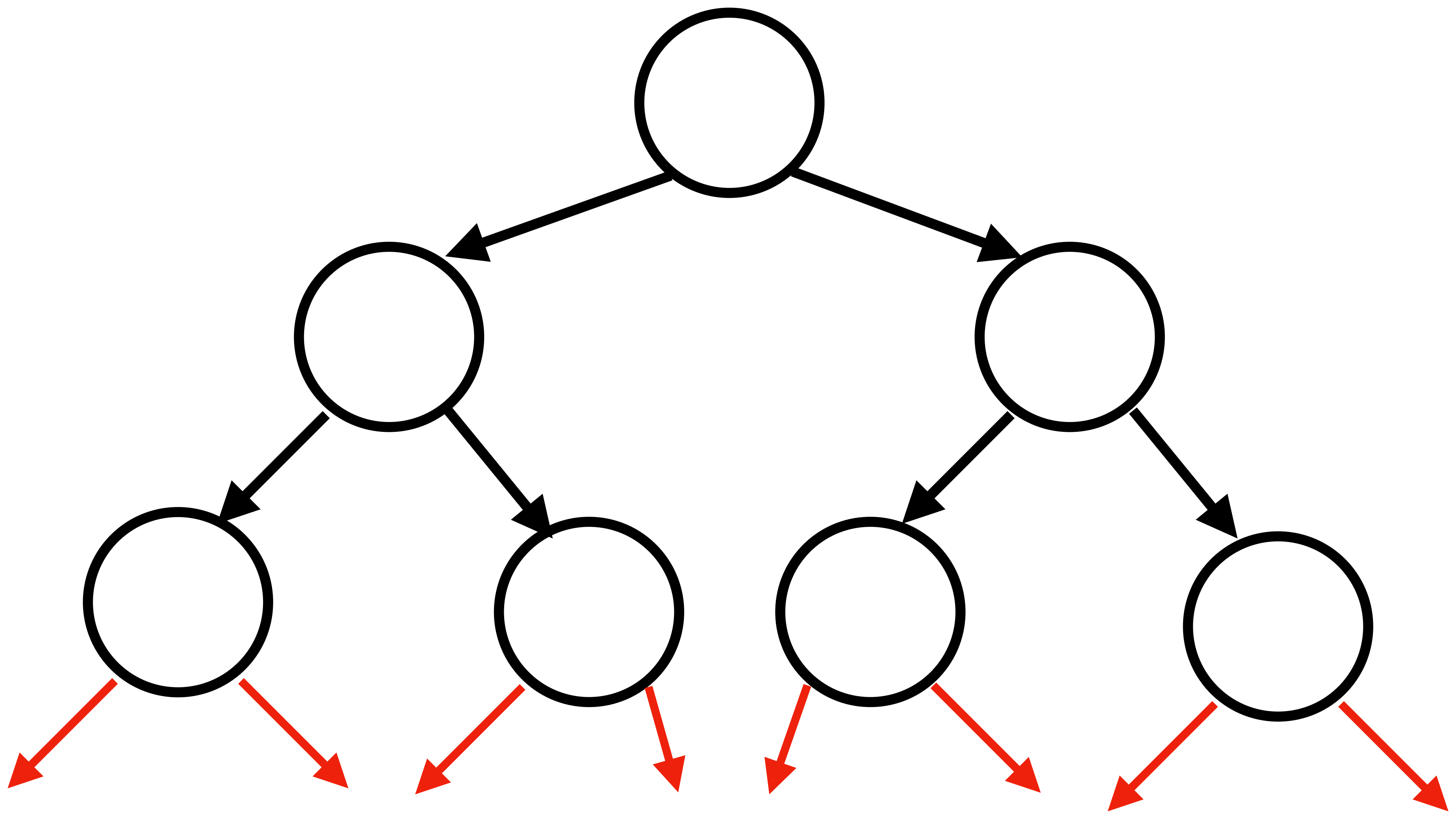
In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.



Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.

In what order should we insert these keys in to obtain a full binary tree?



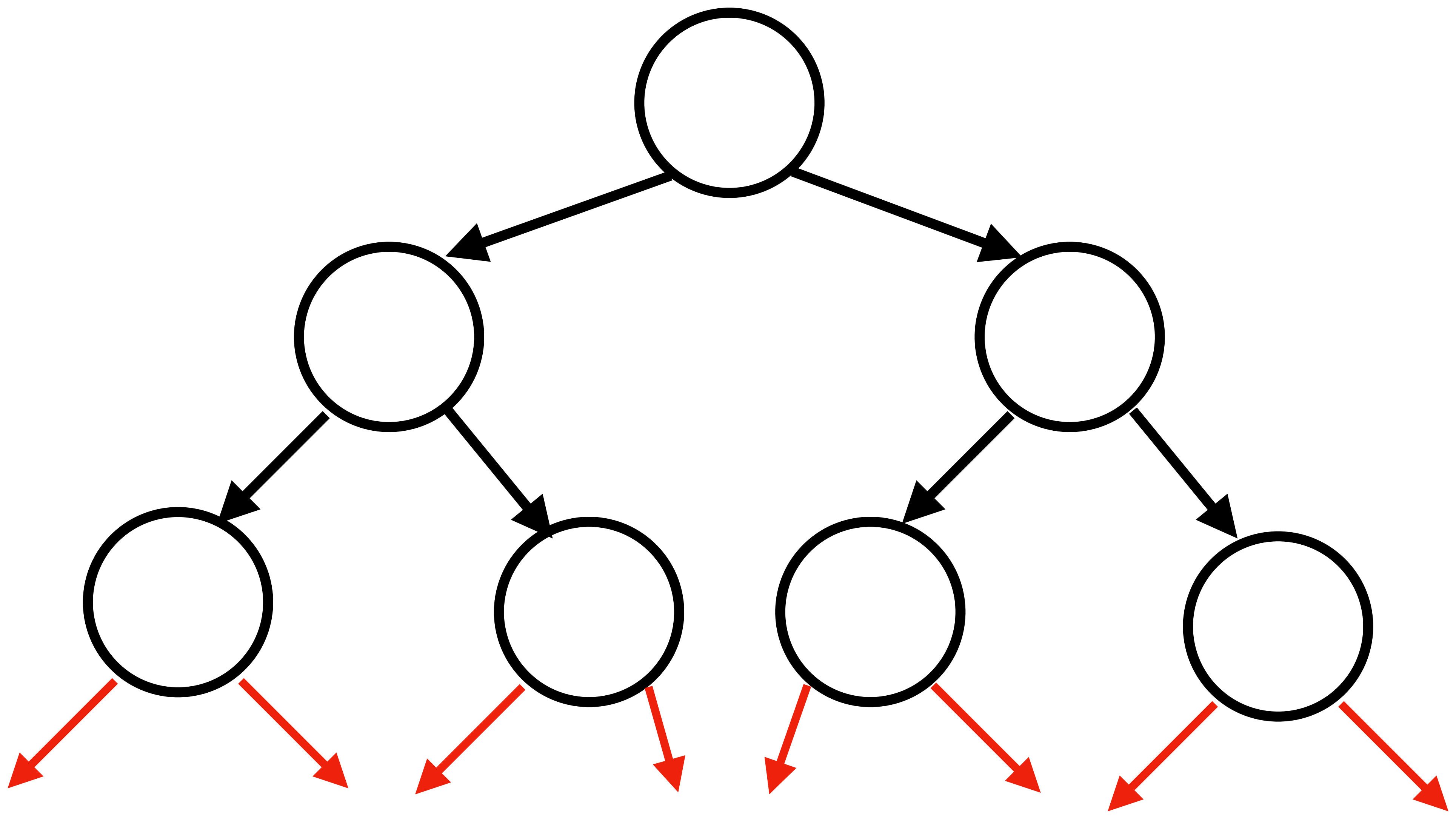
In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

If the root has key x how many nodes will be in its left subtree?

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.

In what order should we insert these keys in to obtain a full binary tree?



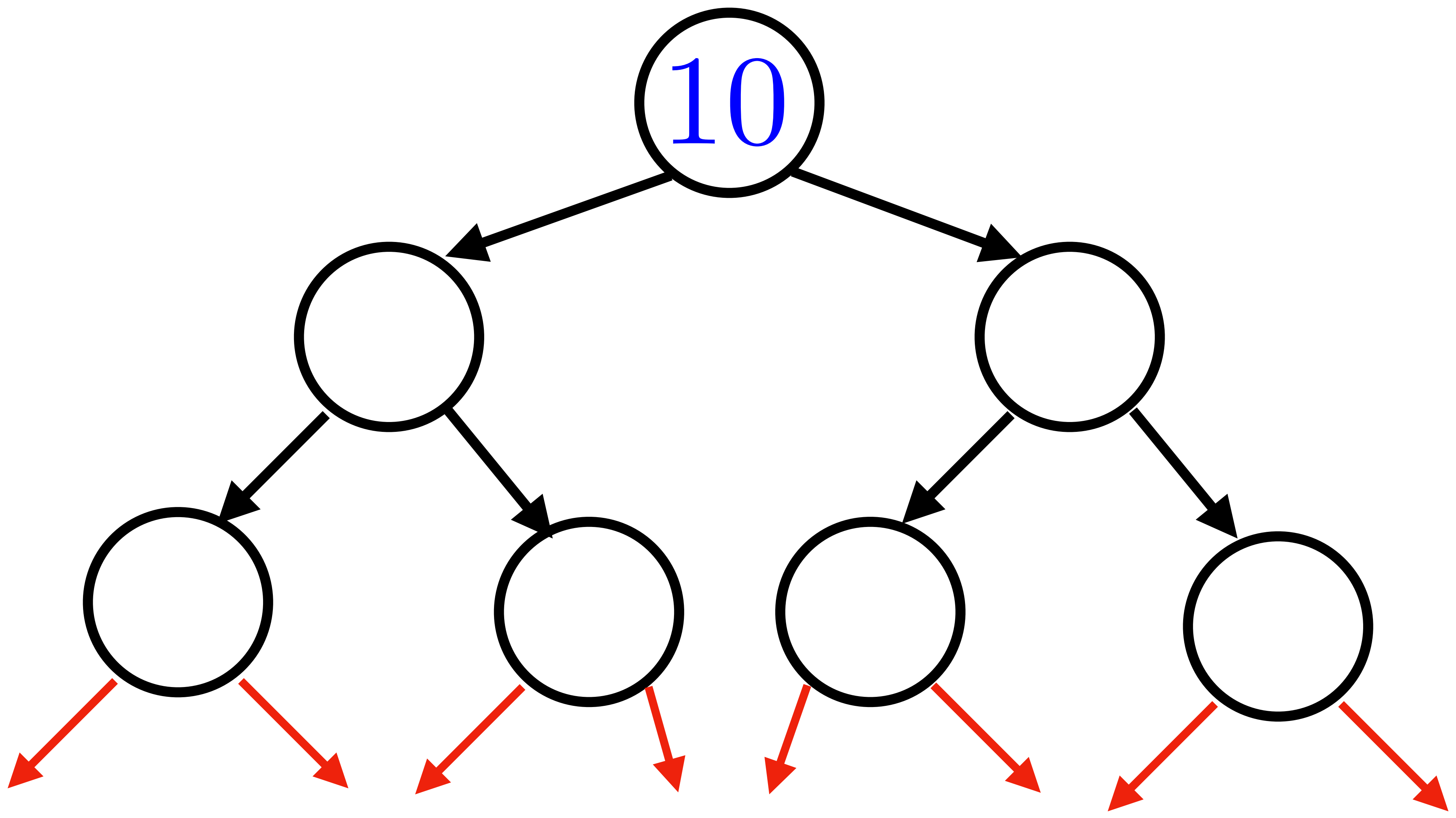
In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

If the root has key x how many nodes will be in its left subtree?

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.

In what order should we insert these keys in to obtain a full binary tree?

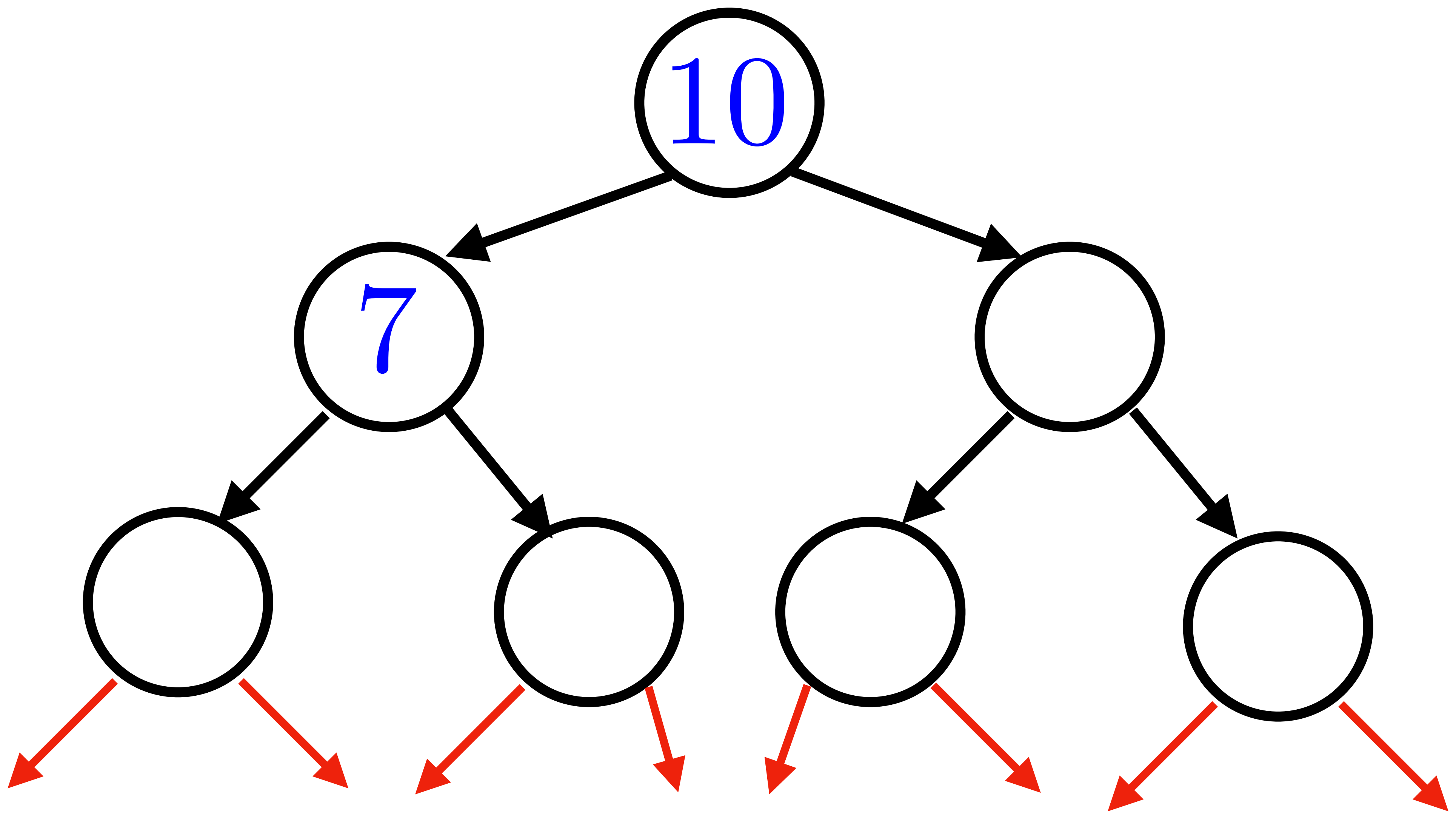


In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

If the root has key x how many nodes will be in its left subtree?

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.



In what order should we insert these keys in to obtain a full binary tree?

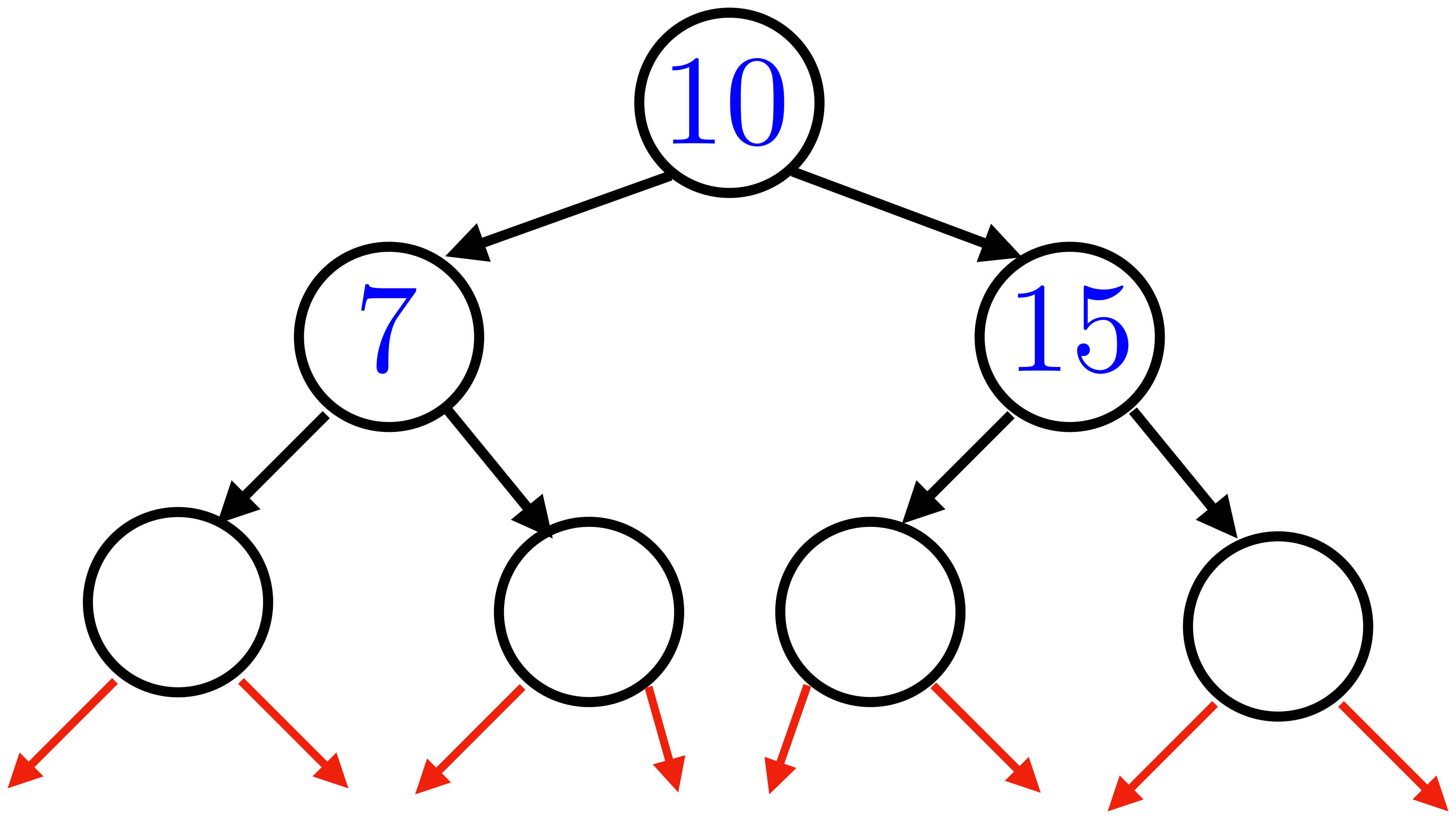
In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

If the root has key x how many nodes will be in its left subtree?

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.

In what order should we insert these keys in to obtain a full binary tree?

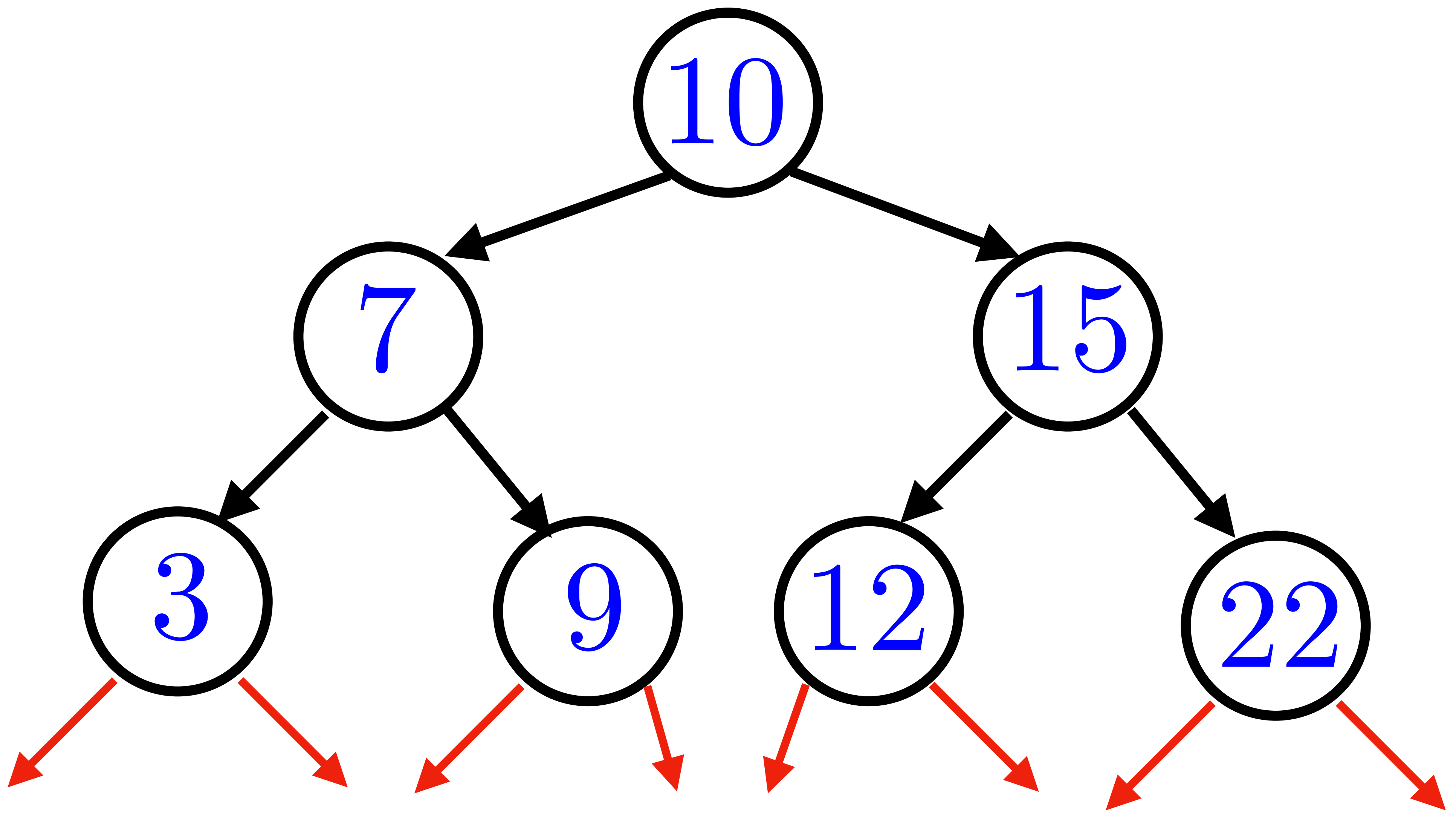


In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

If the root has key x how many nodes will be in its left subtree?

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.



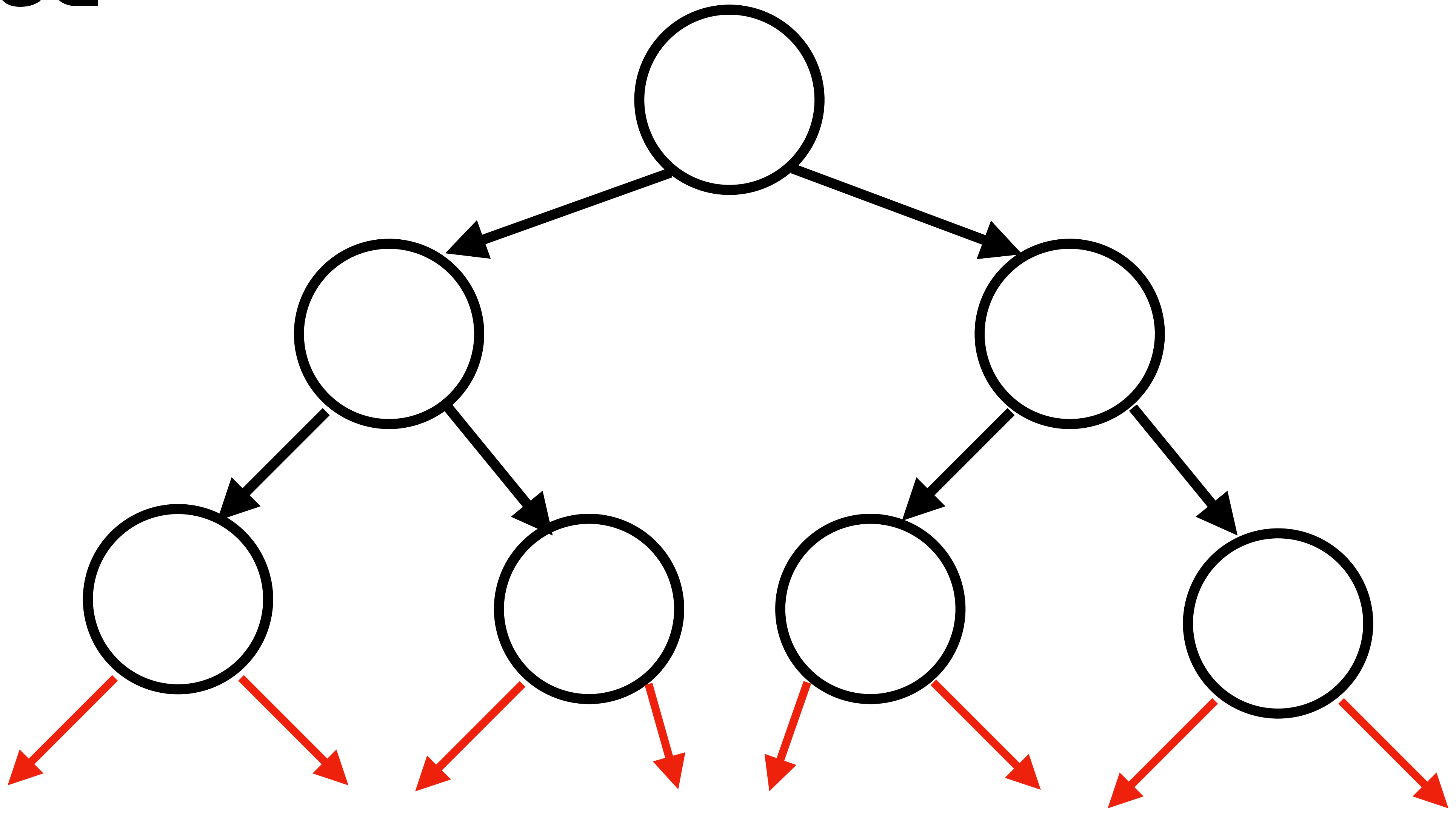
In what order should we insert these keys in to obtain a full binary tree?

In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

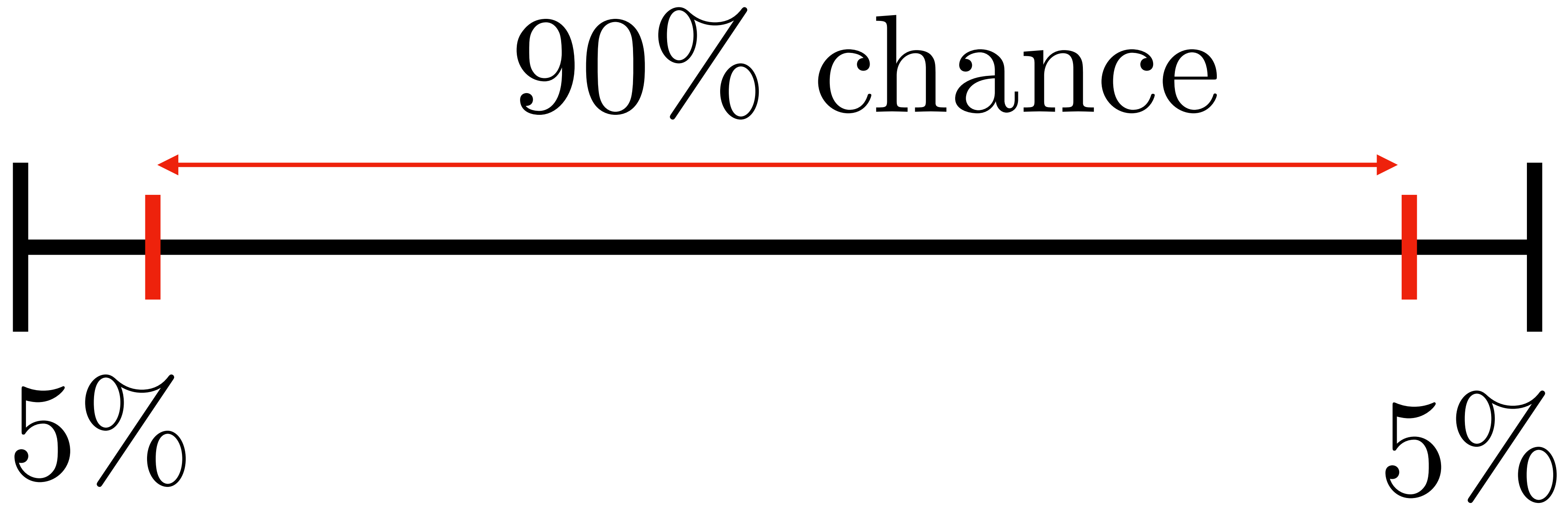
If the root has key x how many nodes will be in its left subtree?

Random case

If the keys come in a random order, the expected height of a BST is $O(\log n)$.



Intuition: we do not expect the first key to be the median, but with good probability it will be near the middle.



See Theorem 12.4 of [CLRS] for a proof.

AVL Trees

Balanced Binary Trees

It is desirable to maintain a tree height of $O(\log n)$ when the tree has n keys.

We now look at a way of actively ensuring this property.

Whenever we insert or remove a key, if the tree becomes too unbalanced we change the structure to fix it.

There are several (related) techniques to do this: AVL trees, 2-3 trees, AA trees, red-black trees.

These achieve $O(\log n)$ worst-case time for all our operations.

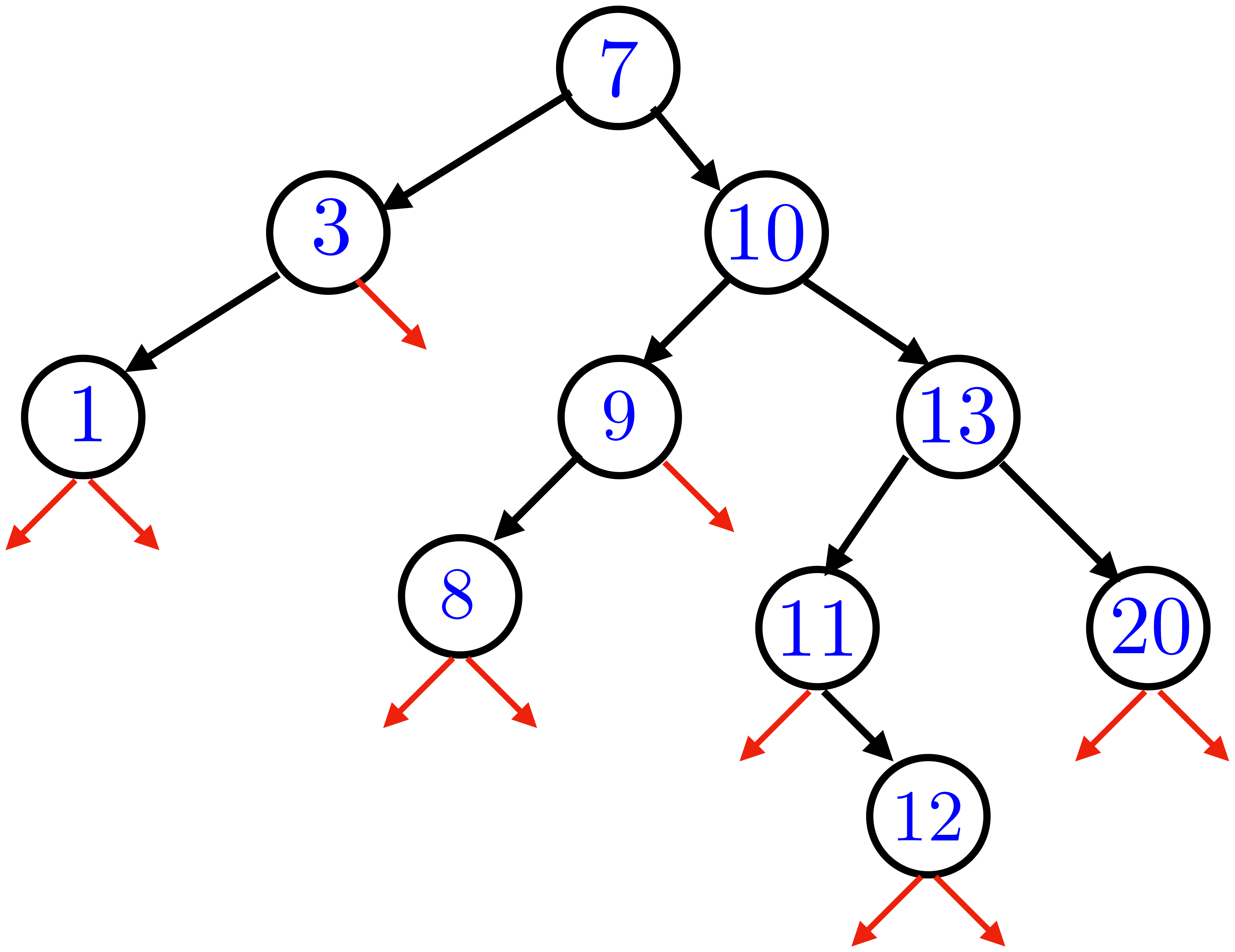
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



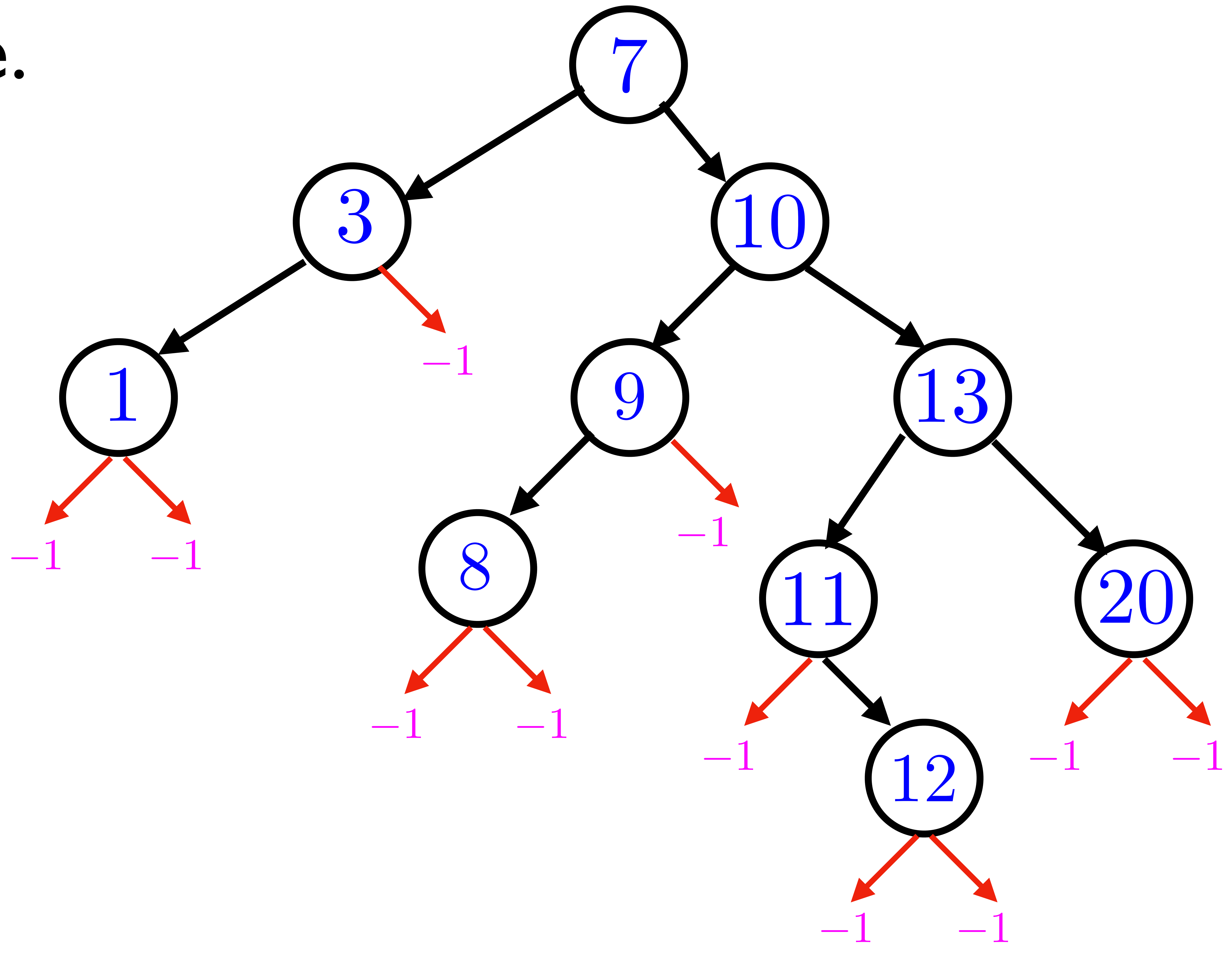
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



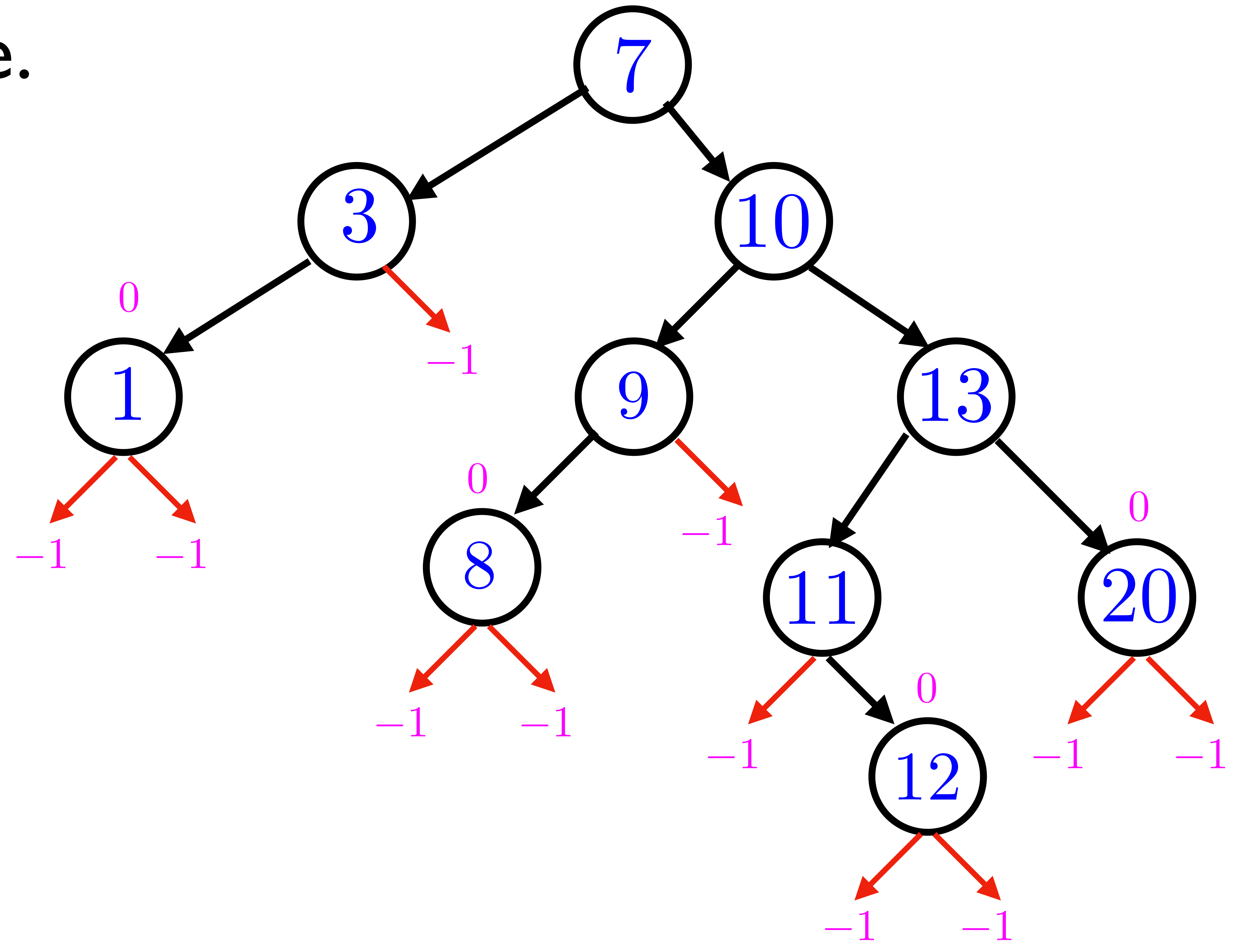
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



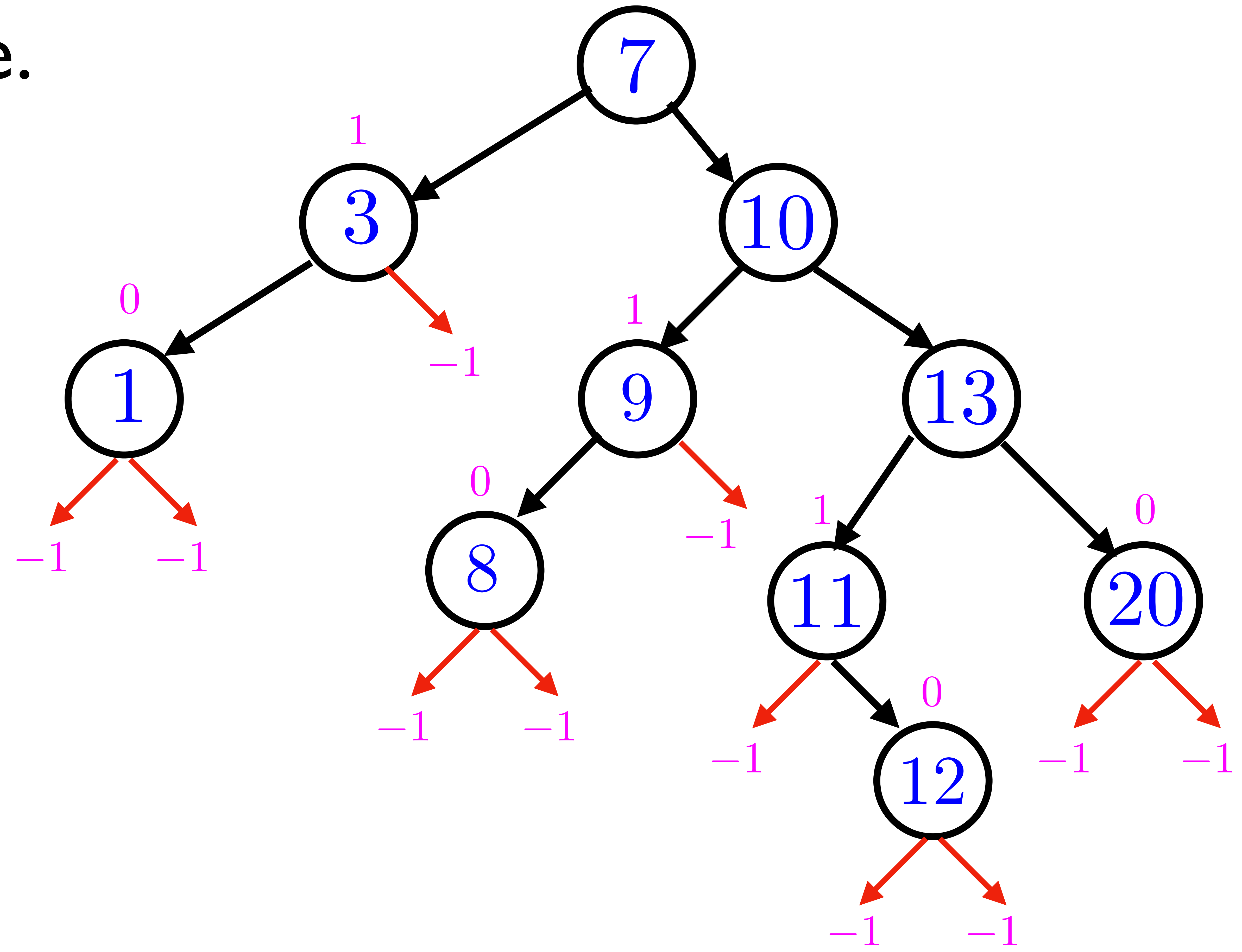
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



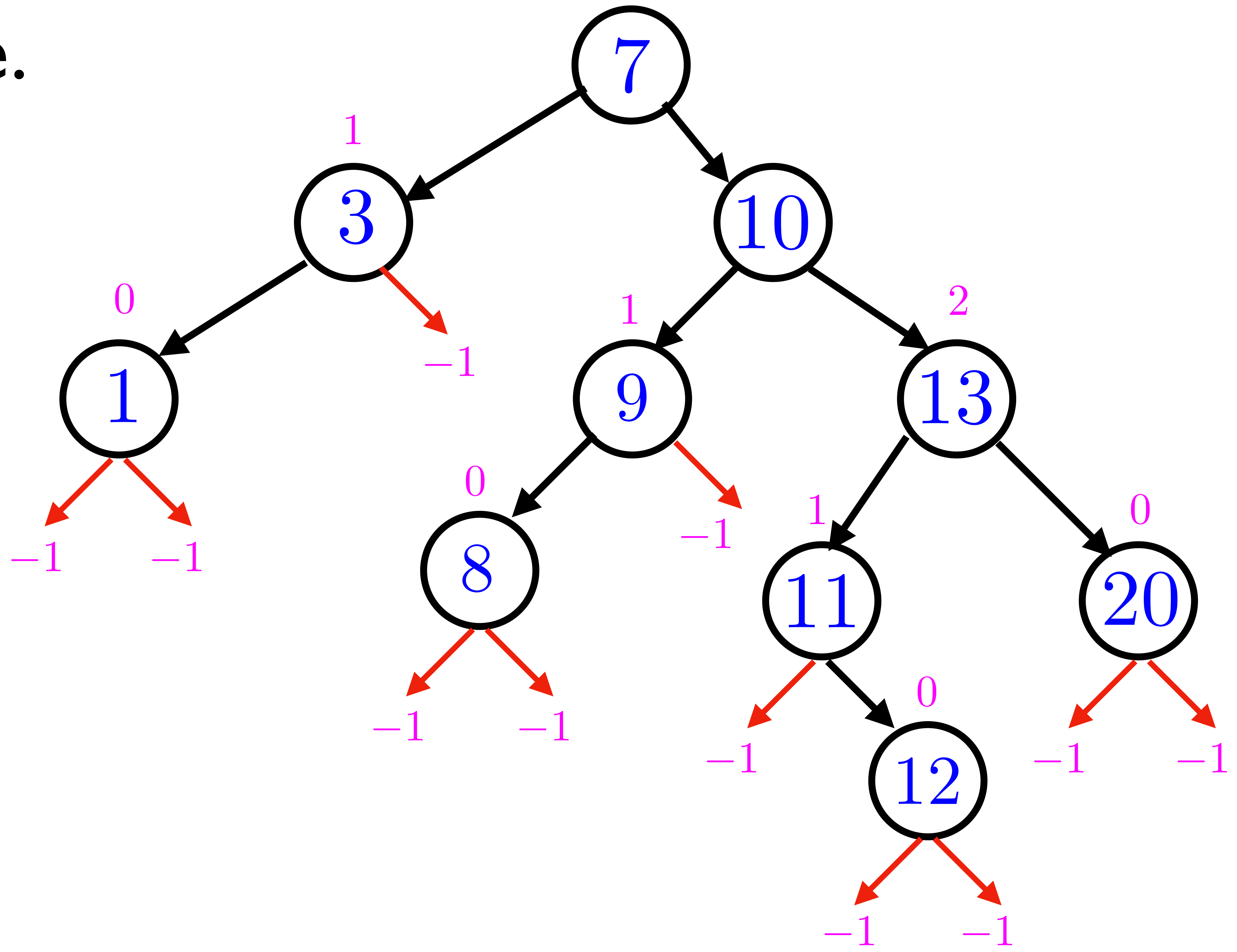
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



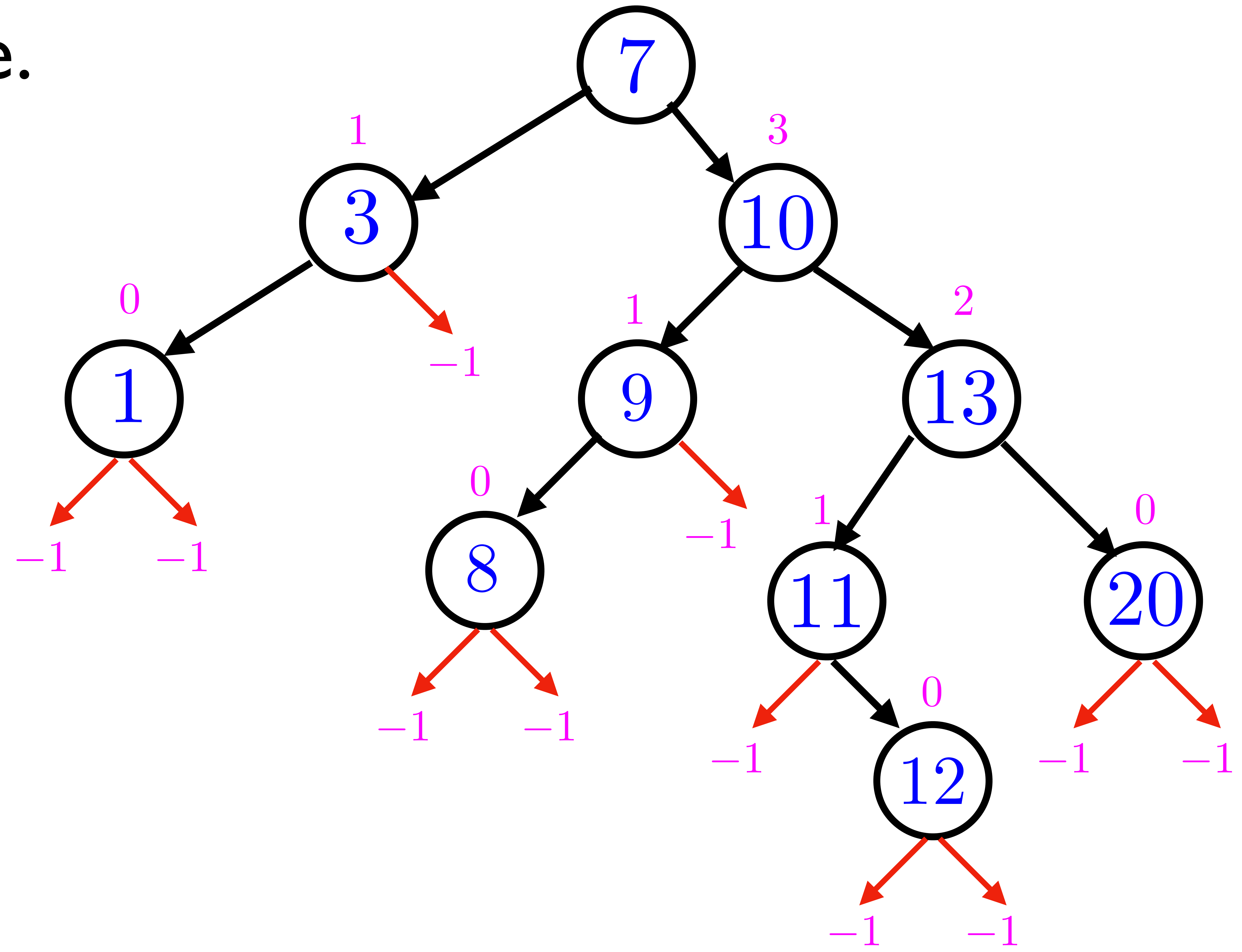
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



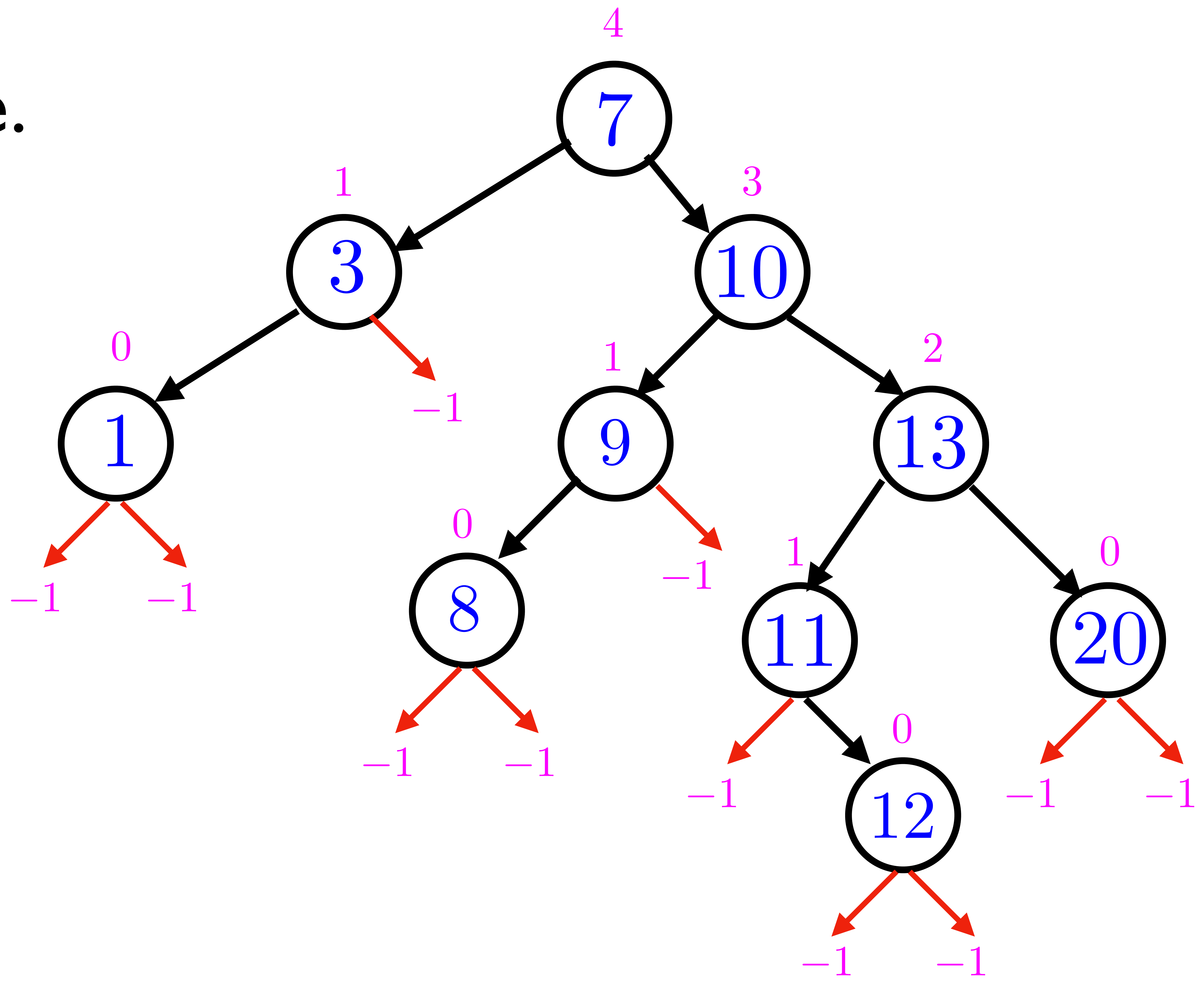
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



AVL trees

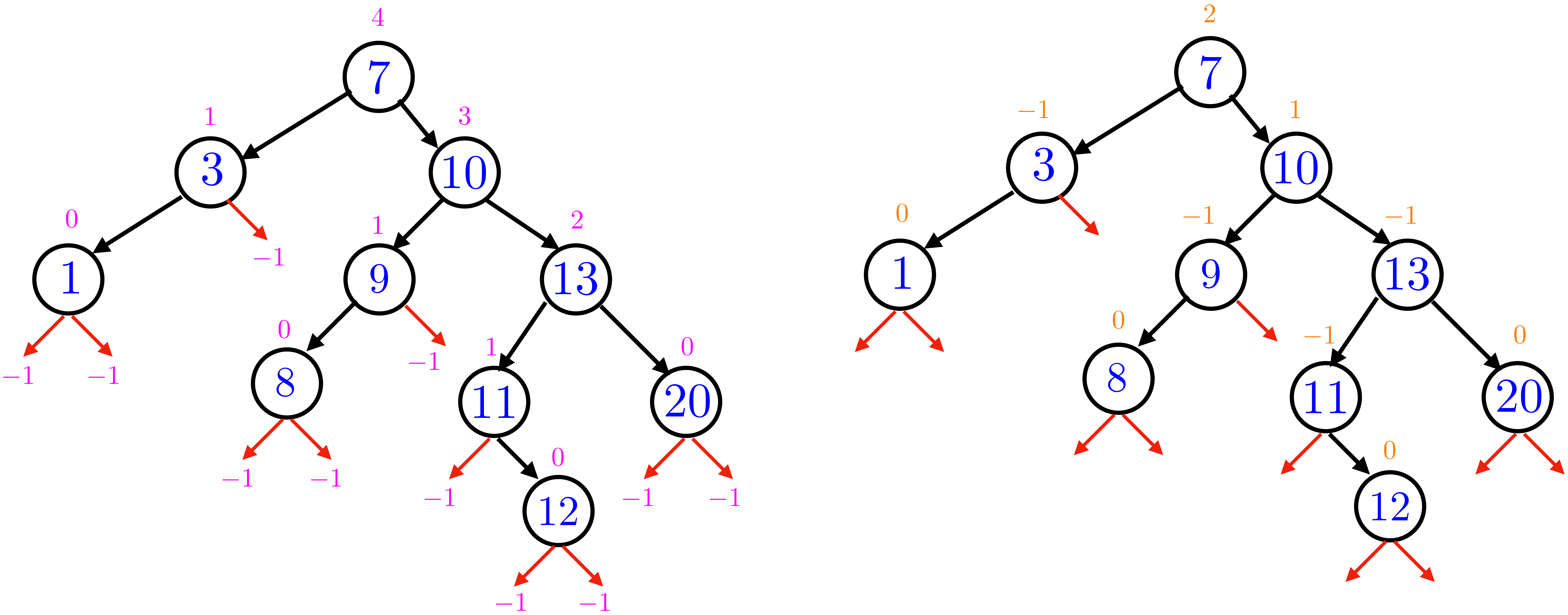
An AVL tree maintains the property that left and right children differ in height by at most one.

Call the balance factor of a node to be the height of its right subtree minus the height of its left subtree.

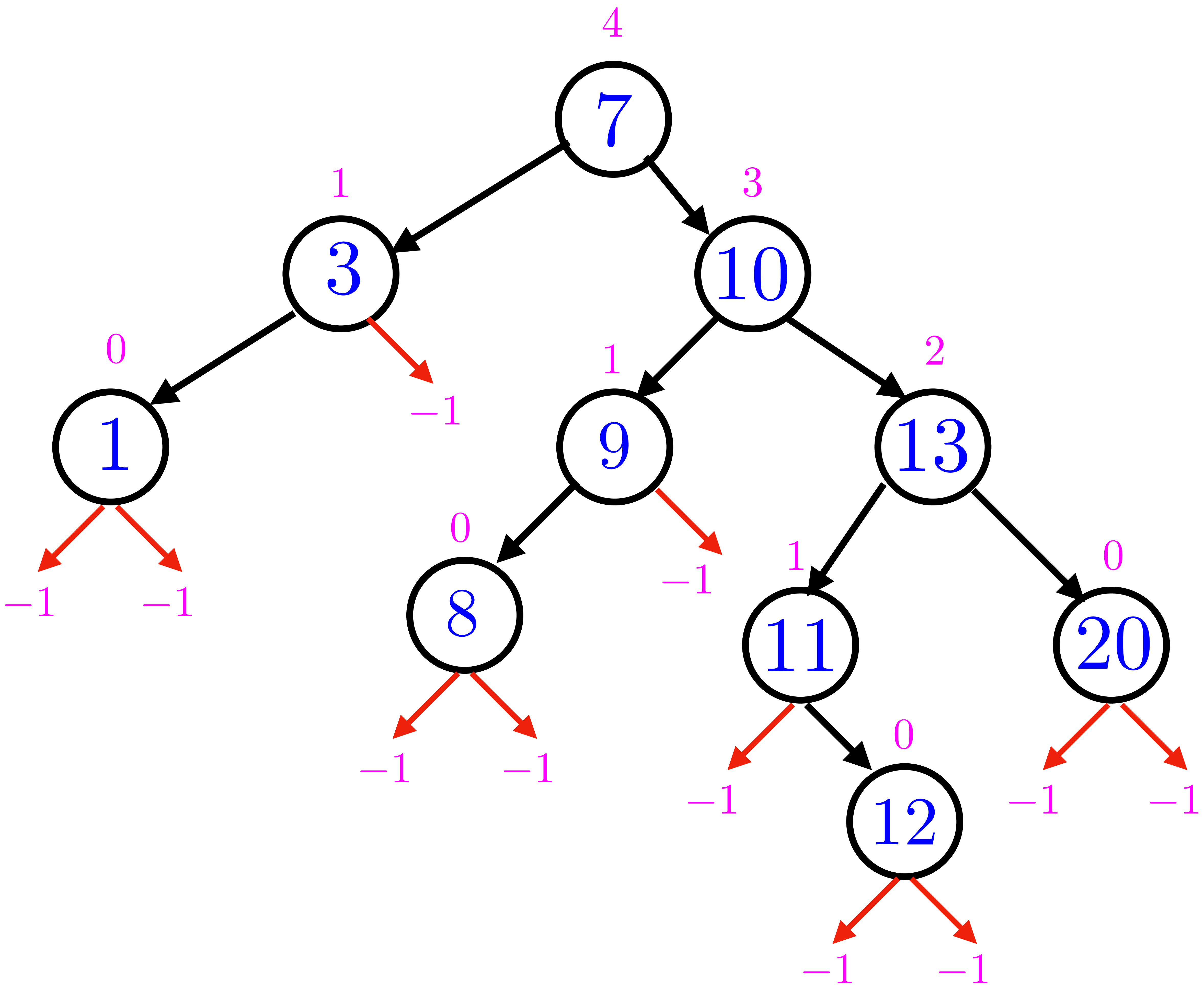
We say a node has the AVL property if it balance factor is in $\{-1, 0, 1\}$.

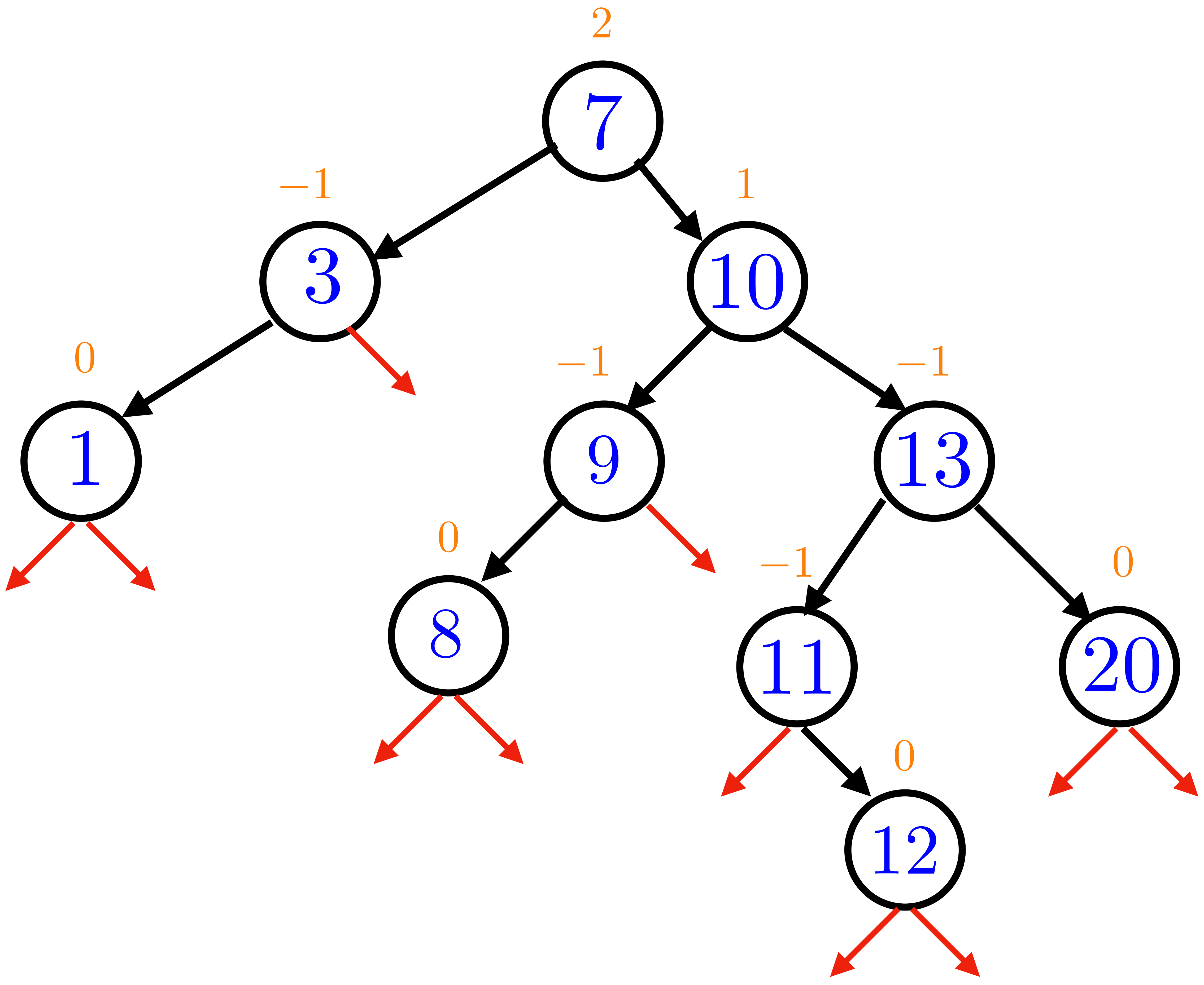
In an AVL tree every node has the AVL property.

Balance Factor



Balance Factor





Is this an AVL tree?

Height of an AVL tree

AVL trees

In a moment we will see how to insert keys and maintain the AVL property.

First let's see why we would want to do this.

Key fact: an AVL tree with n nodes has height at most $2\log n$.

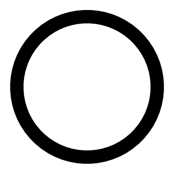
How do we maximise the height of an AVL tree with n nodes?

How do we minimise the number of nodes in an AVL tree with height h ?

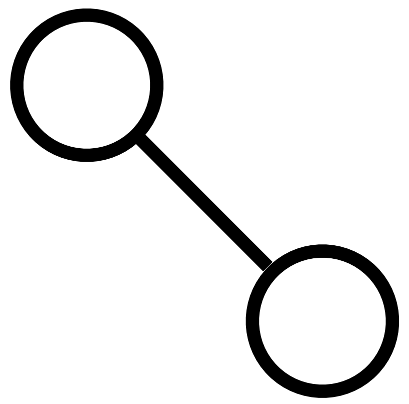
Small examples

Let $T(h)$ be the minimum number of nodes in an AVL tree of height h.

$T(0) = 1$



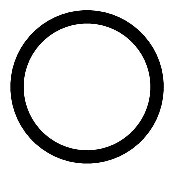
$T(1) = 2$



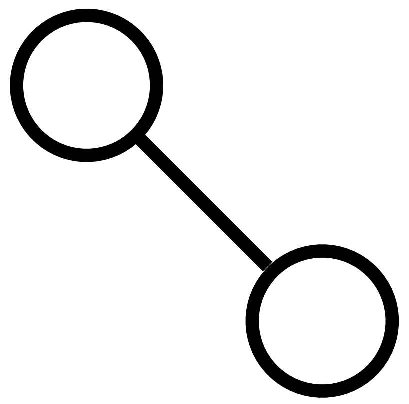
Small examples

Let $T(h)$ be the minimum number of nodes in an AVL tree of height h .

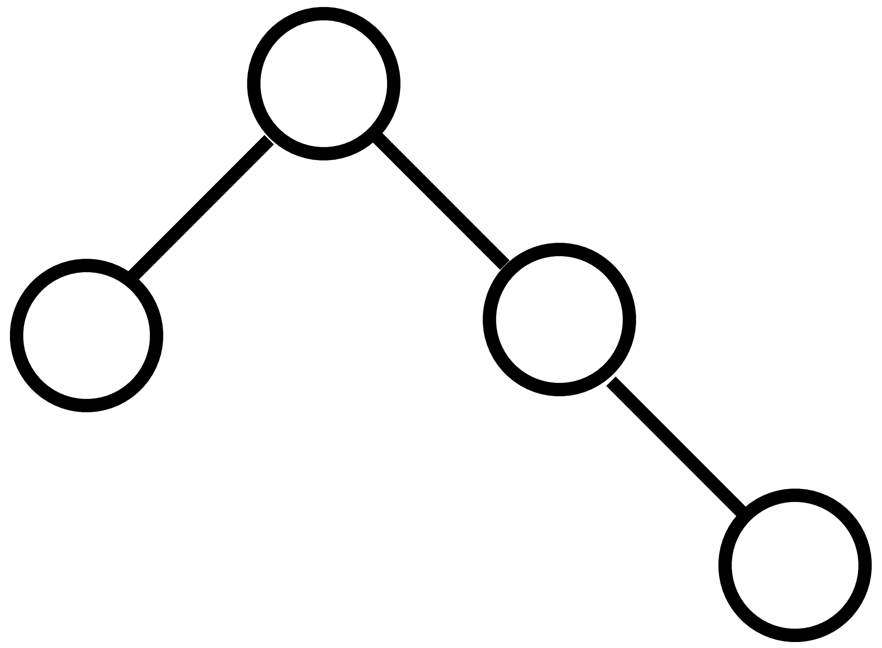
$T(0) = 1$



$T(1) = 2$



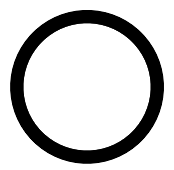
$T(2) = 4$



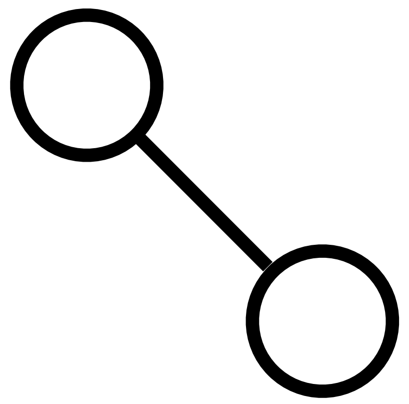
Small examples

Let $T(h)$ be the minimum number of nodes in an AVL tree of height h.

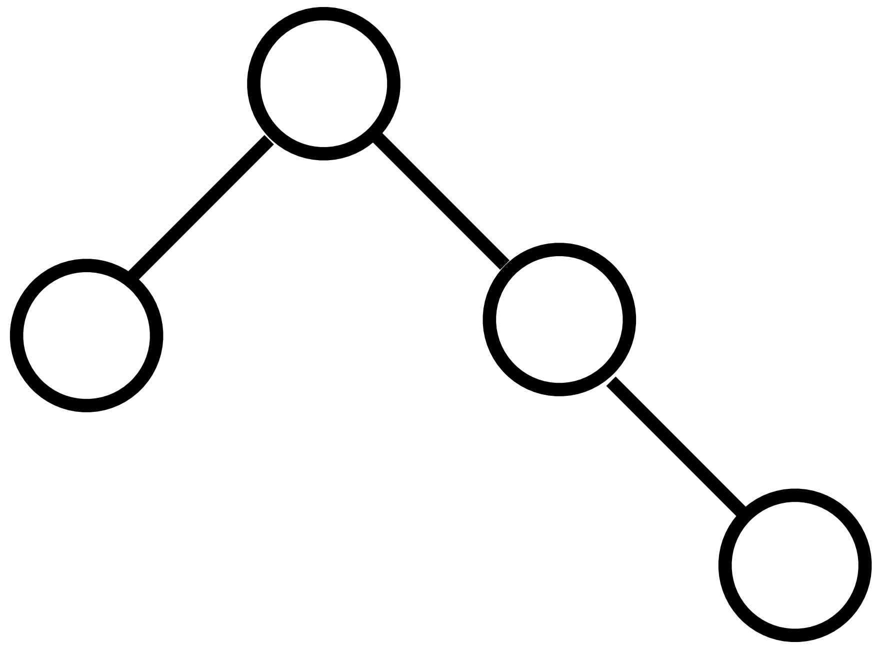
$$T(0) = 1$$



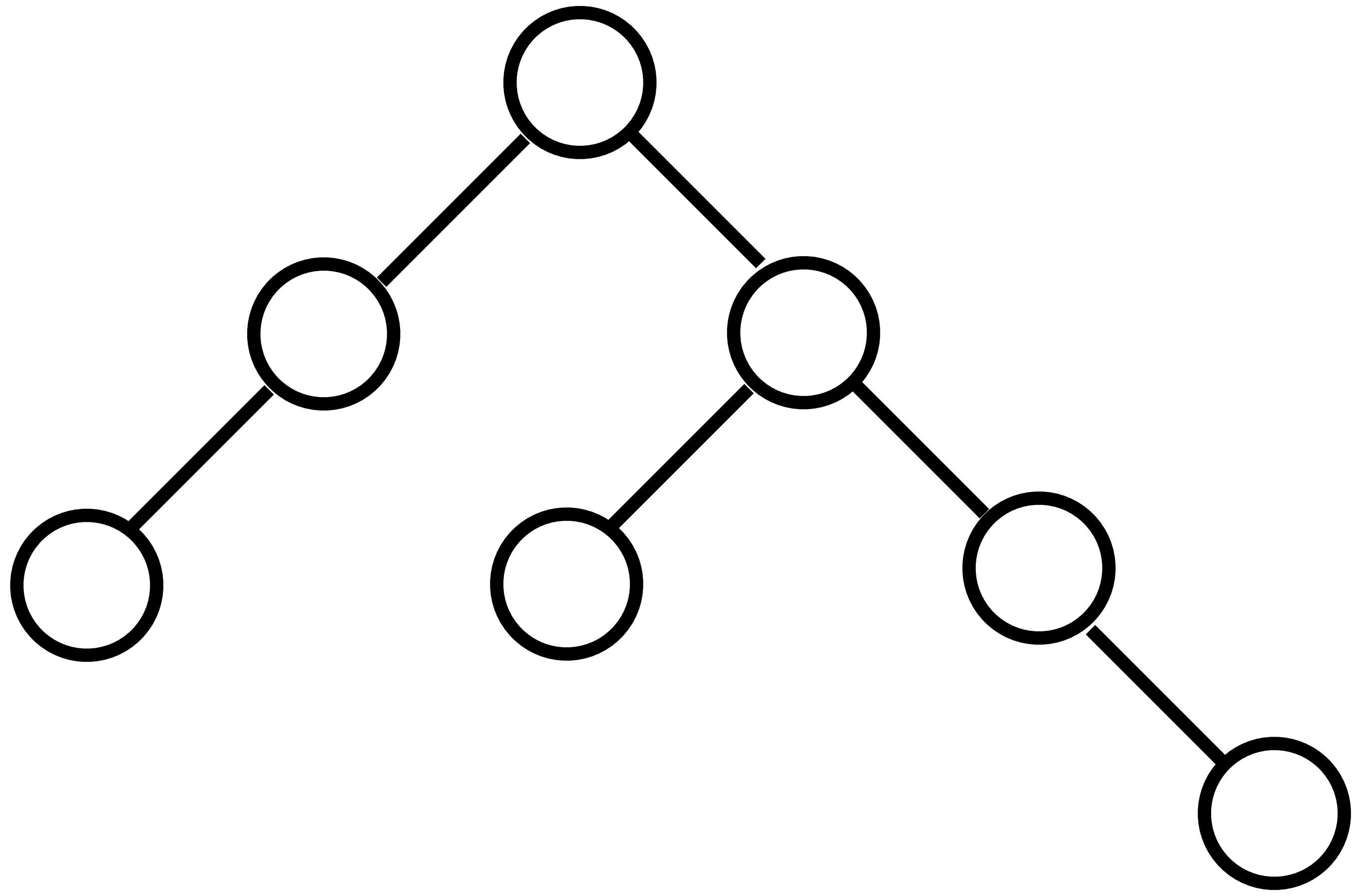
$$T(1) = 2$$



$$T(2) = 4$$



$$T(3) = 7$$

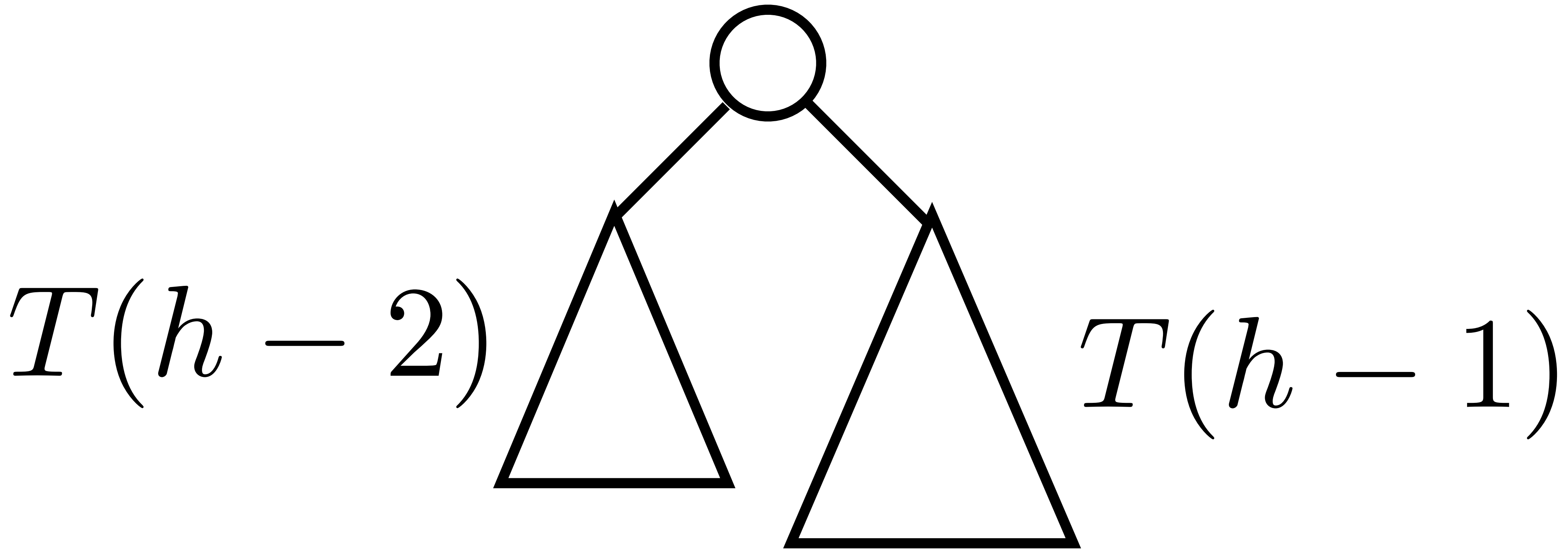


General Case

Let $T(h)$ be the minimum number of nodes in an AVL tree of height h.

Clearly $T(h)$ is an increasing function.

One child of the root should have height $h-1$ and the other $h-2$.



$$T(h) = 1 + T(h-1) + T(h-2)$$

Simple Bound

$$T(h) = 1 + T(h-1) + T(h-2)$$
$$\geq 2T(h-2)$$

$$T(0) = 1$$

$$T(2) \geq 2$$

$$T(4) \geq 4$$

$$T(h) \geq 2^{h/2}$$

Simple Bound

$$\begin{aligned} T(h) &= 1 + T(h-1) + T(h-2) \\ &\geq 2T(h-2) \end{aligned}$$

$$T(0) = 1$$

$$T(2) \geq 2$$

$$T(4) \geq 4$$

$$T(h) \geq 2^{h/2}$$

In a tree with n nodes its height h must satisfy

$$n \geq T(h) \geq 2^{h/2}$$

Simple Bound

$$T(h) = 1 + T(h-1) + T(h-2)$$
$$\geq 2T(h-2)$$

$$T(0) = 1$$
$$T(2) \geq 2$$
$$T(4) \geq 4$$

$$T(h) \geq 2^{h/2}$$

In a tree with n nodes its height h must satisfy

$$n \geq T(h) \geq 2^{h/2}$$

$$\implies 2\log n \geq h$$

Simple Bound

$$T(h) = 1 + T(h-1) + T(h-2)$$
$$\geq 2T(h-2)$$

$$T(0) = 1$$
$$T(2) \geq 2$$
$$T(4) \geq 4$$

$$T(h) \geq 2^{h/2}$$

In a tree with n nodes its height h must satisfy

$$n \geq T(h) \geq 2^{h/2}$$

$$\implies 2 \log n \geq h$$

An AVL tree with n nodes has height $\leq 2 \log n$.

Fibonacci numbers

$$T(h) = 1 + T(h-1) + T(h-2)$$

This looks a lot like the recurrence for Fibonacci numbers!

	0	1	2	3	4	5	6	7	8
Fibonacci	0	1	1	2	3	5	8	13	21
$T(h)$	1	2	4	7	12	20	33	54	88

$$T(h) = F(h+3) - 1$$

This gives a better upper bound on the depth of roughly $1.44 \log n$.

AVL Insert

AVL insert

Say that we have an AVL tree. We want to insert a key and keep the AVL property.

We first insert the key in the usual way.

The insertion changes the height of a node by at most one.

After usual BST insertion, each node has balance factor in $\{-2, -1, 0, 1, 2\}$.

We fix the tree from the bottom up to restore the AVL property.

Rotation

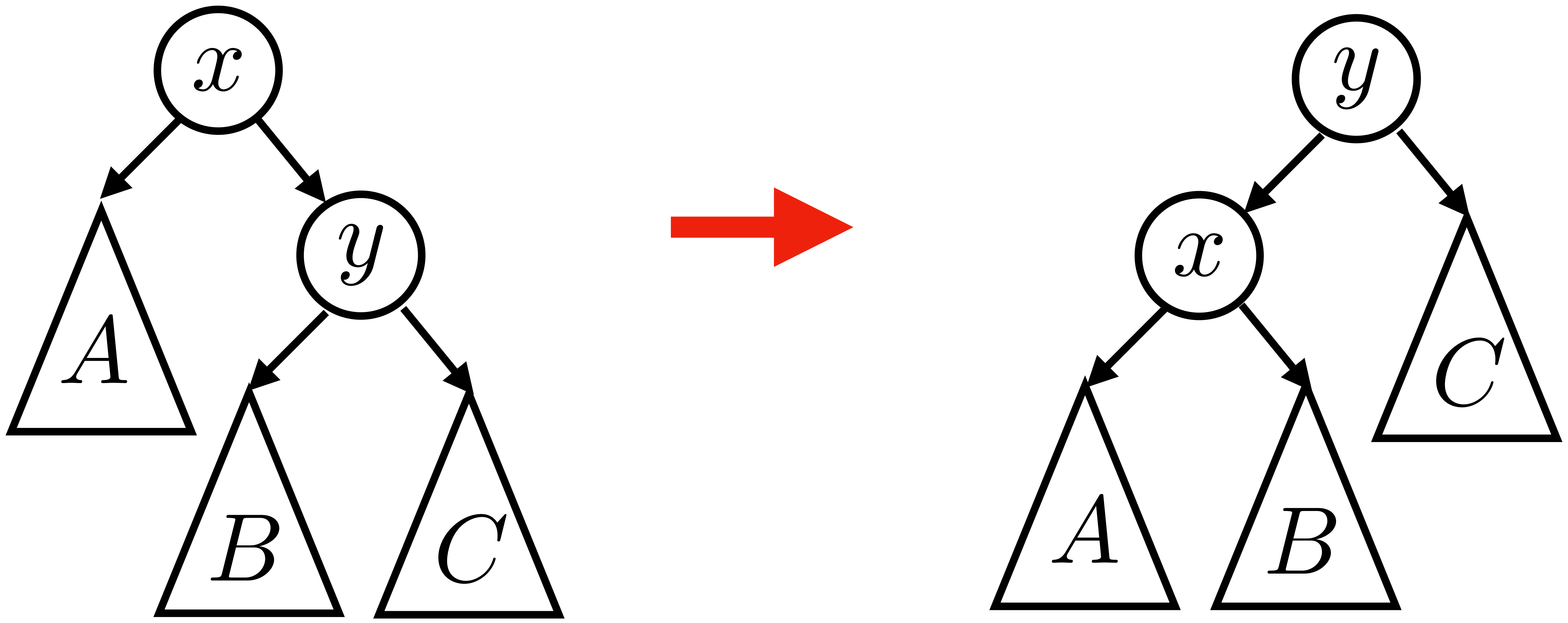
After usual BST insertion, each node has balance factor in $\{-2, -1, 0, 1, 2\}$.

We fix the tree from the bottom up to restore the AVL property.

Let's see how to fix a minimal violating node---all its children satisfy the AVL property.

The key to this fix is an operation on BSTs called rotation.

Left Rotate



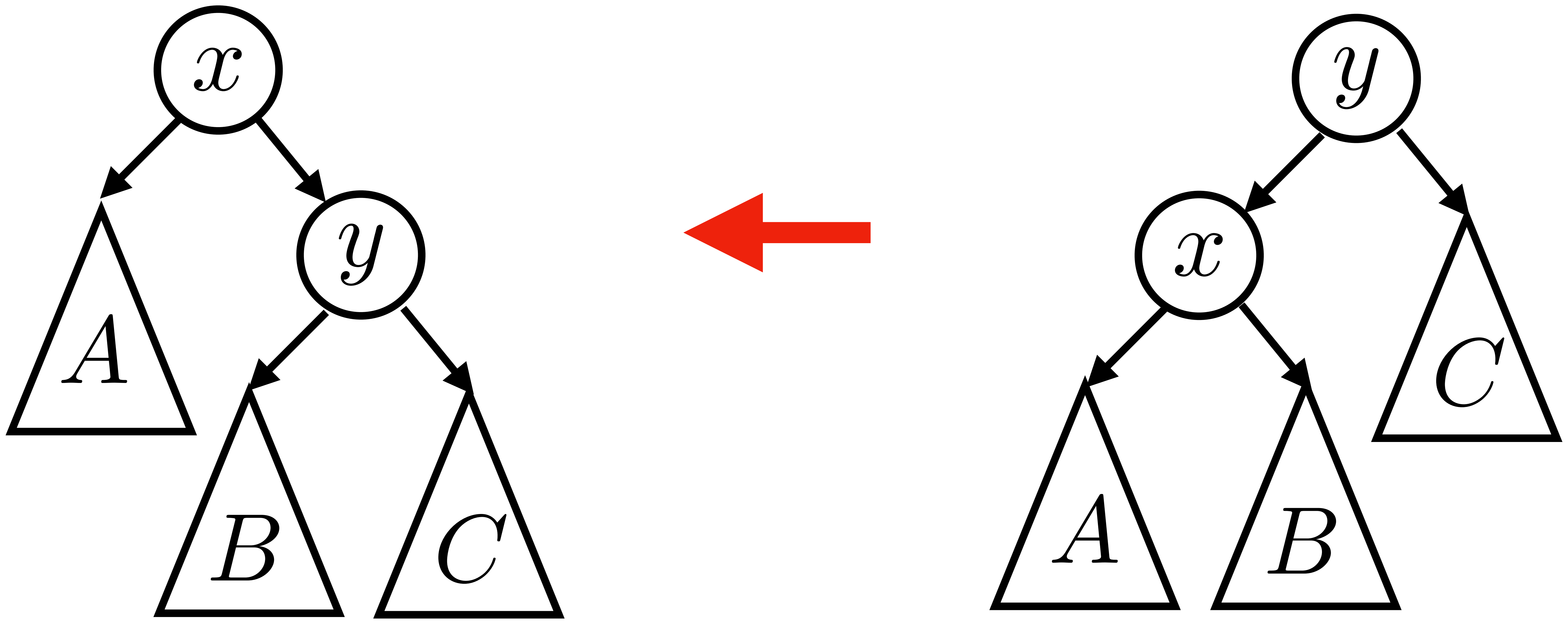
This is a left rotation of x .

Left rotate can be done with a constant number of pointer changes.

Key fact: Left rotation preserves the BST property.

Everything in B is greater than the key at x and less than the key at y.

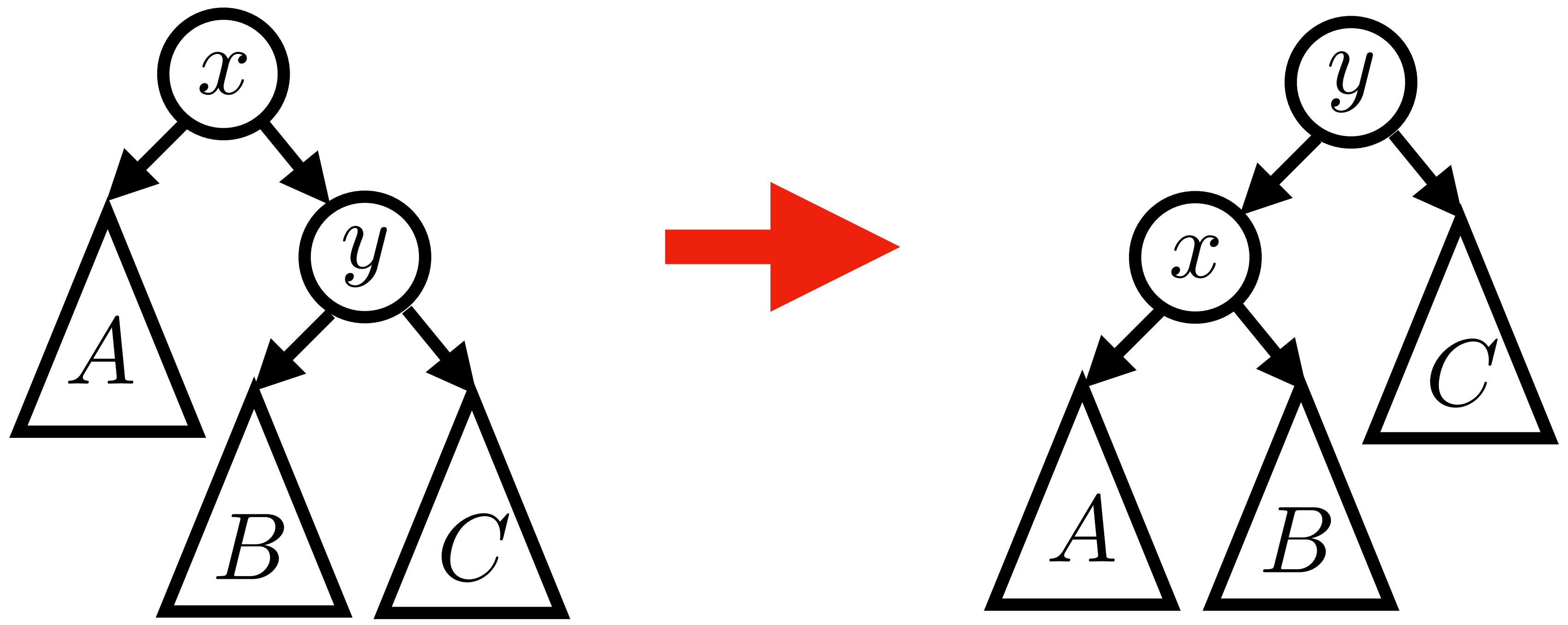
Right Rotate

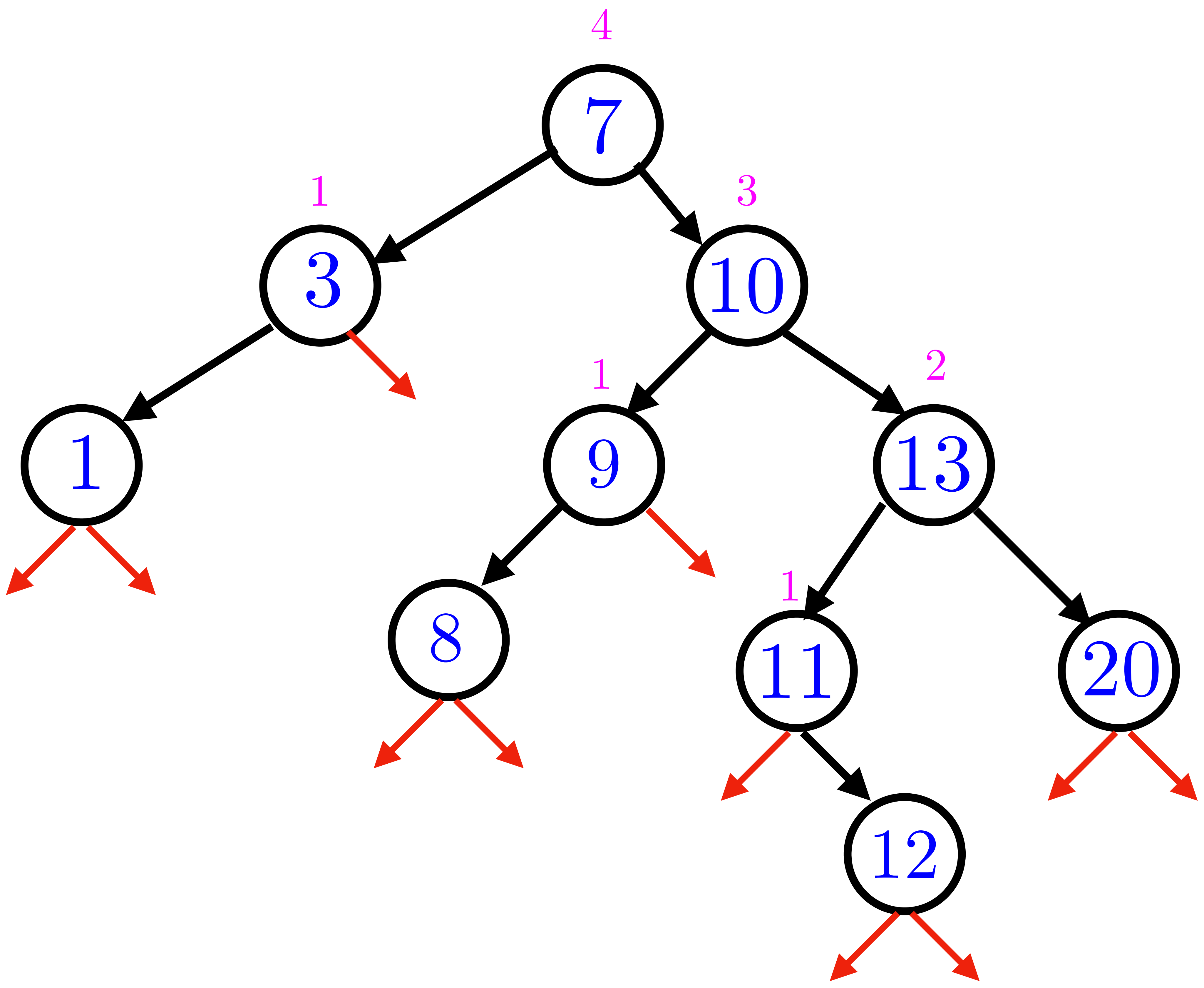


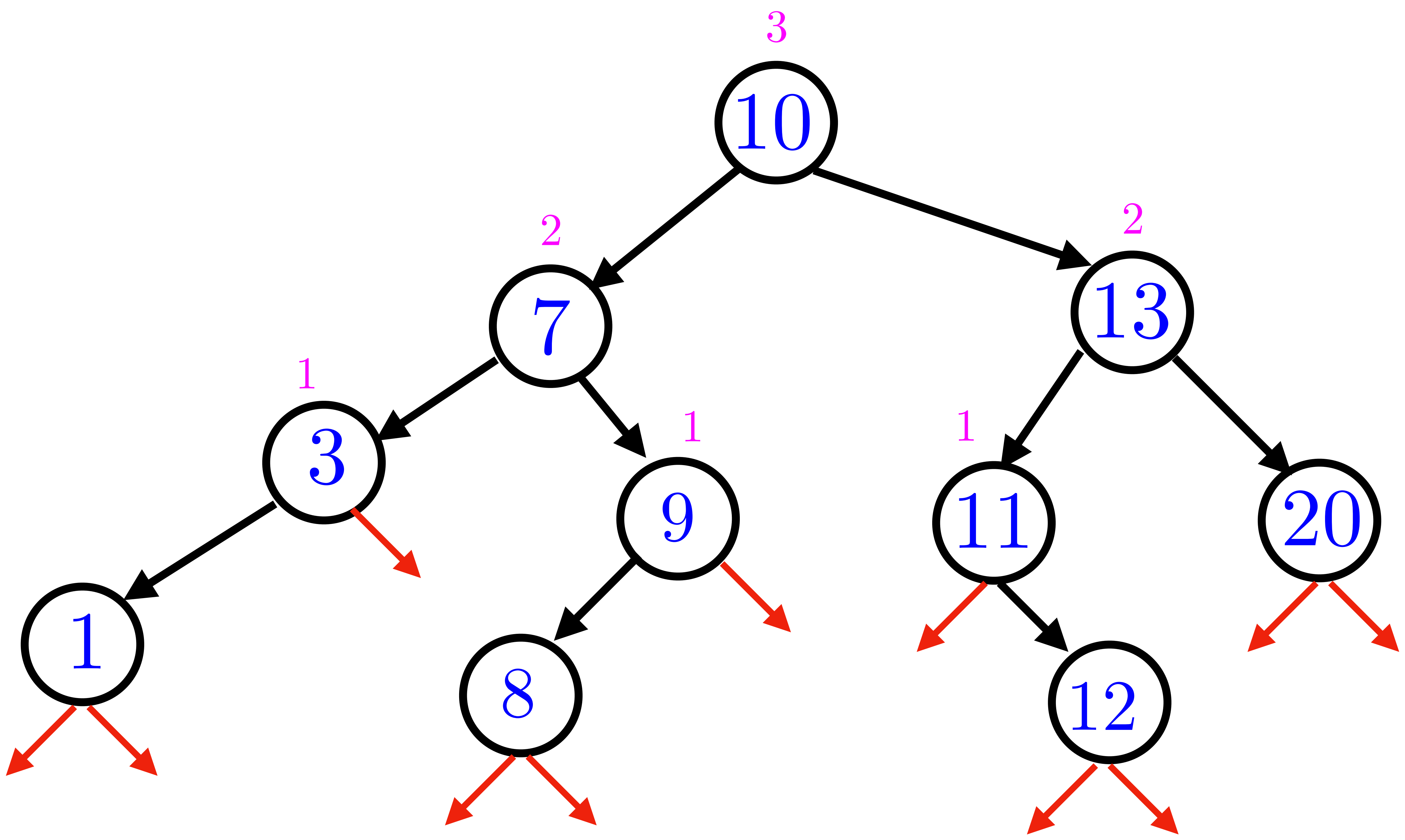
Right rotation is the inverse of left rotation. This is a right rotation of y .

It also preserves the BST property.

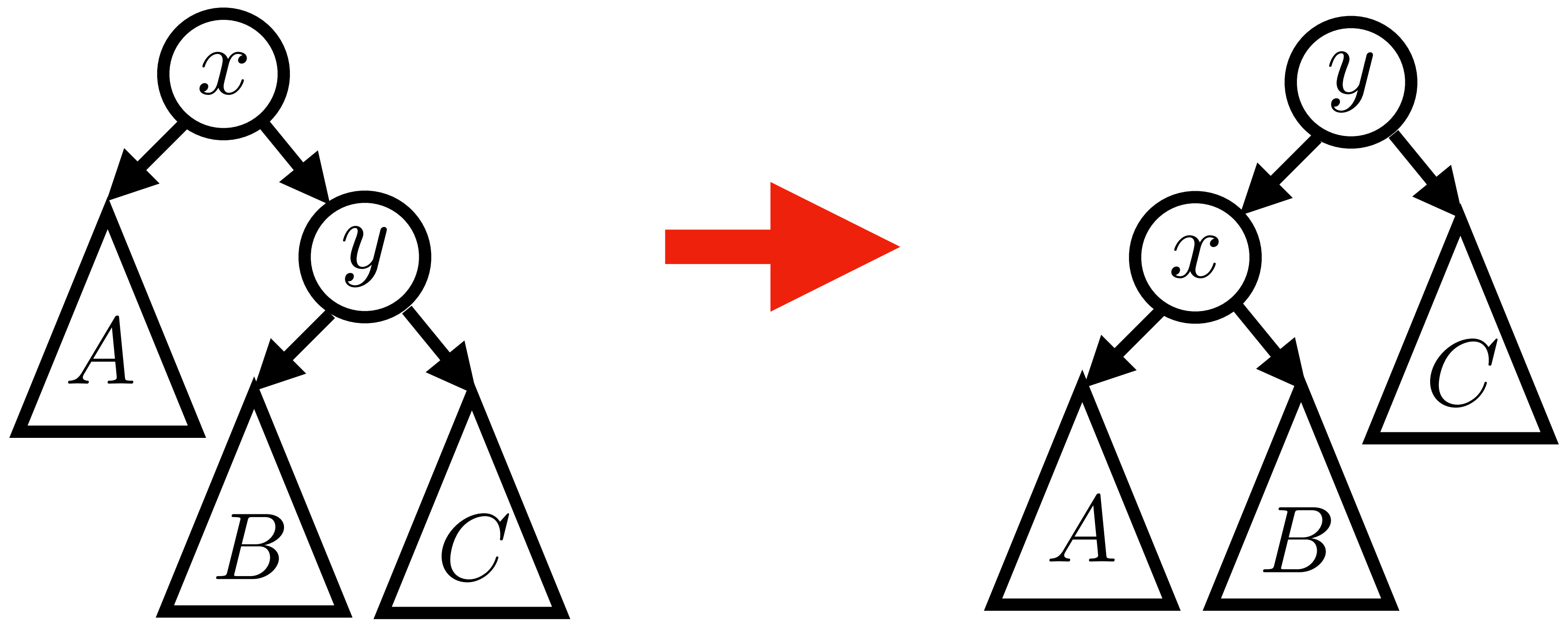
Example: Left Rotate



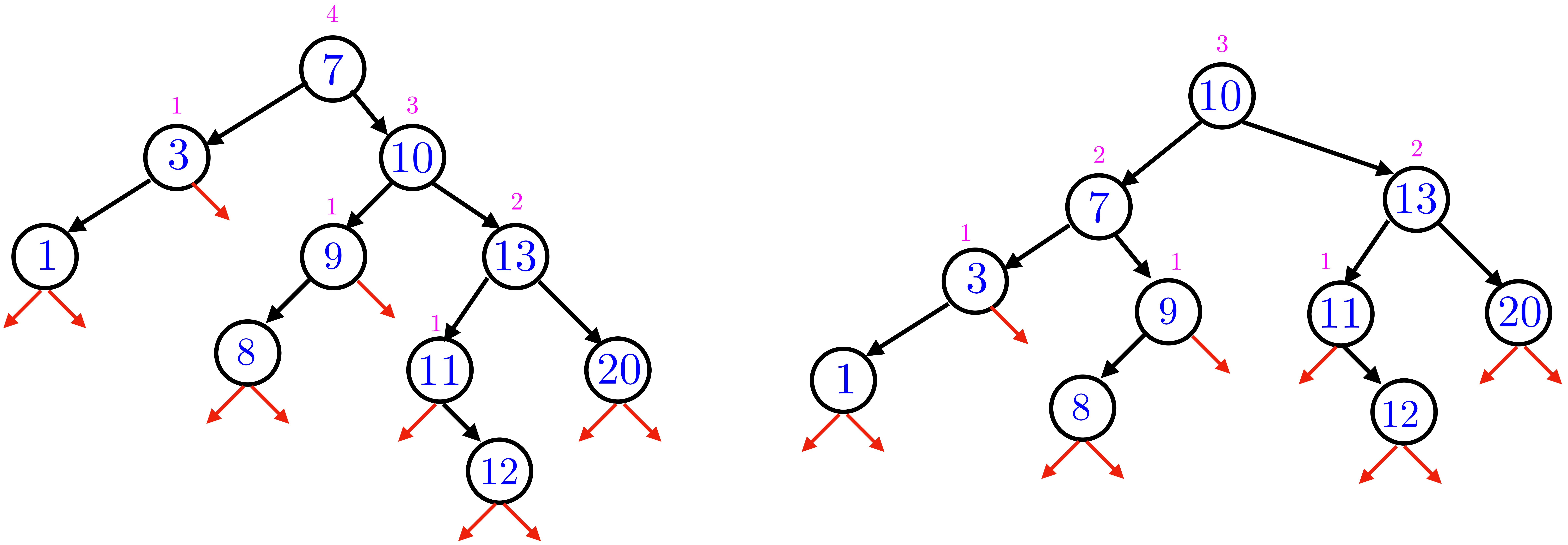




Example: Left Rotate



This restores the AVL property!

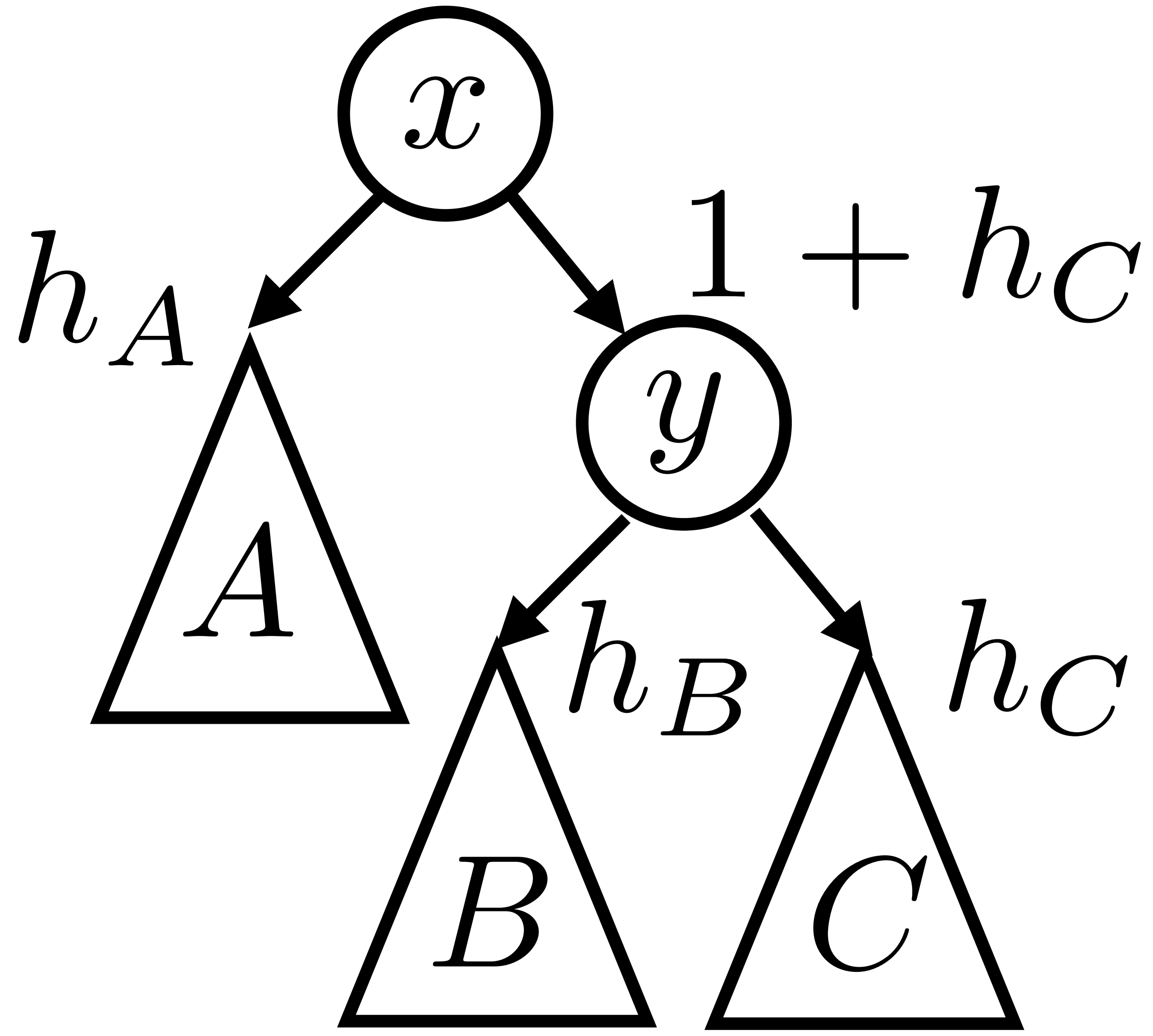


General Case 1

Assume x is a minimal violating node with balance factor 2.

For case 1 we further assume $h_C \geq h_B$.

In this case a left rotation restores the AVL property.



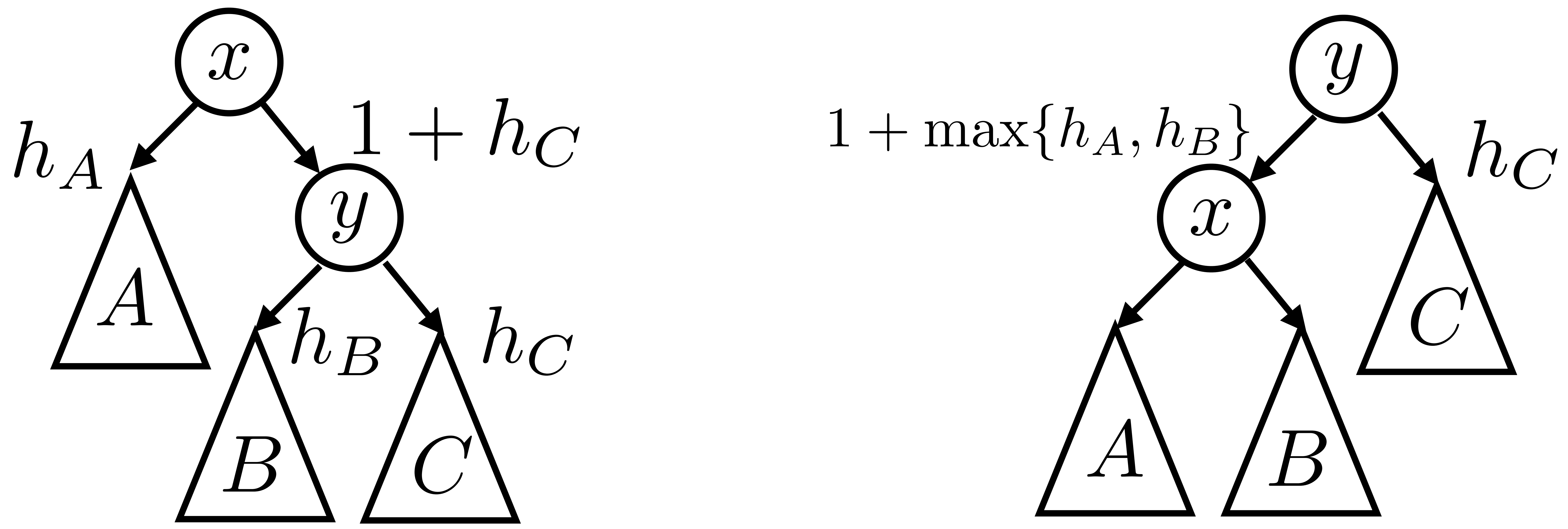
We know that $1 + h_C = h_A + 2$ so $h_A = h_C - 1$.

General Case 1

Assume x is a minimal violating node with balance factor 2.

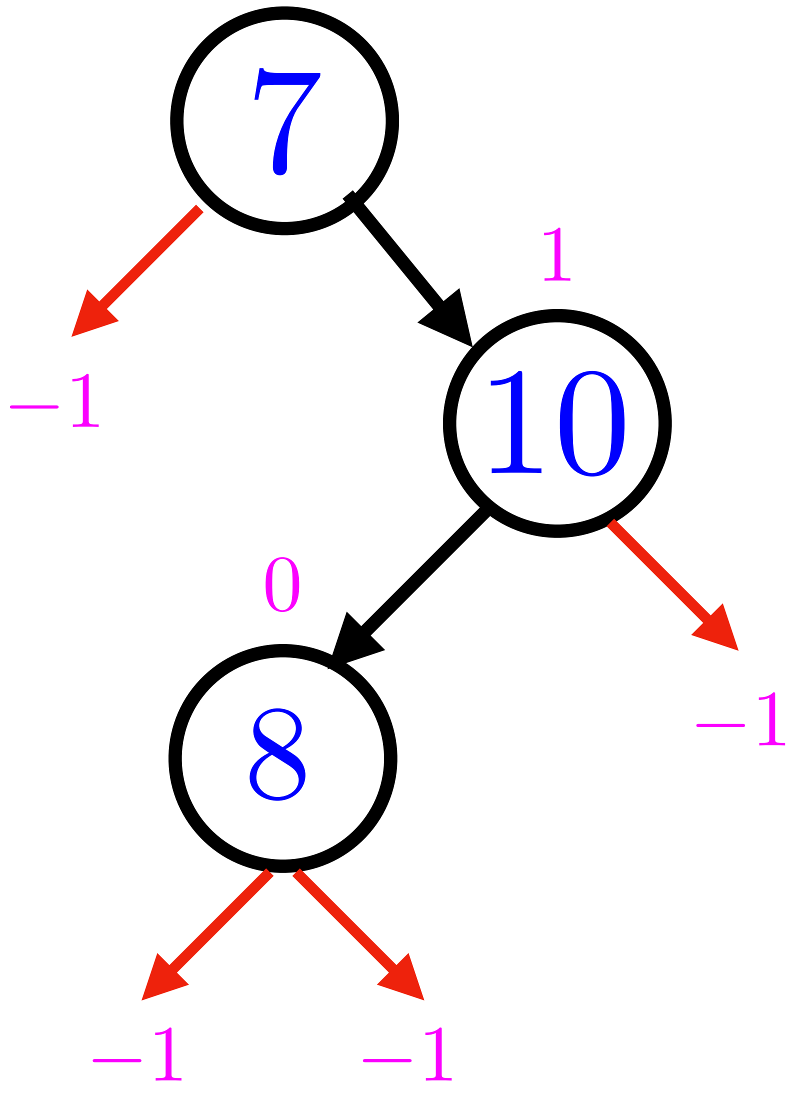
For case 1 we further assume $h_C \geq h_B$.

This means $h_A = h_C - 1$ and so $\max\{h_A, h_B\} \in \{h_C - 1, h_C\}$.



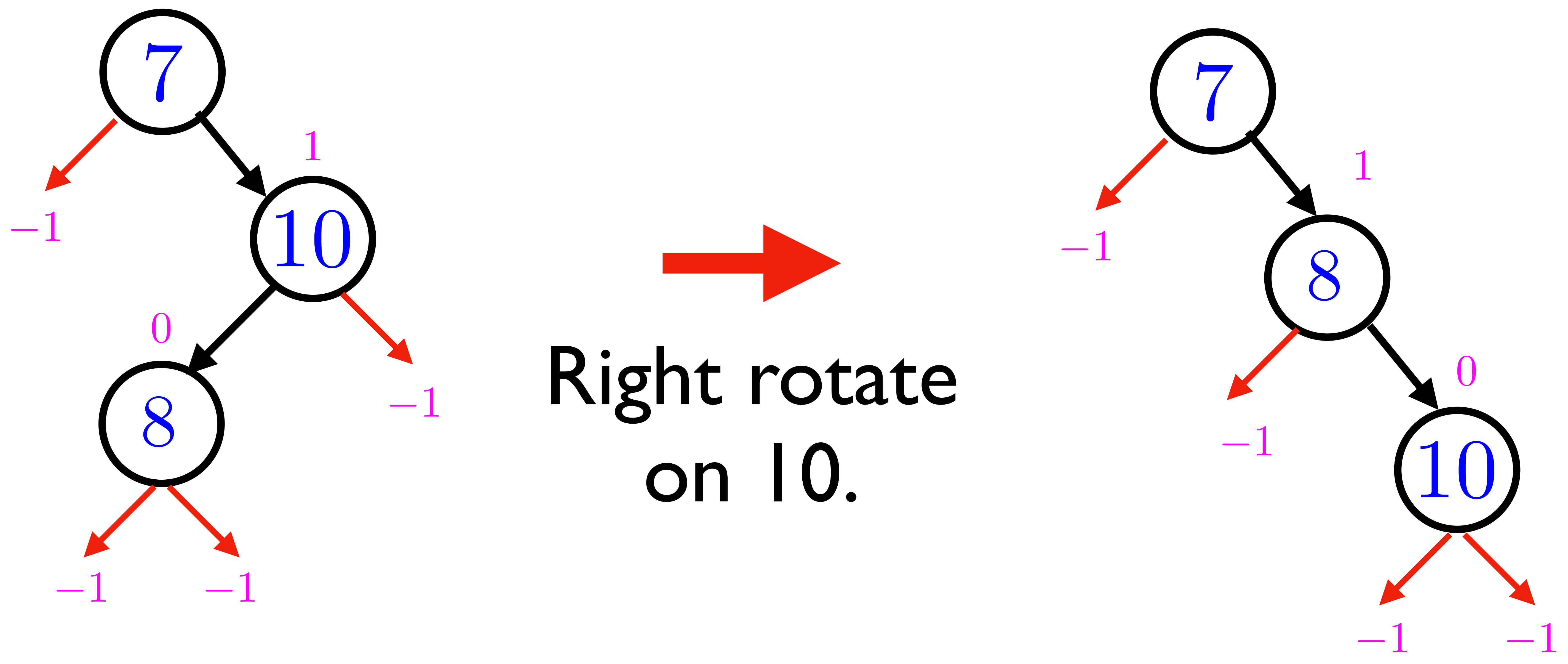
Case 2: Example

The case we haven't handled is where $h_B > h_C$.



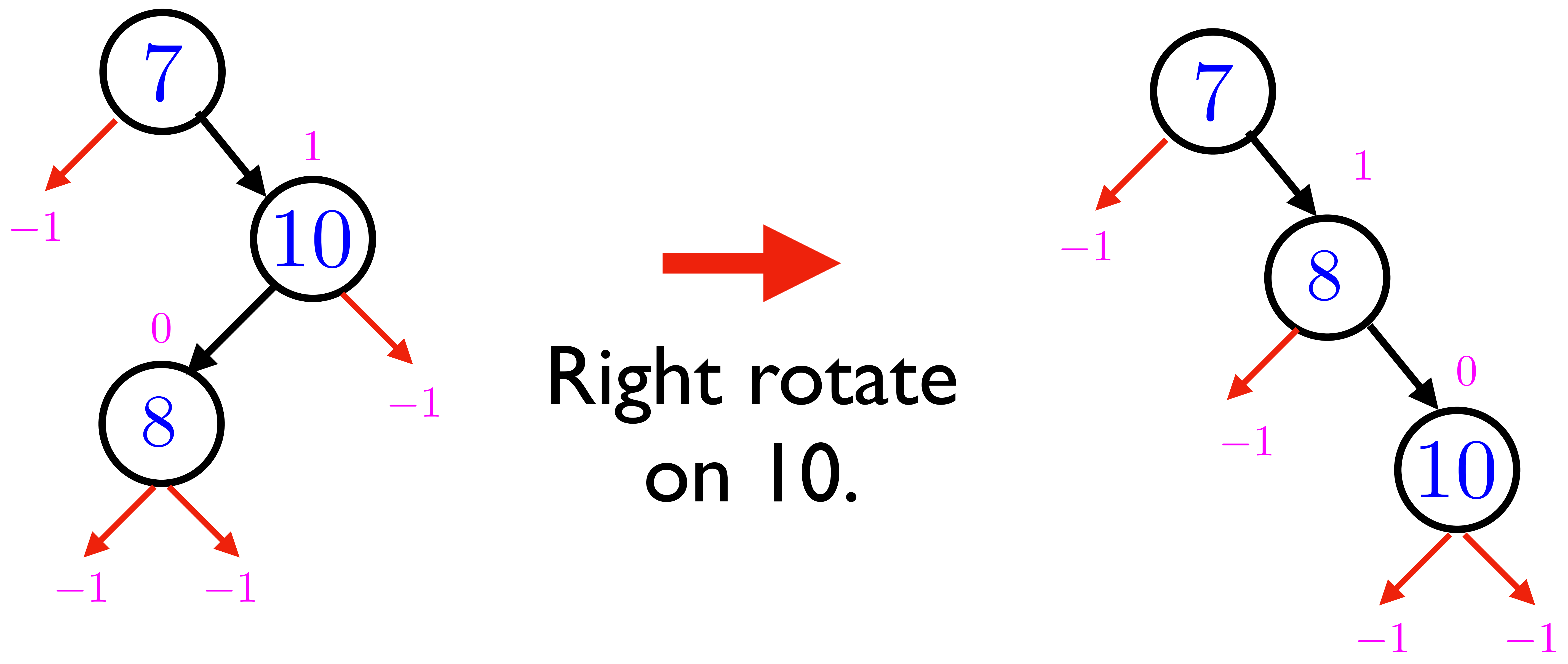
Case 2: Example

The case we haven't handled is where $h_B > h_C$.



Case 2: Example

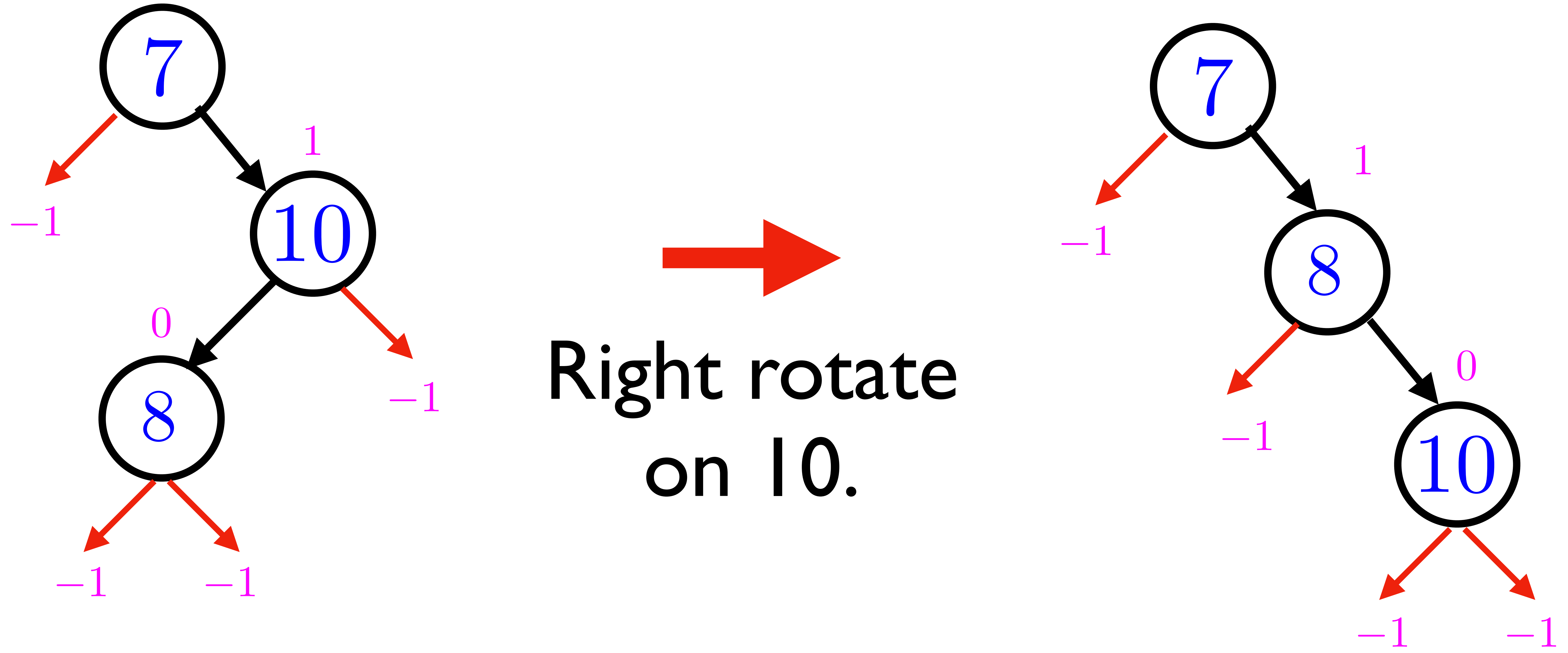
The case we haven't handled is where $h_B > h_C$.



Wait, the tree is still not AVL!

Case 2: Example

The case we haven't handled is where $h_B > h_C$.

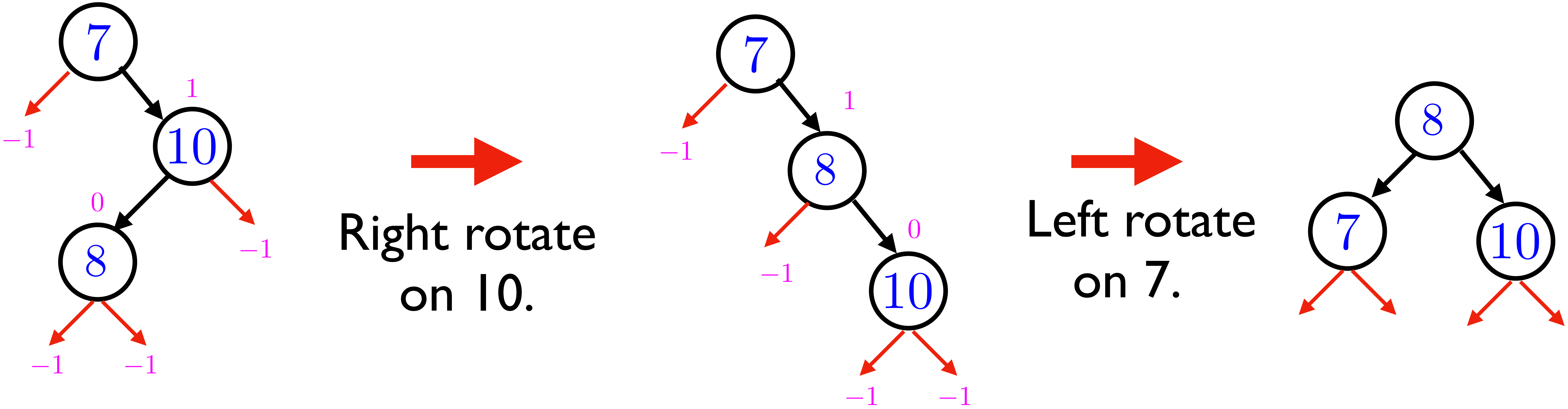


Wait, the tree is still not AVL!

But now we have reduced it to Case 1.

Case 2: Example

The case we haven't handled is where $h_B > h_C$.



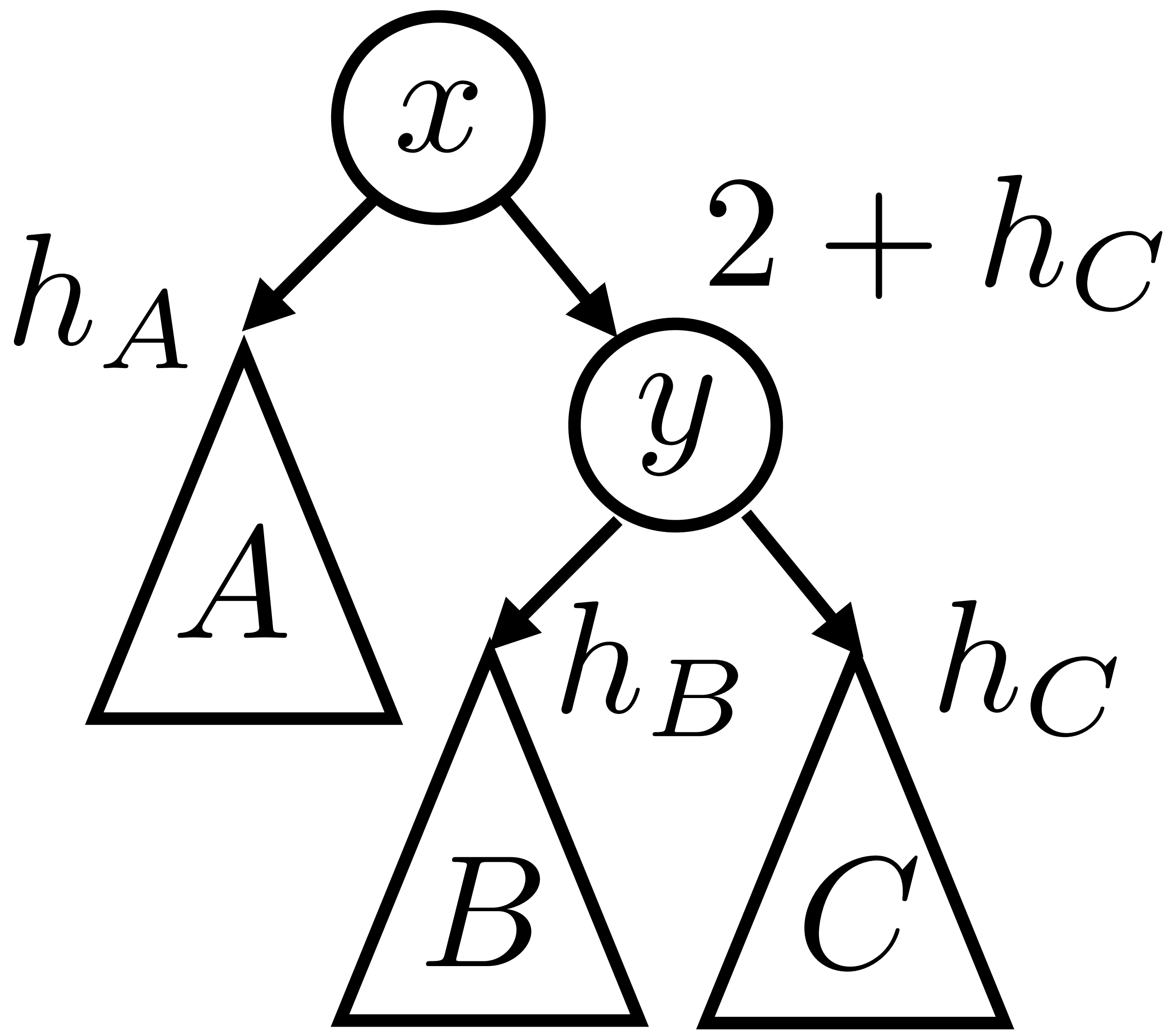
Wait, the tree is still not AVL!

But now we have reduced it to Case 1.

General Case 2

Assume x is a minimal violating node with balance factor 2.

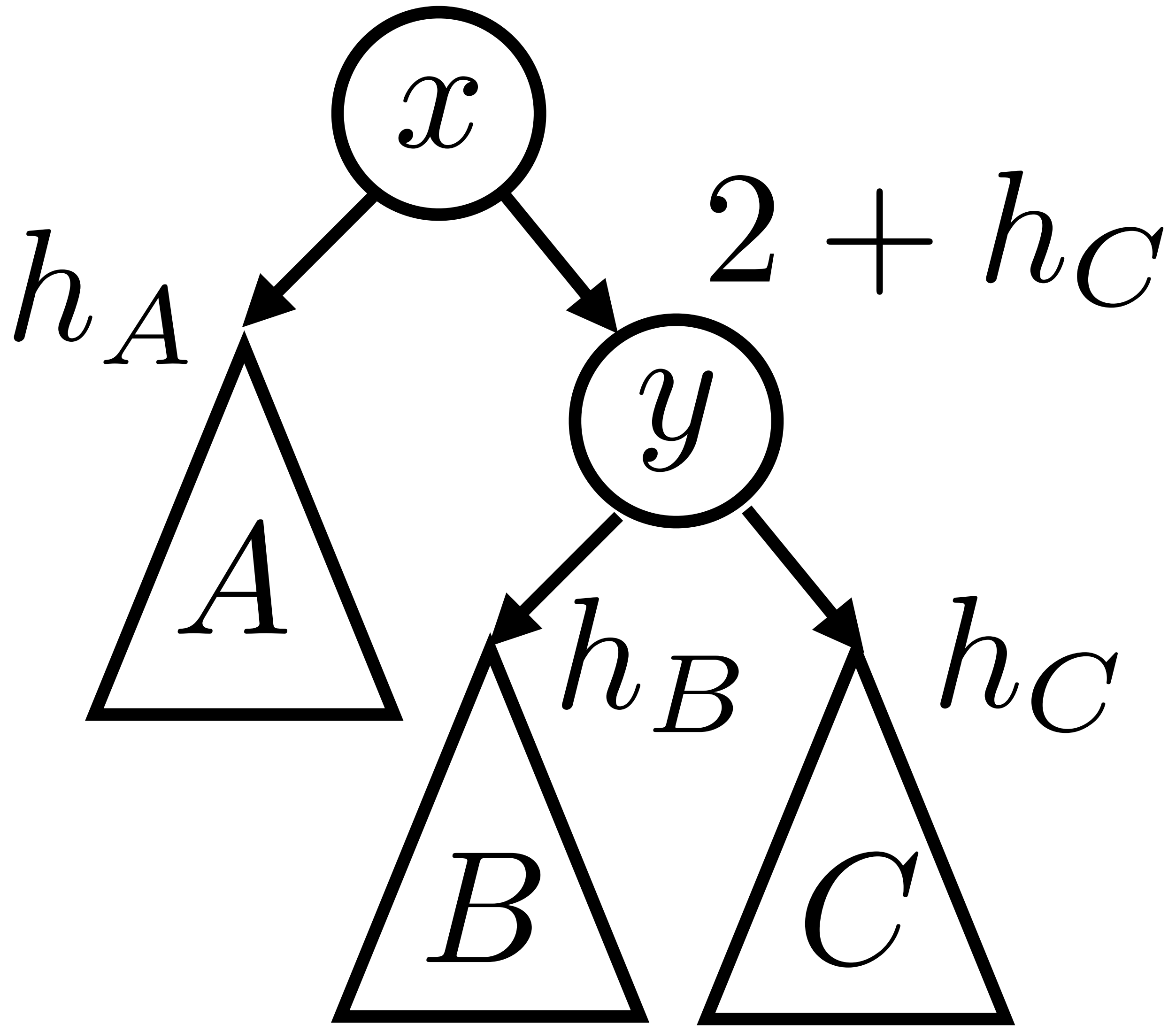
In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



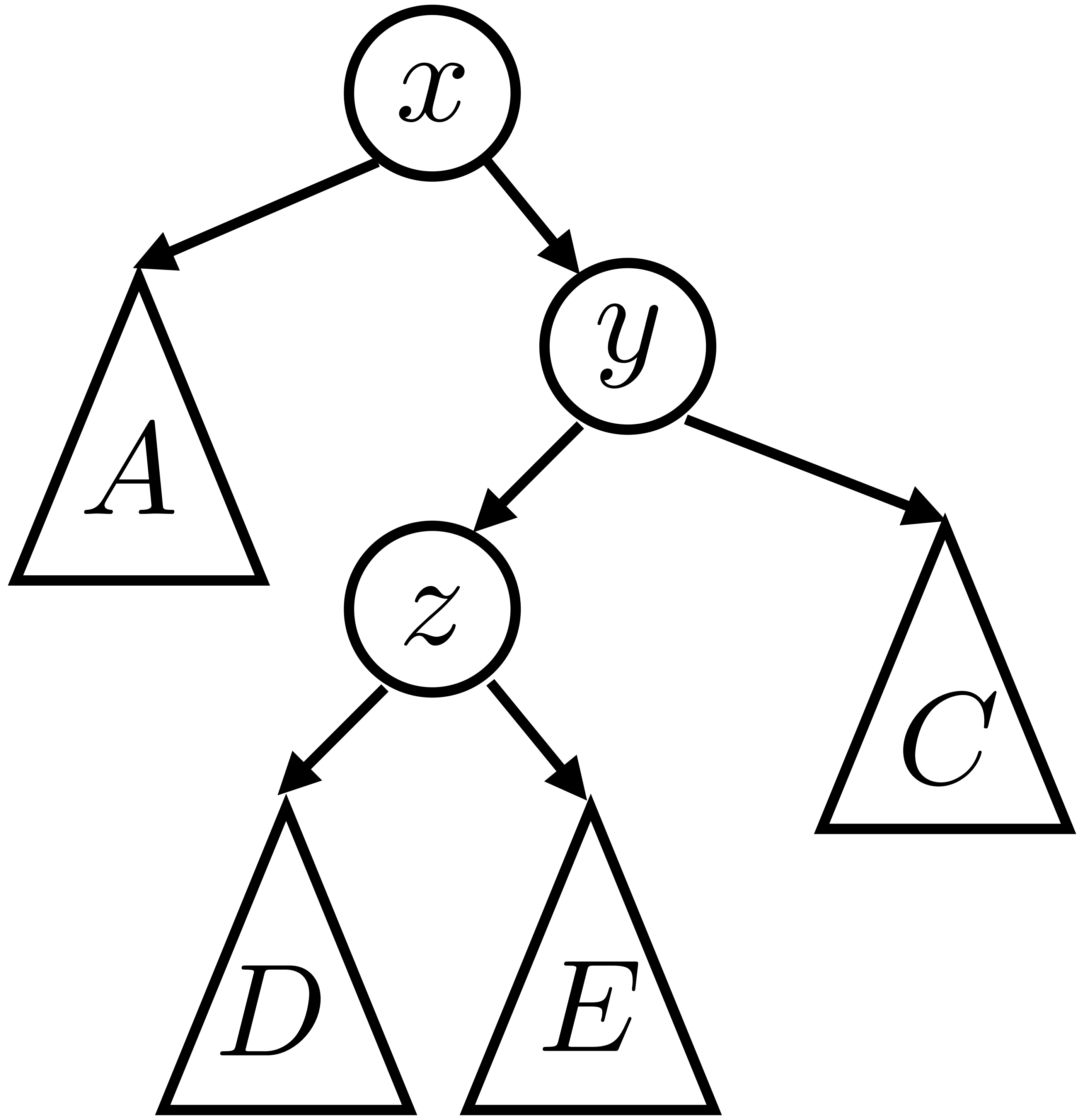
General Case 2

Assume x is a minimal violating node with balance factor 2.

In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



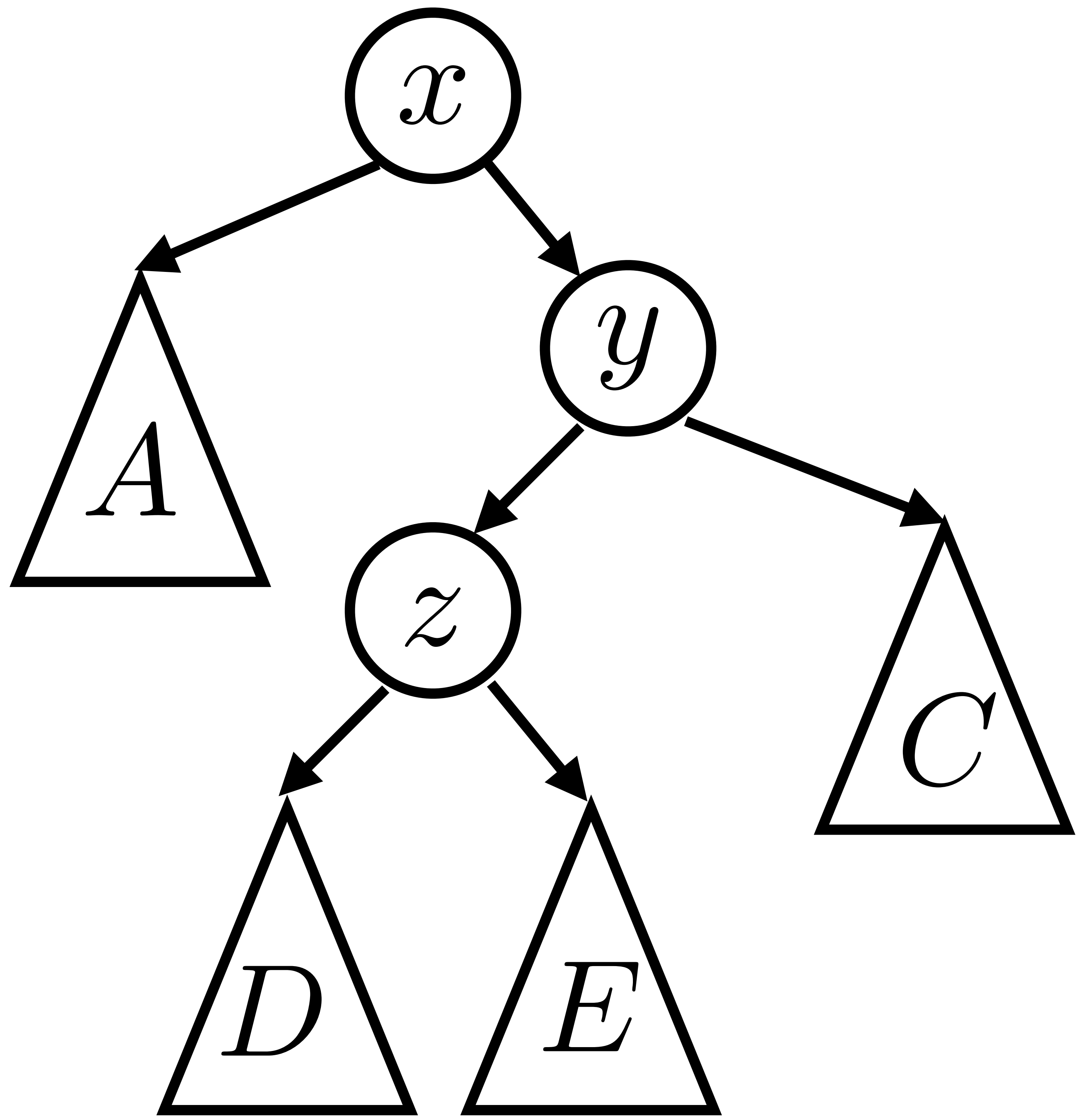
We need to look inside the B tree now.



General Case 2

Assume x is a minimal violating node with balance factor 2.

In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



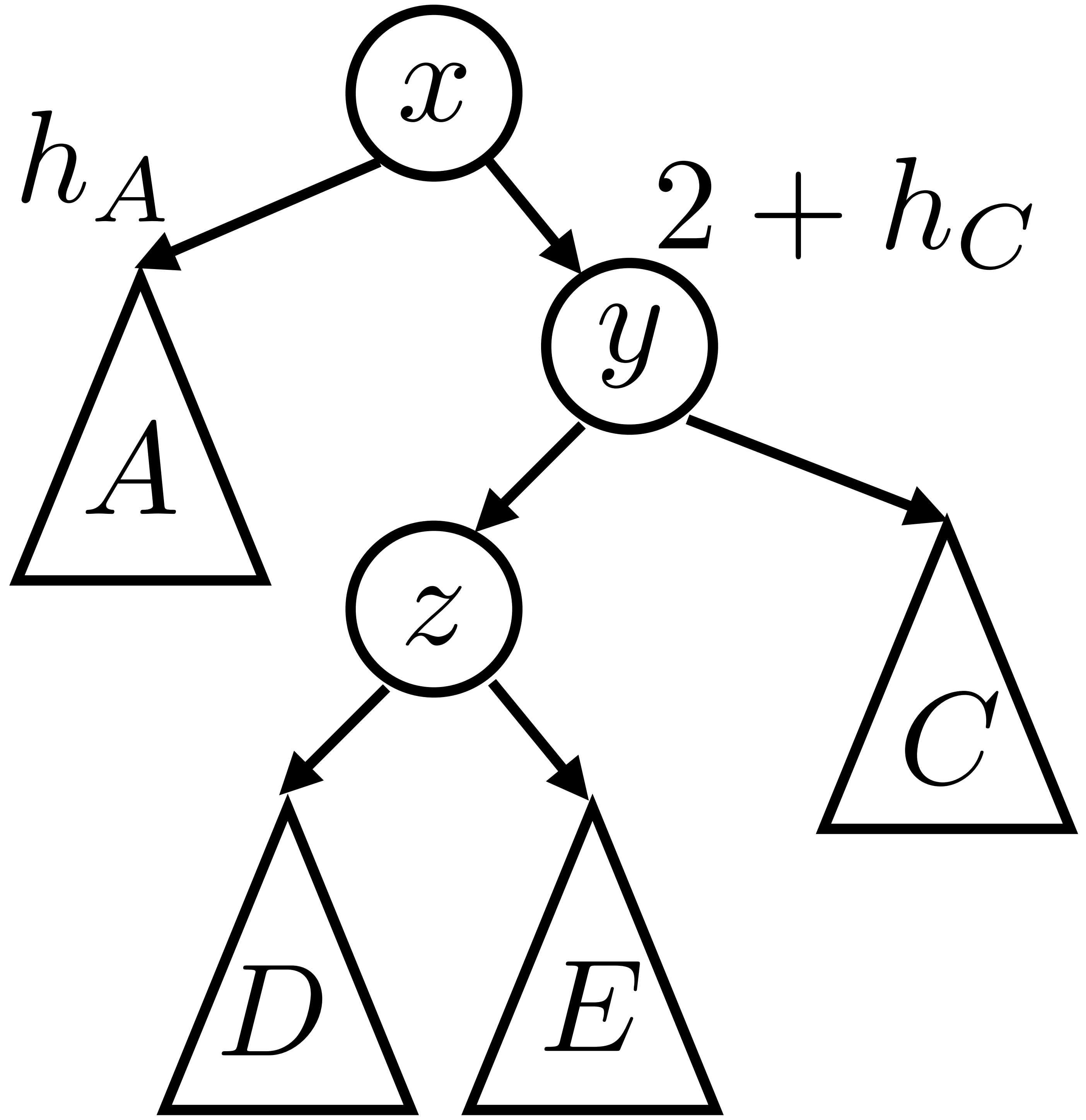
Both of D, E have height $\leq h_C$.

One of them has height h_C .

General Case 2

Assume x is a minimal violating node with balance factor 2.

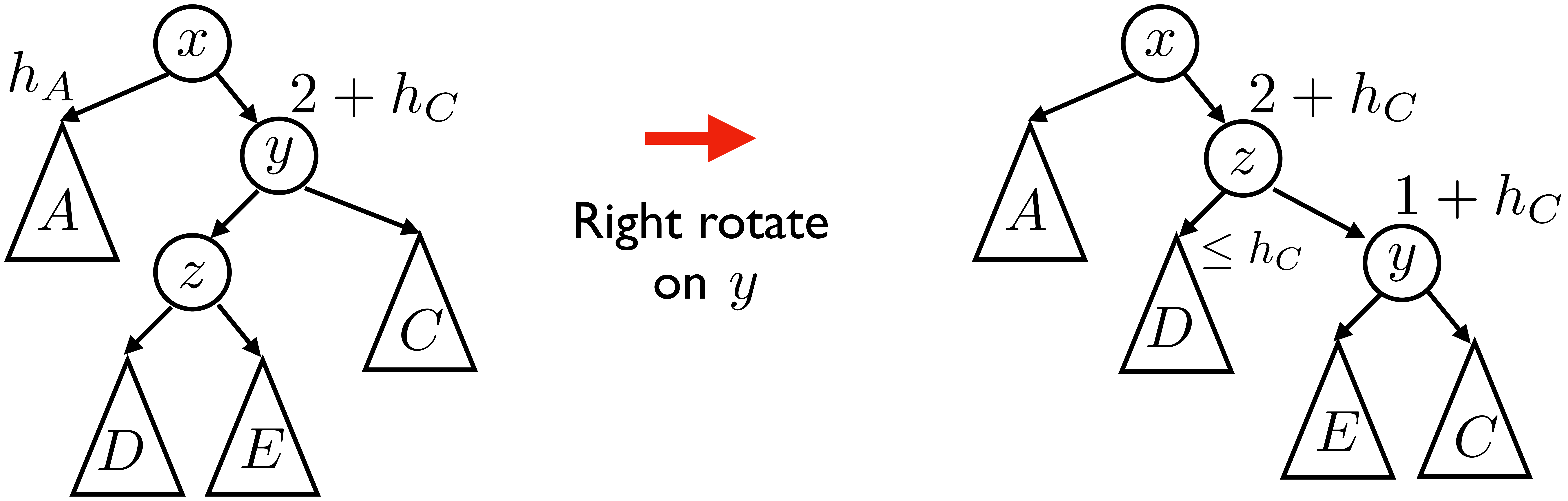
In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



General Case 2

Assume x is a minimal violating node with balance factor 2.

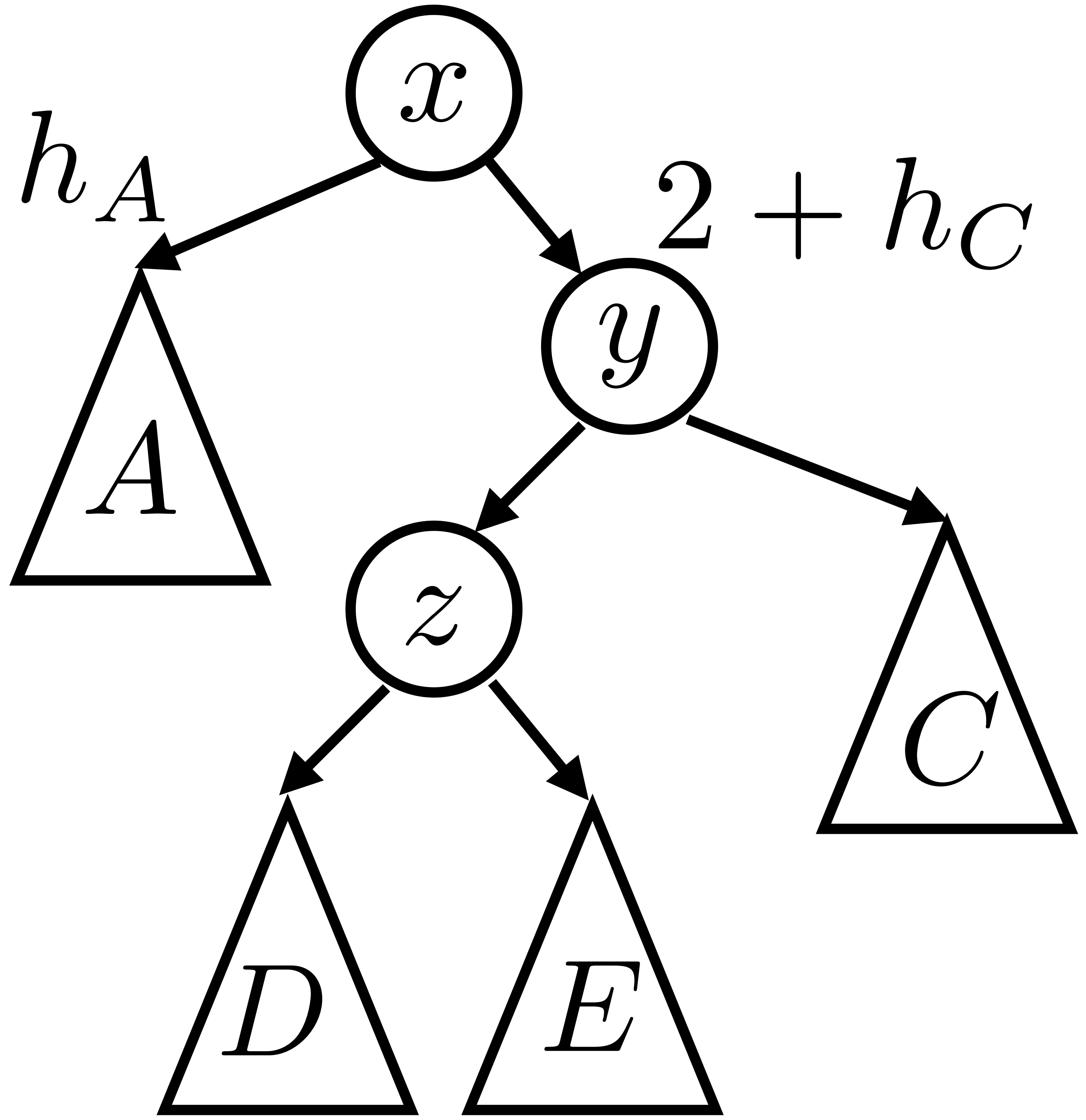
In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



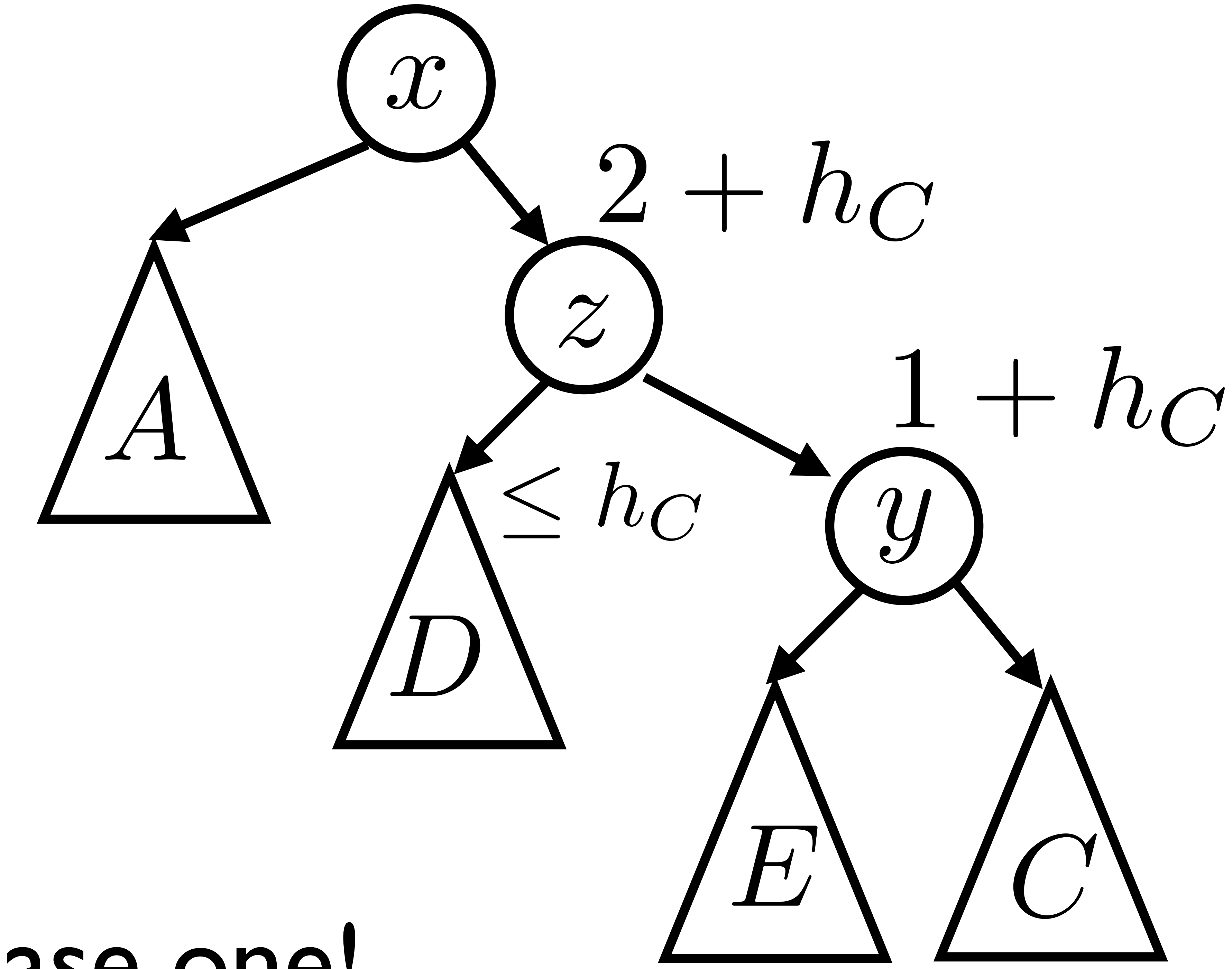
General Case 2

Assume x is a minimal violating node with balance factor 2.

In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.

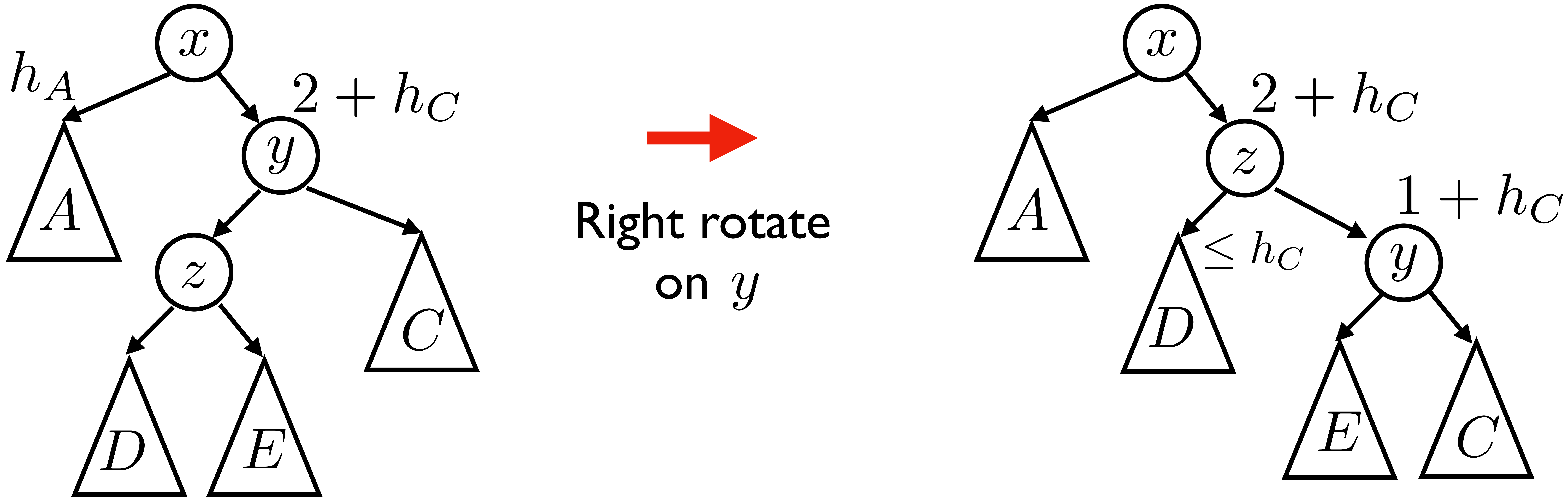


Right rotate on y



Now we are back in case one!

Case 2: Summary



Case 2 can be solved with 2 rotations.

We right rotate on y, and then left rotate on x .

AVL insertion: summary

We have now seen how to repair the balance factor of a single node with a constant number of rotations.

Inserting a node can upset the balance factor of any node on the path from the insertion point to the root.

We may have to do this repair $\Theta(h)$ times.

This still gives us $O(\log n)$ insertion time in an AVL tree.

AVL tree: summary

An AVL tree gives $O(\log n)$ worst case time for all our operations.

operation	worst case
`insert`	$O(\log n)$
`remove`	$O(\log n)$
`contains`	$O(\log n)$
`successor`	$O(\log n)$