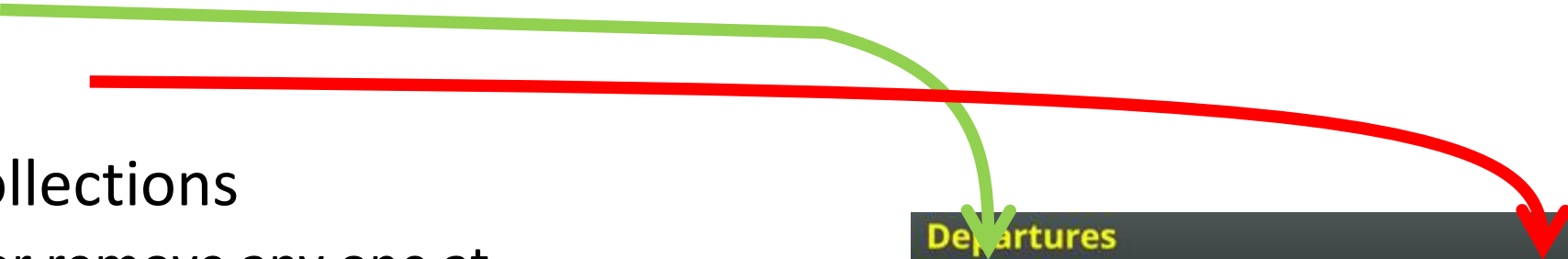


Trees

Problem: Flight Database

- One record of data:
 - Key
 - Data
- Dynamic collections
 - Can add or remove any one at anytime
- Can query the database
 - Find a particular record by the key
 - E.g. what is the flight at 09:05?
 - Or check the existence
 - Is there any flight between 09:00 to 09:20?
 - Find the one before or after
 - successor or predecessor



Departures		
Time	To/From	Flight
08:20	LONDON	UL 125
08:45	NEW YORK	TH 9599
09:05	BARCELONA	AX 571
09:30	MOSCOW	BE 25836
09:55	DUBAI	LK 12121
10:20	PARIS	DM 7324
10:45	ROME	RS 1703
11:10	BERLIN	FX 50714

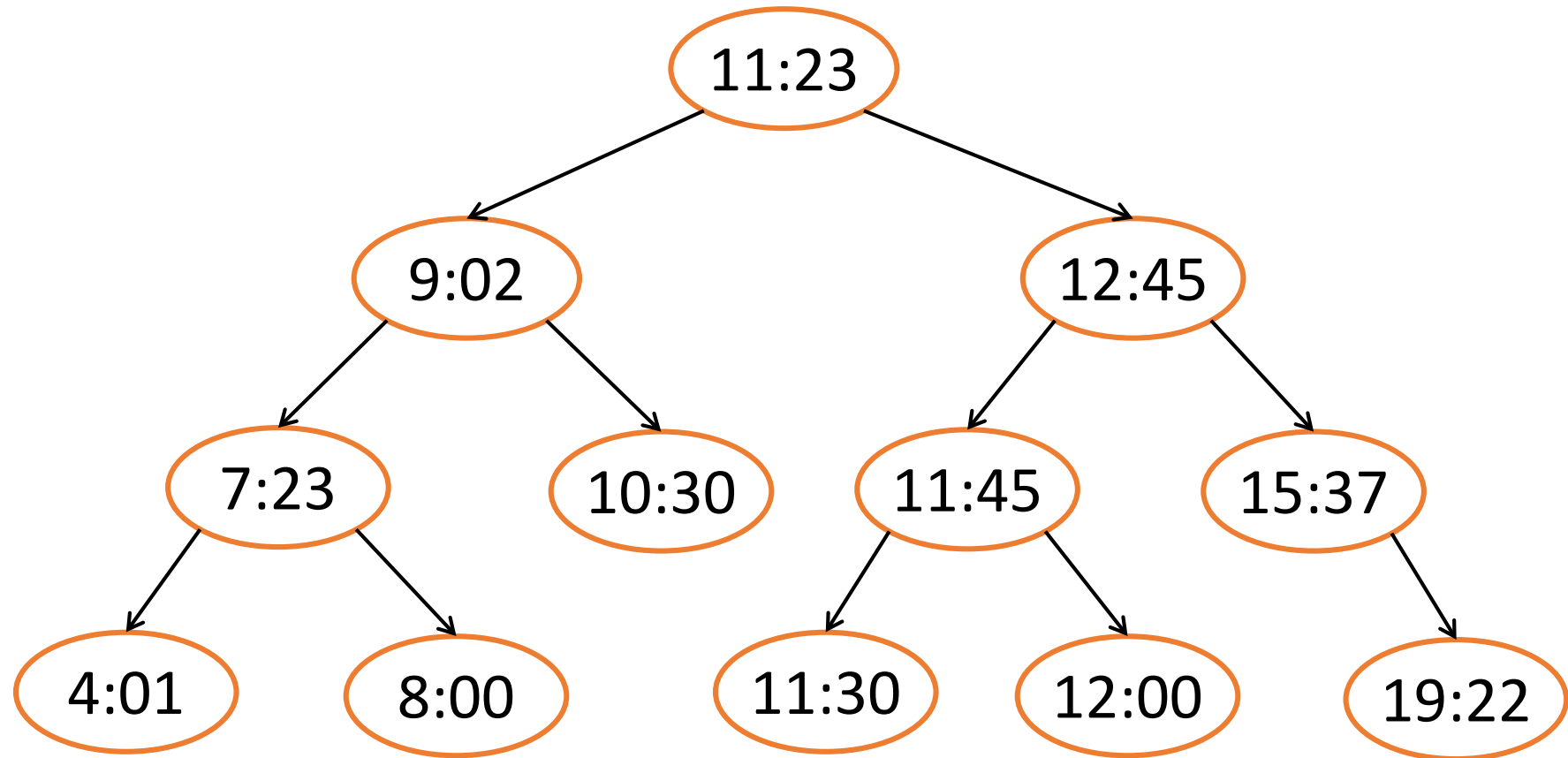
Dictionary ADT

<code>void insert(Key k, Value v)</code>	<i>insert (k,v) into table</i>
<code>Value search(Key k)</code>	<i>get value paired with k</i>
Key successor(Key k)	<i>find next key > k</i>
Key predecessor(Key k)	<i>find next key < k</i>
<code>void delete(Key k)</code>	<i>remove key k (and value)</i>
<code>boolean contains(Key k)</code>	<i>is there a value for k?</i>
<code>int size()</code>	<i>number of (k,v) pairs</i>

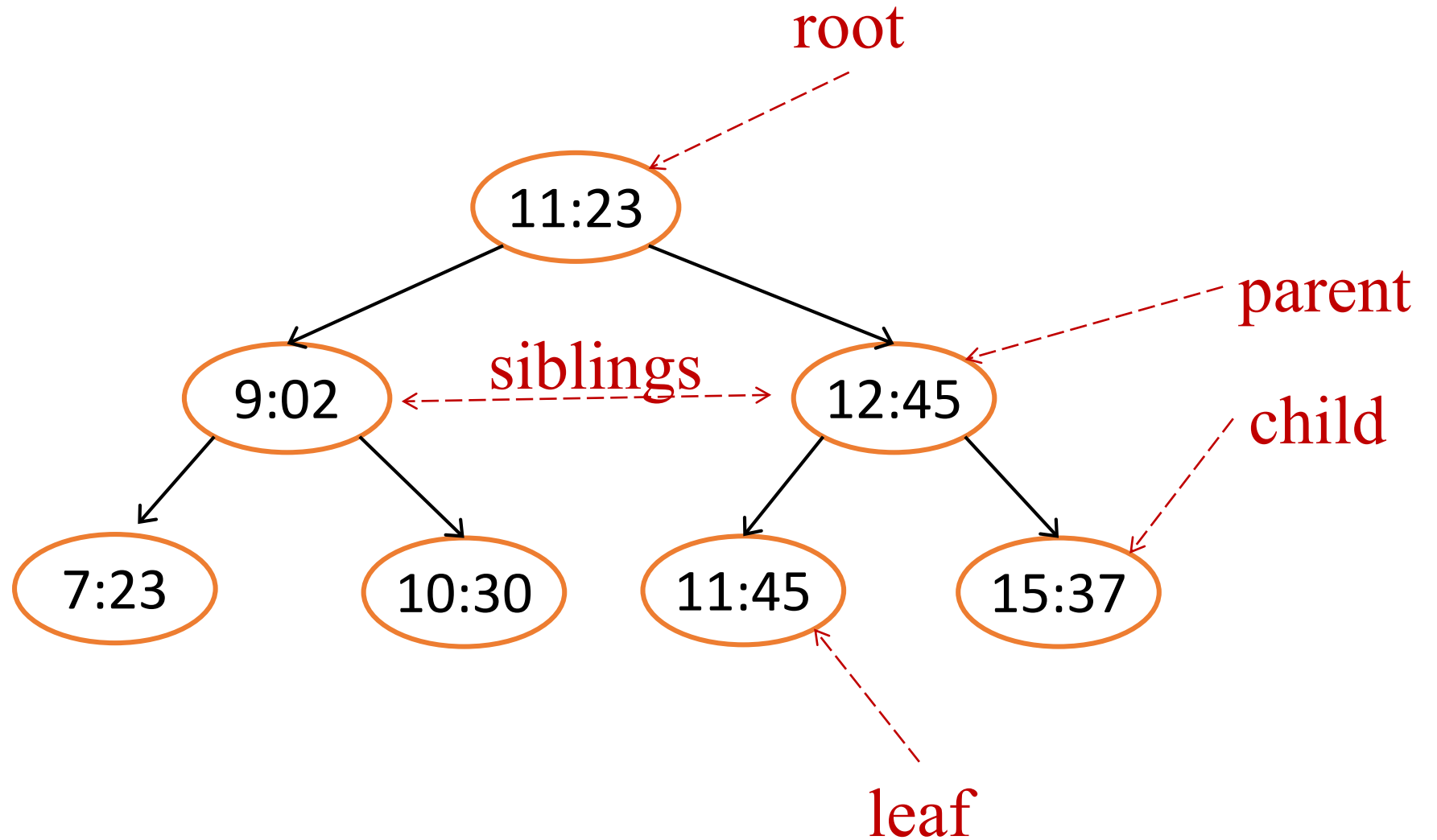
Dictionary Implementation

- Option 1: Sorted array
 - insert: add to middle of array --- $O(n)$
 - search : binary search through array --- $O(\log n)$
- Option 2: Linked list
 - insert: add to middle of array --- $O(n)$
 - search : no binary search in array --- $O(n)$

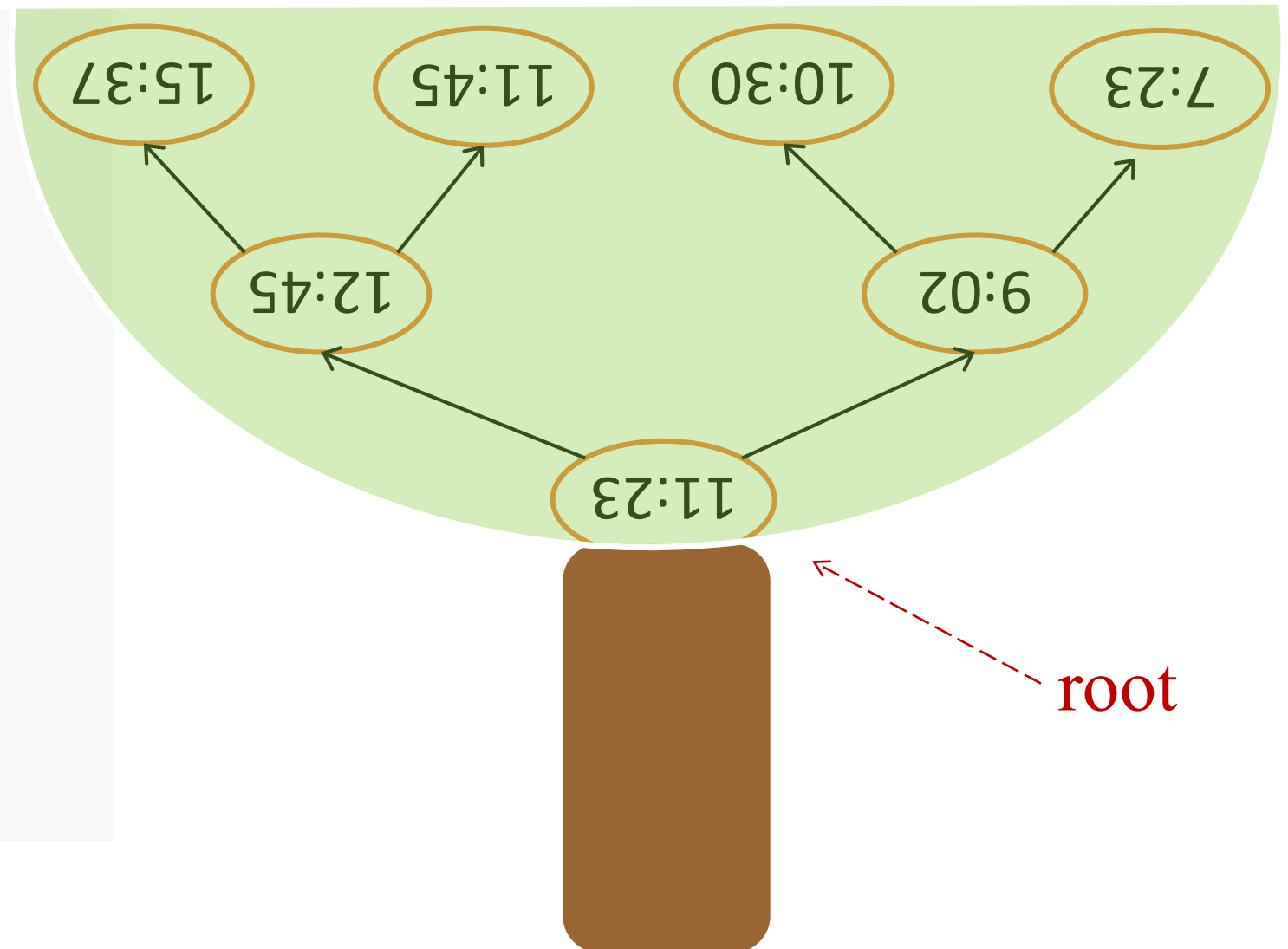
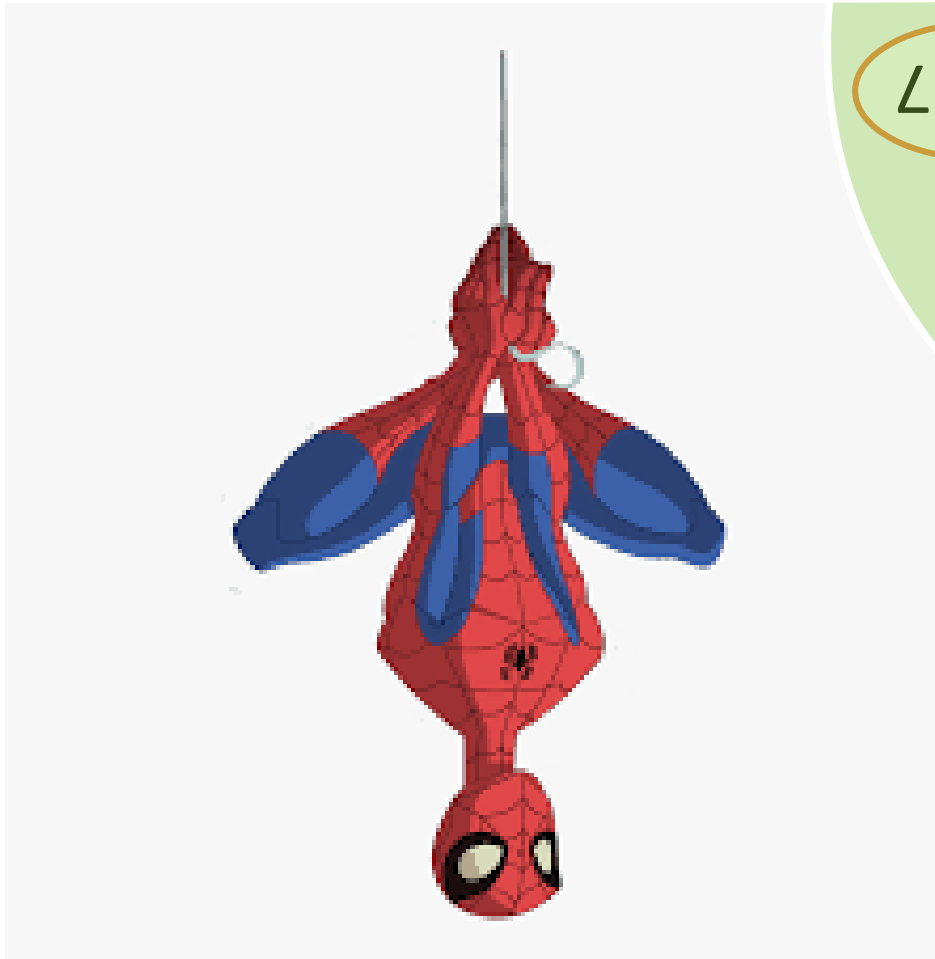
Tree!



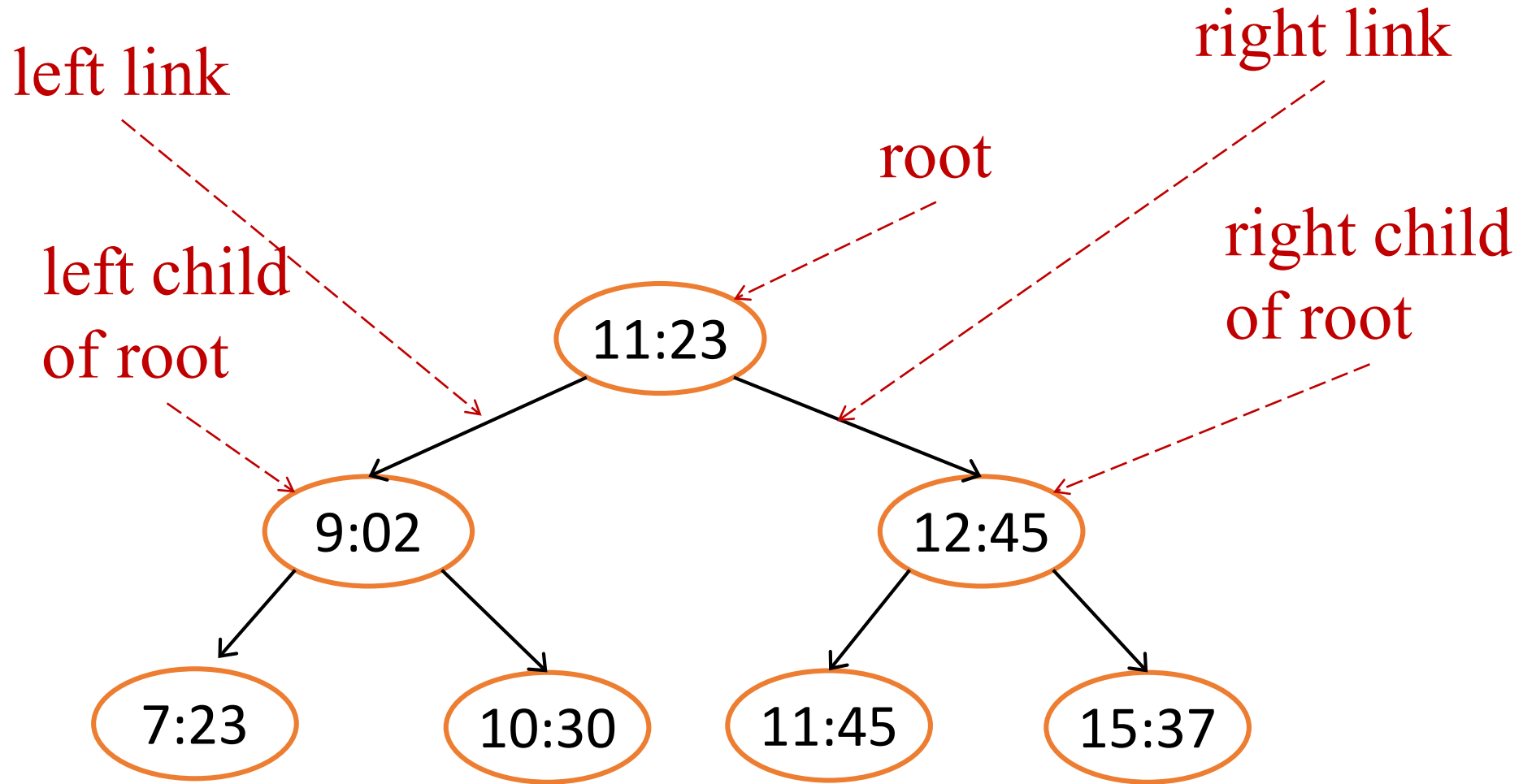
Terminology



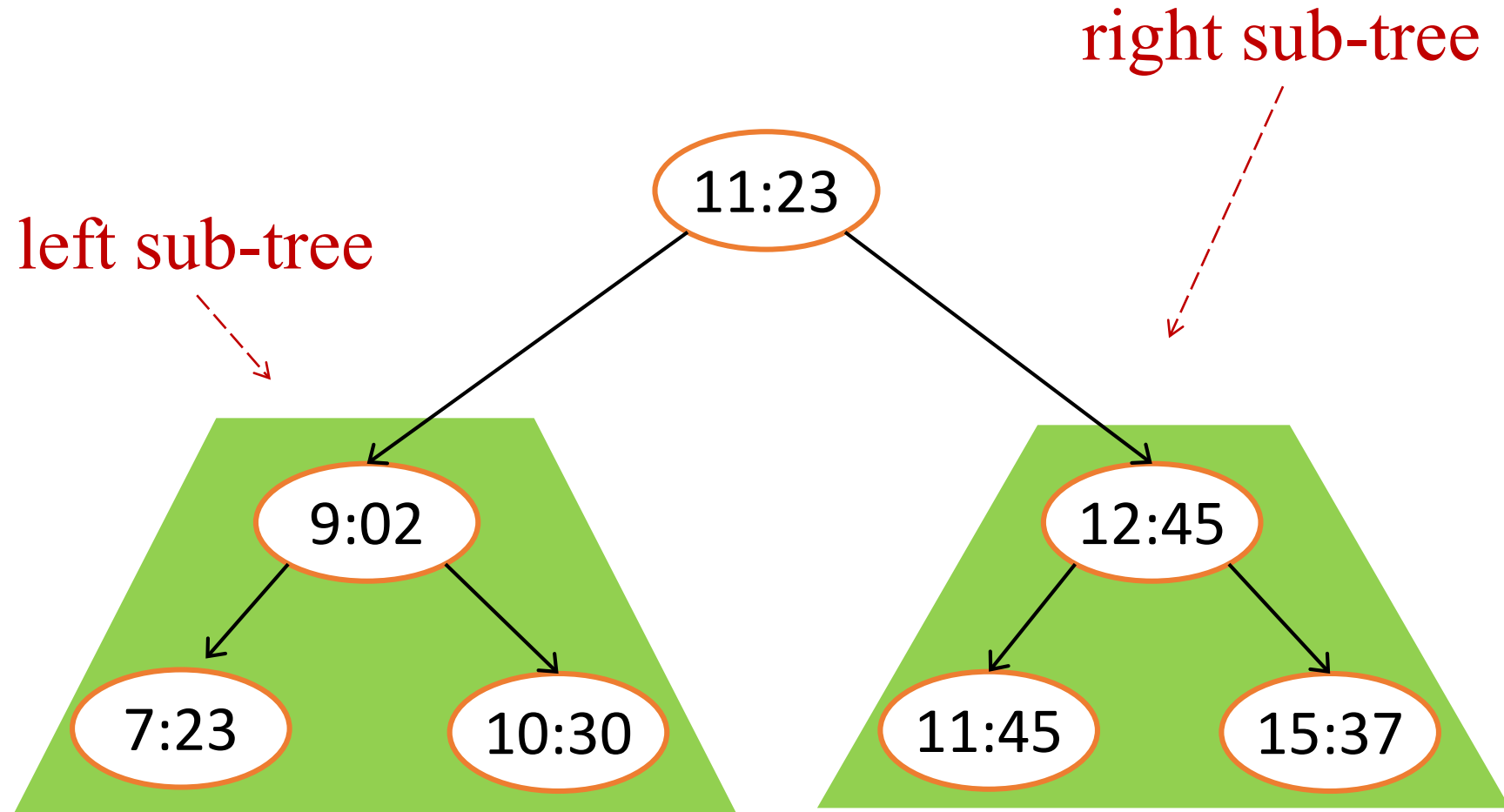
Look at it like Spiderman



Terminology

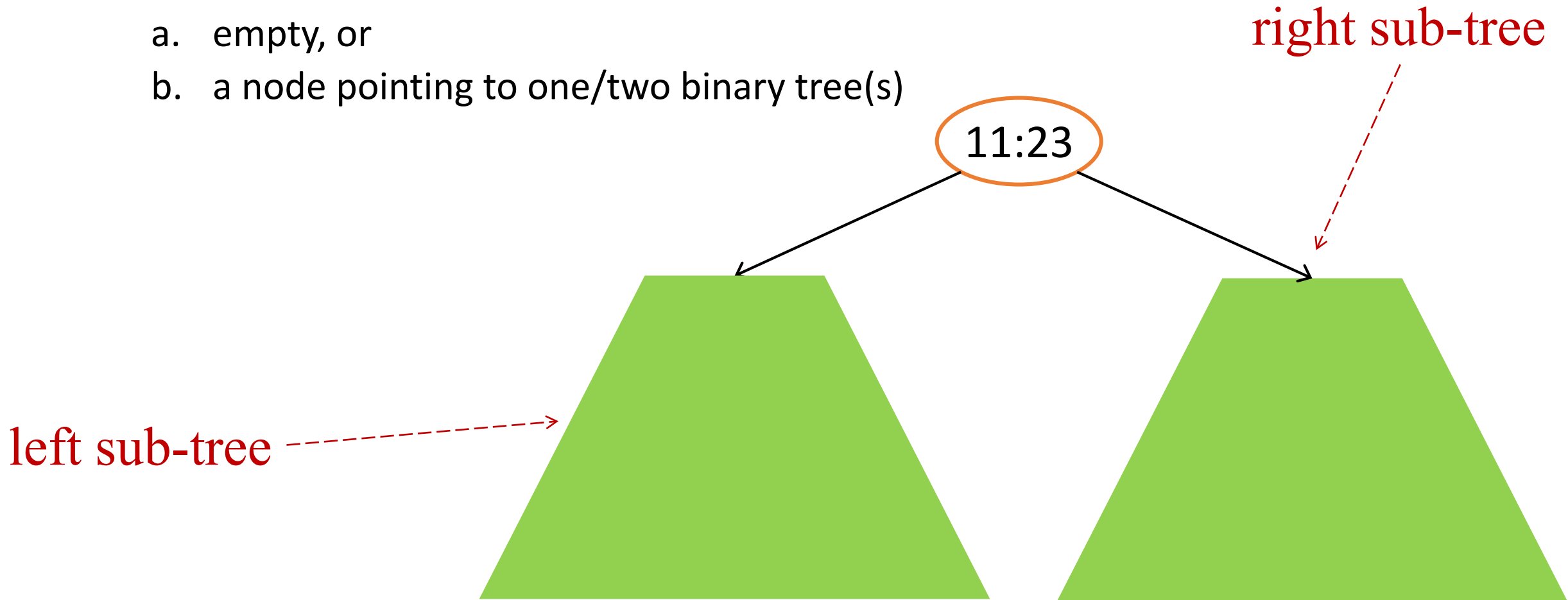


Terminology



Binary Tree (Recursive) Definition

- A binary tree is either:
 - a. empty, or
 - b. a node pointing to one/two binary tree(s)



C++ Code

```
template <class T>
class TreeNode {
    private:
        T _item;
        TreeNode<T>* _left;
        TreeNode<T>* _right;

    public:
        TreeNode(T x)
        { _left = _right = NULL; _item = x; _height = 0; };

        friend BinarySearchTree<T>;
};
```

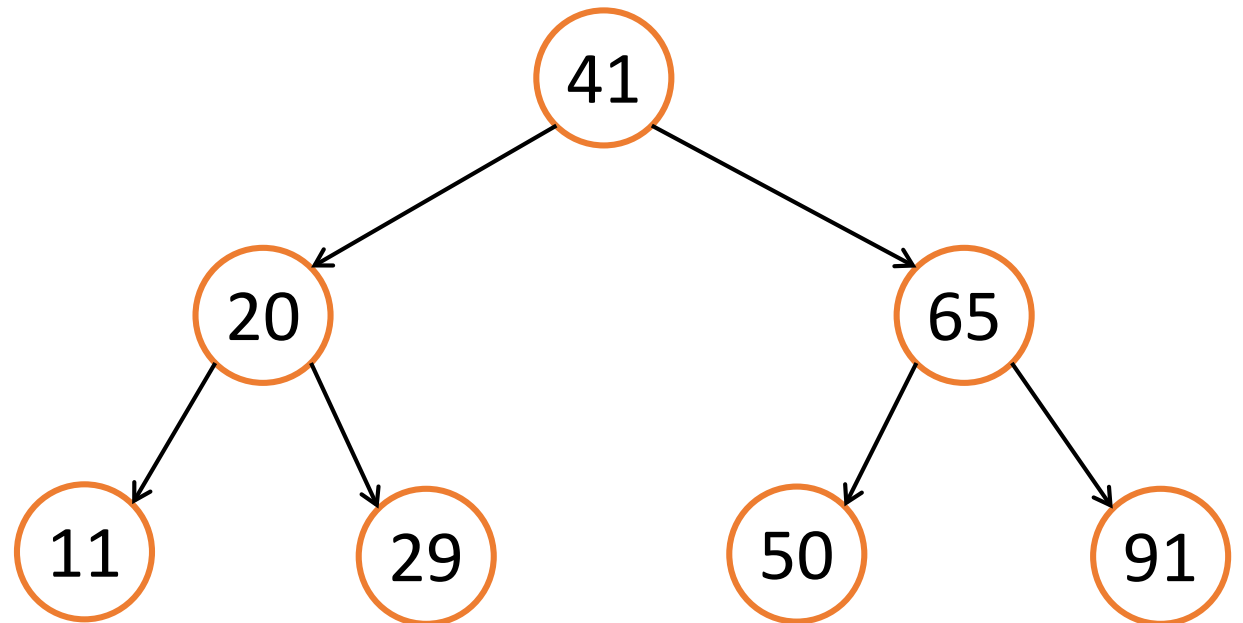
C++ Code

```
template <class T>
class BinarySearchTree {
    private:
        TreeNode<T>* _root;

    public:
        BinarySearchTree() { _root = NULL; }
        void insert(T);
};
```

Binary Search Trees (BST)

- BST Property:
 - all in left sub-tree $<$ key $<$ all in right sub-right

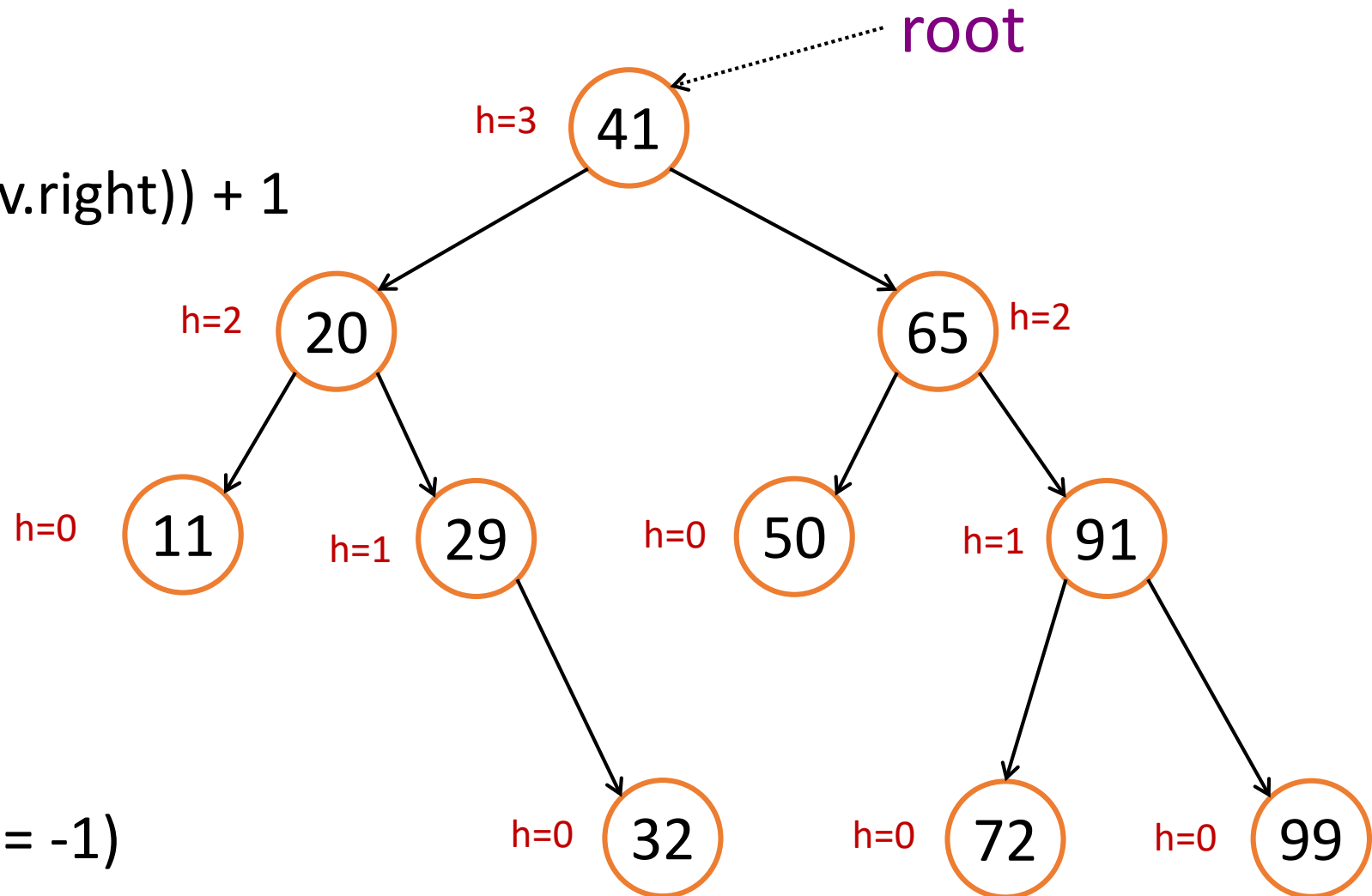


Basic Operations

- height
- search, insert
- searchMin, searchMax

Heights


- For a tree node v
- $h(v) = 0$ if v is a leaf
- $h(v) = \max(h(v.\text{left}), h(v.\text{right})) + 1$




- (For simplicity: $h(\text{null}) = -1$)

Computing the Height of a Node

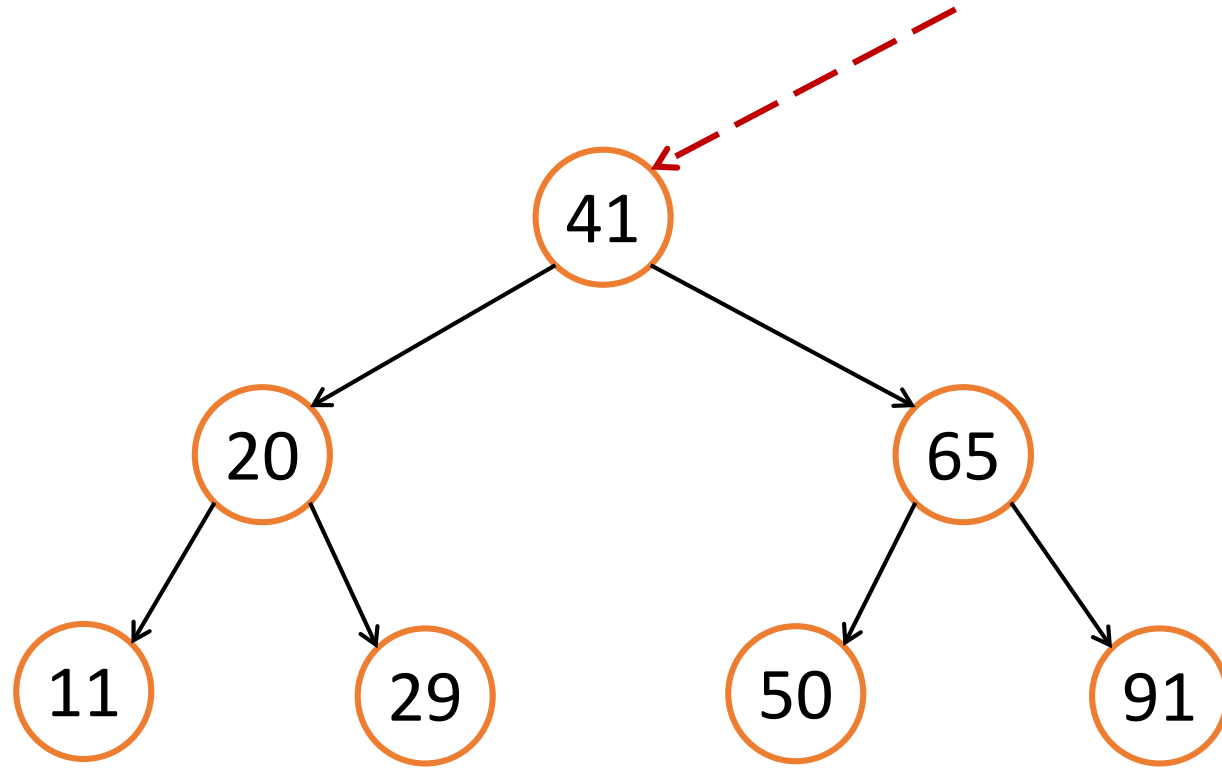
```
int TreeNode<T>::height() {  
    int leftHeight = -1;  
    int rightHeight = -1;  
  
    if (_left != null)  
        leftHeight = _left->height();  
    if (_right != null)  
        rightHeight = _right->height();  
  
    return max(leftHeight, rightHeight) + 1;  
}
```

 **max of subtrees**

 **add 1**

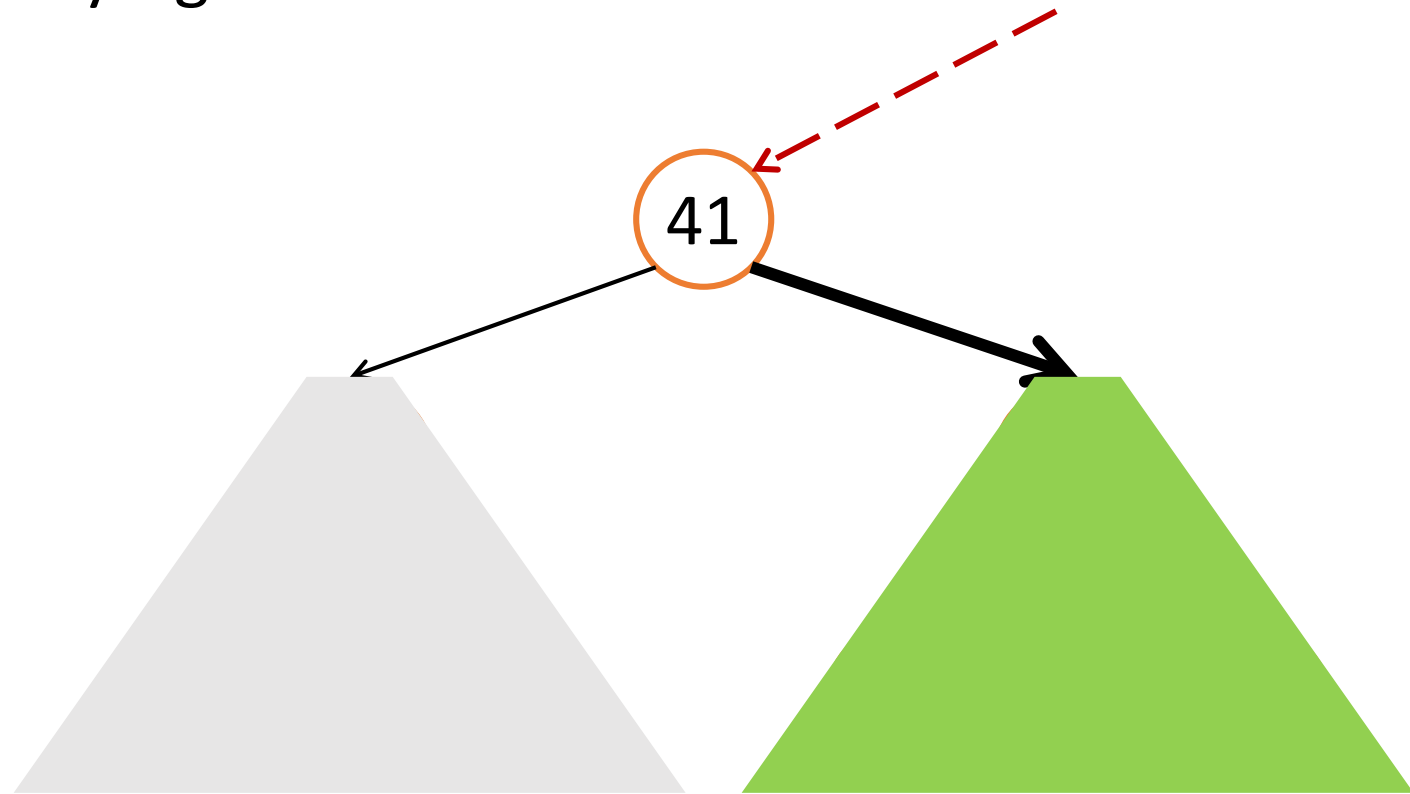
Search for the Maximum

- Starting from the root?



Idea

- For a node v , I know my right subtree contains elements that is greater than v



Searching For Maximum

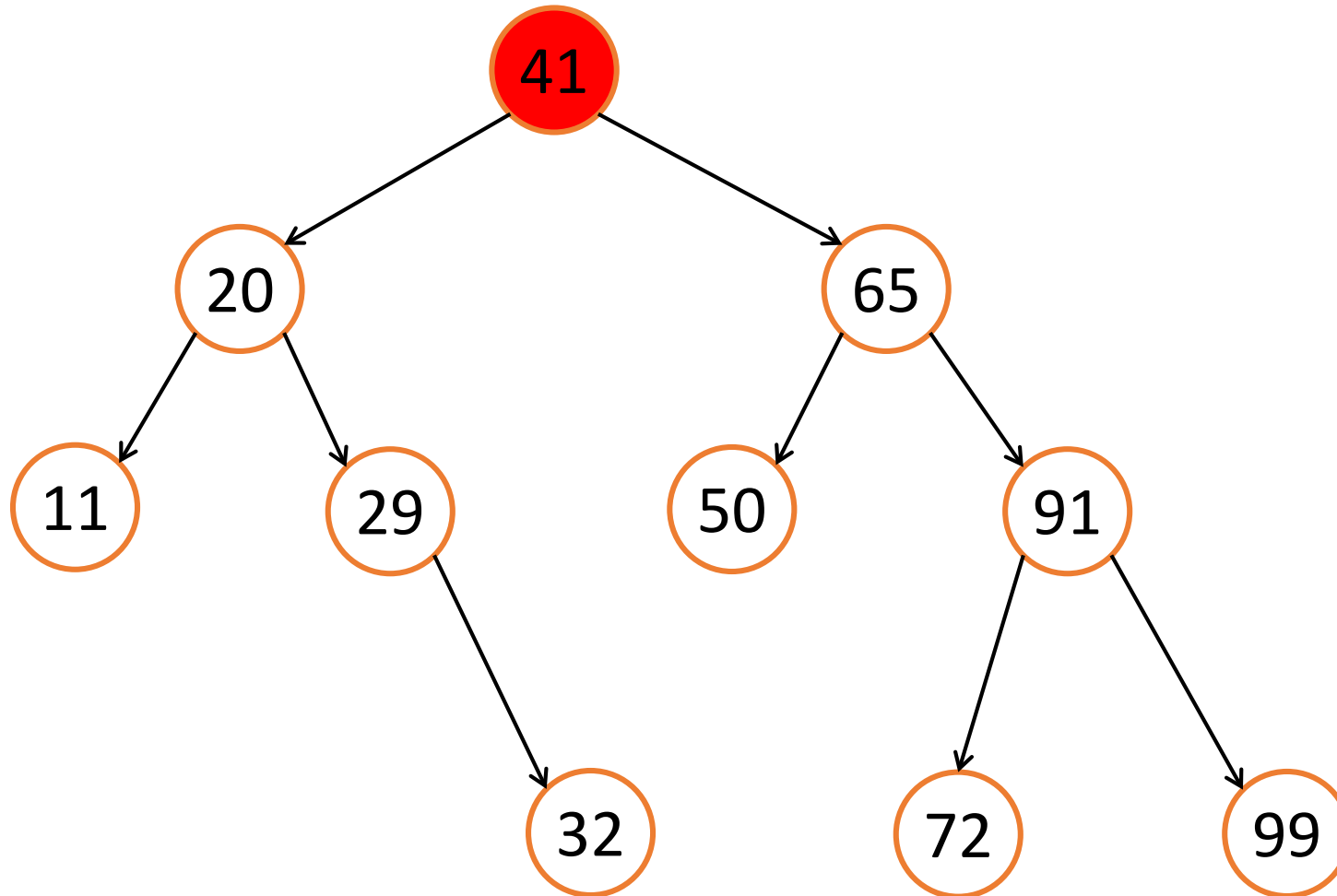
```
template <class T>
T BinarySearchTree<T>::searchMax() {

    TreeNode<T>* current = _root;

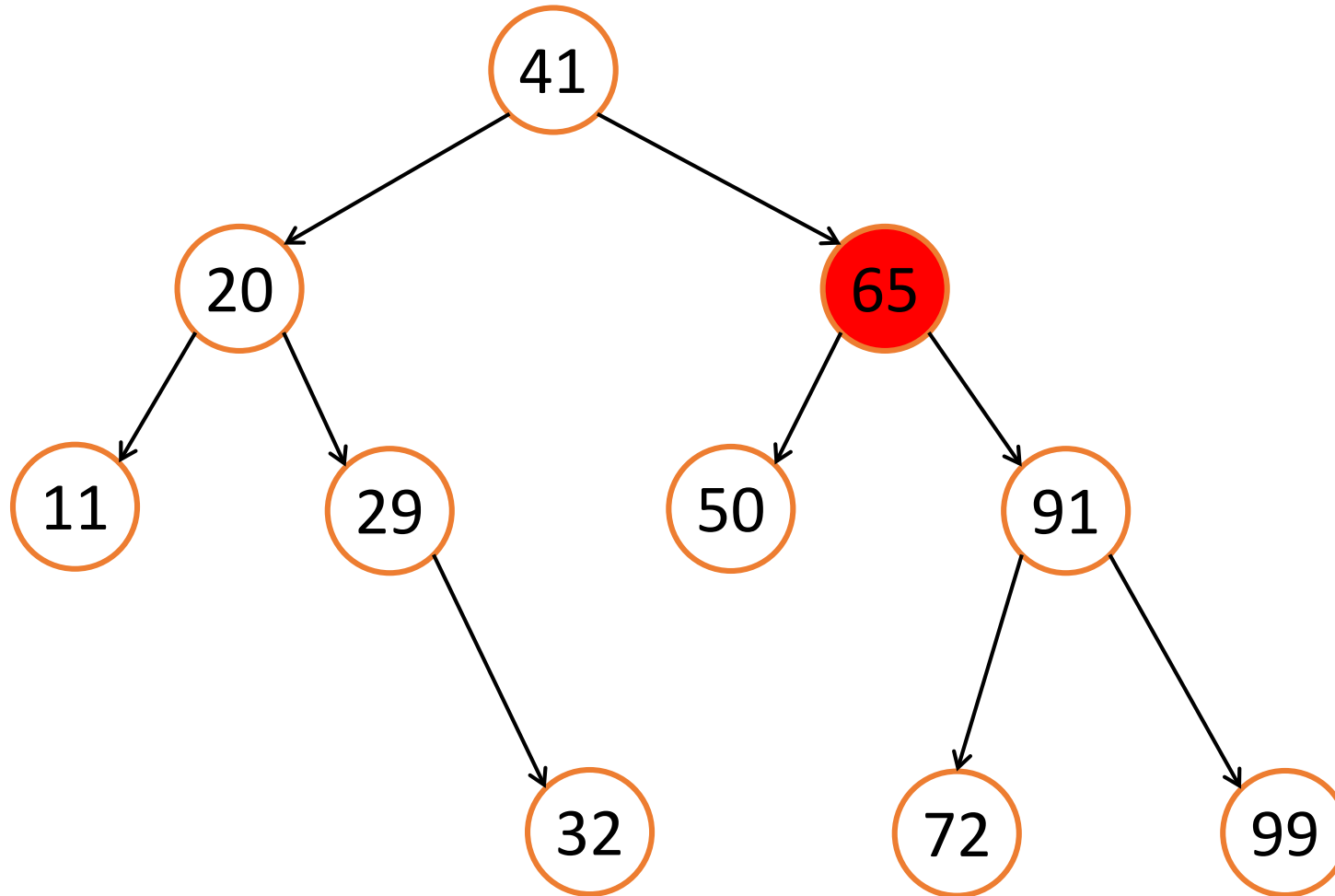
    while (current->_right)
        current = current->_right;
    return current->_item;

}
```

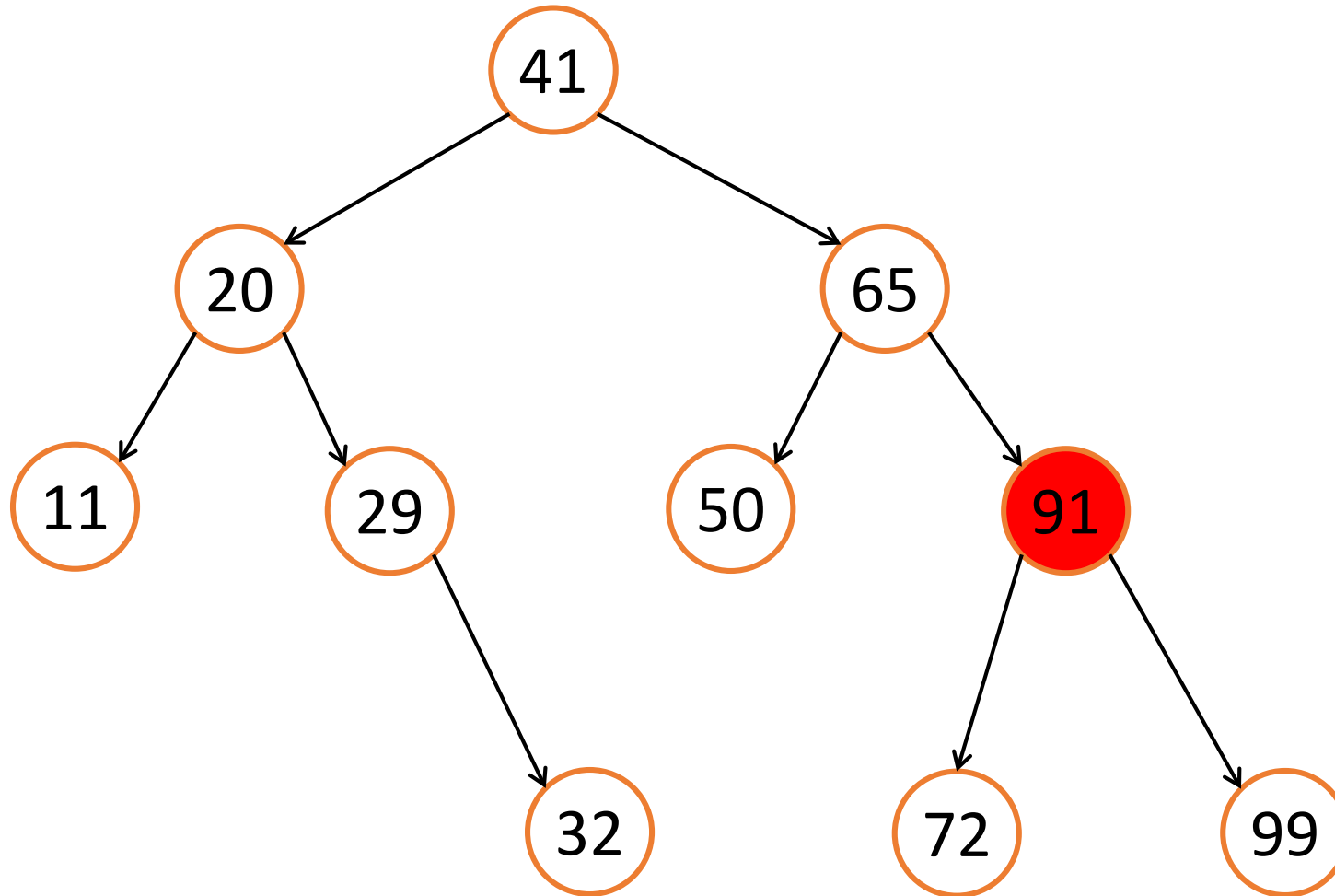
Search Max



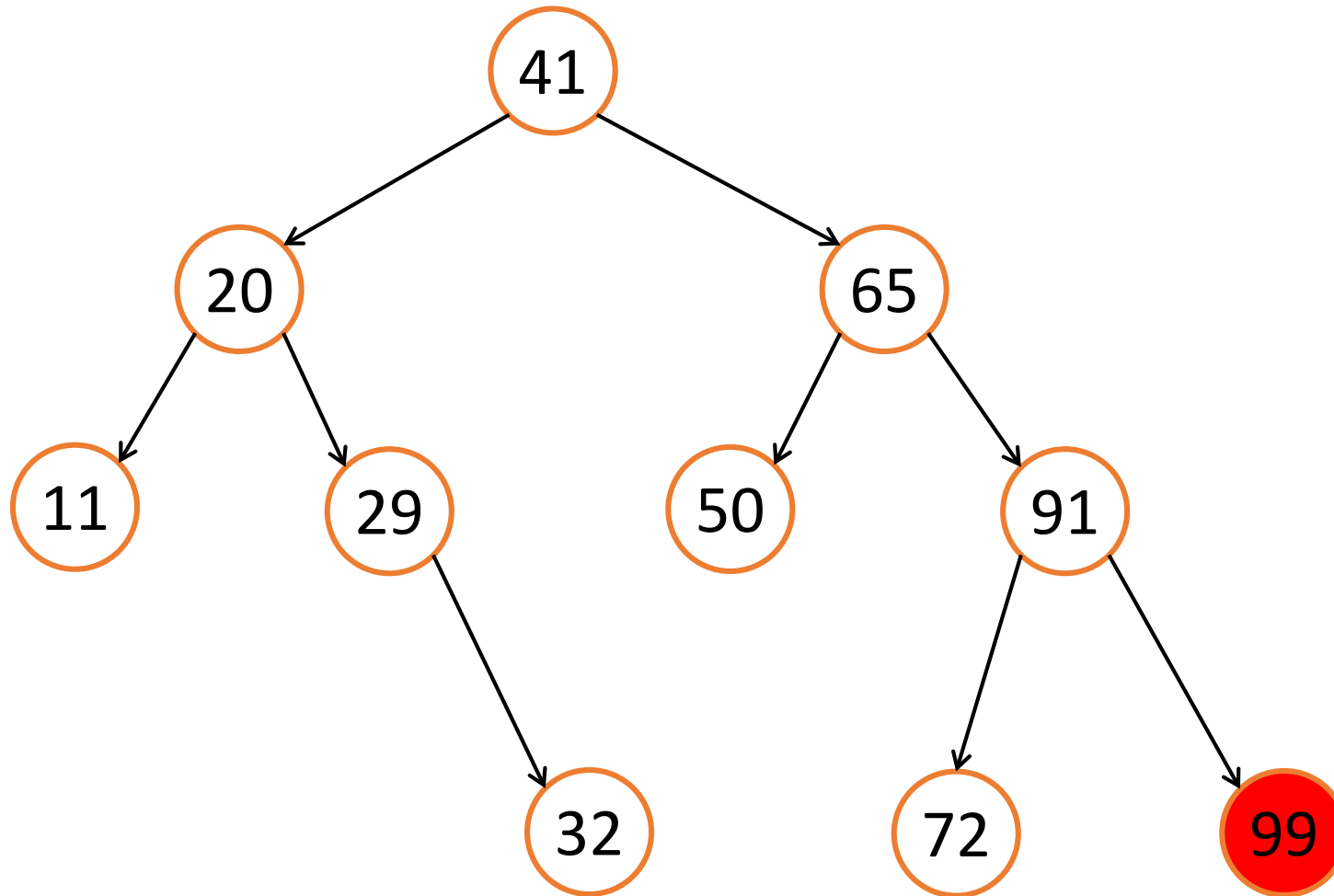
Search Max



Search Max



Search Max



Searching For Maximum

```
template <class T>
T BinarySearchTree<T>::searchMax() {

    TreeNode<T>* current = _root;

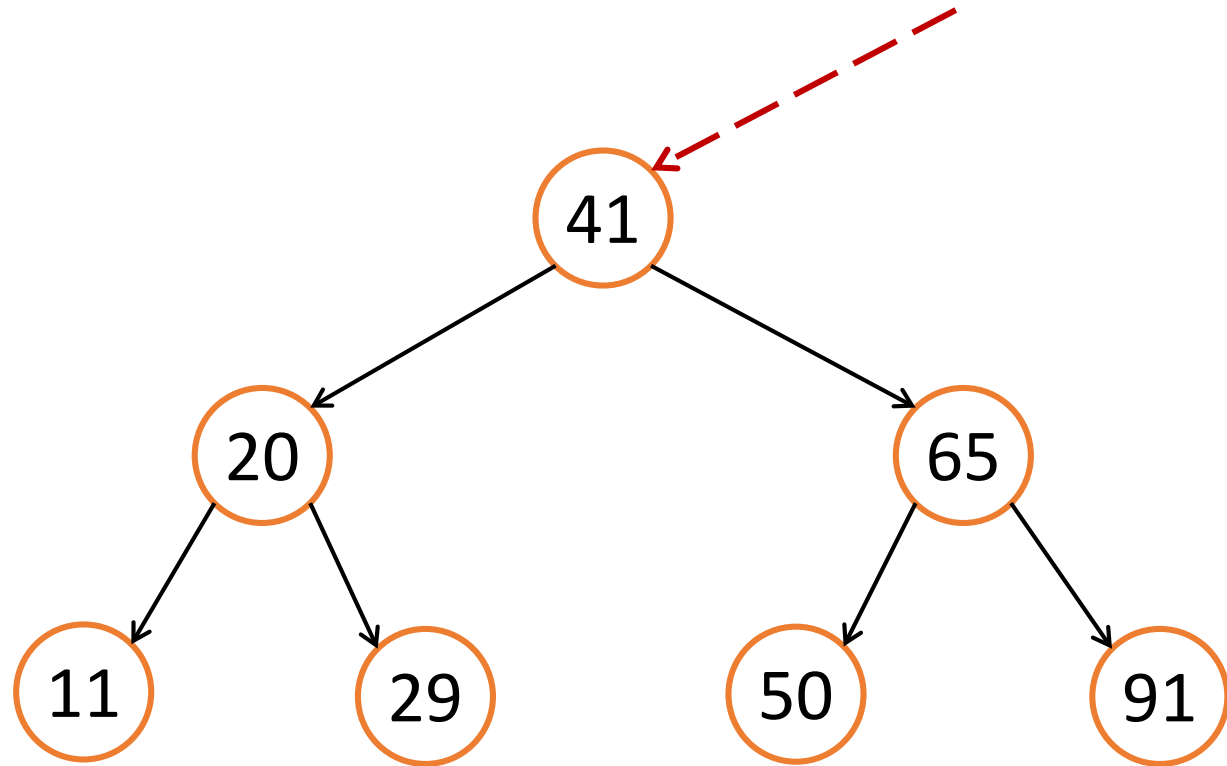
    while (current->_right)
        current = current->_right;
    return current->_item;

}
```

Search for
Minimum?

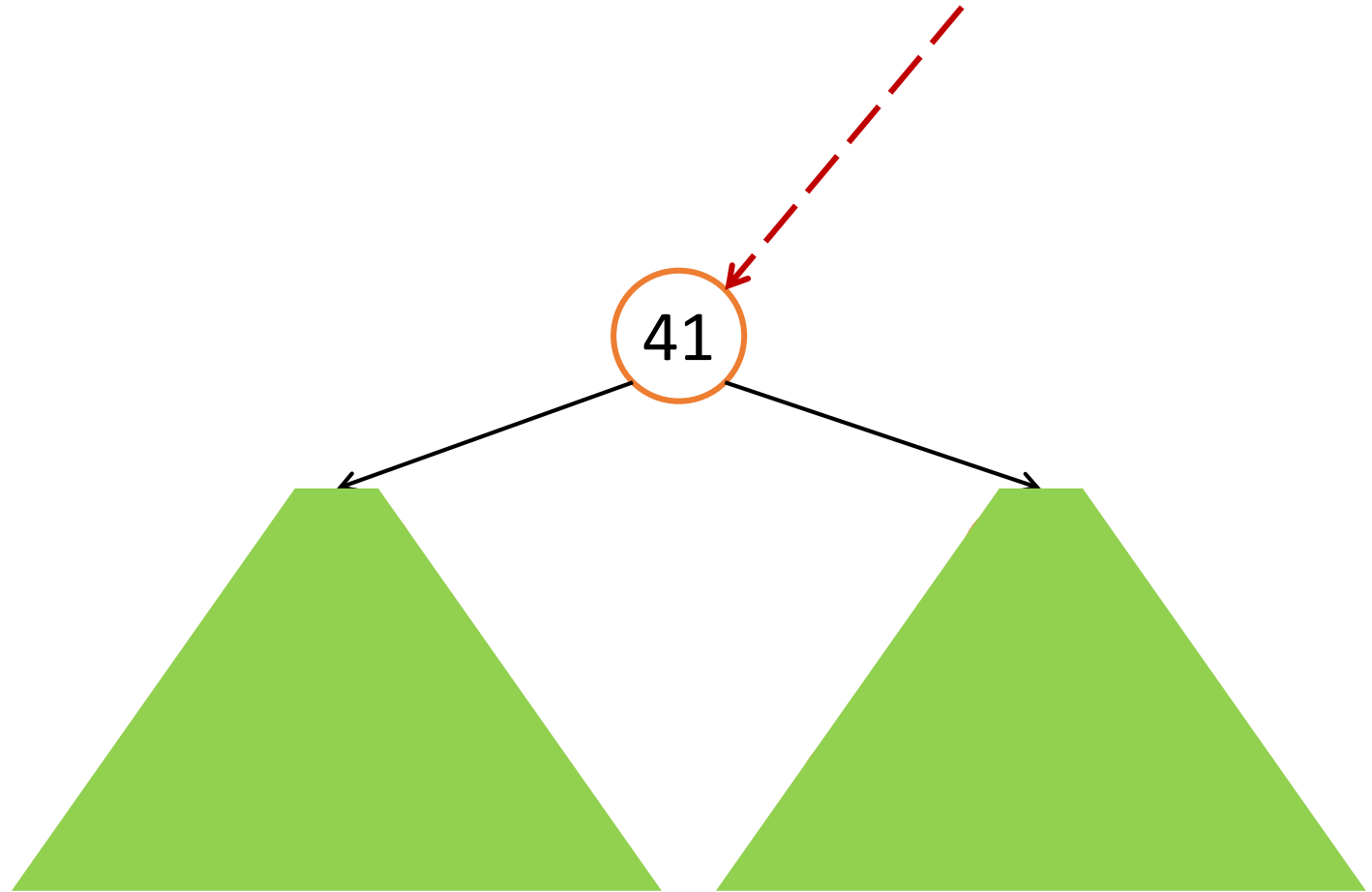
Search for a particular key

- Starting from the root
- Is “25” in the tree?



Go Left or Go Right?

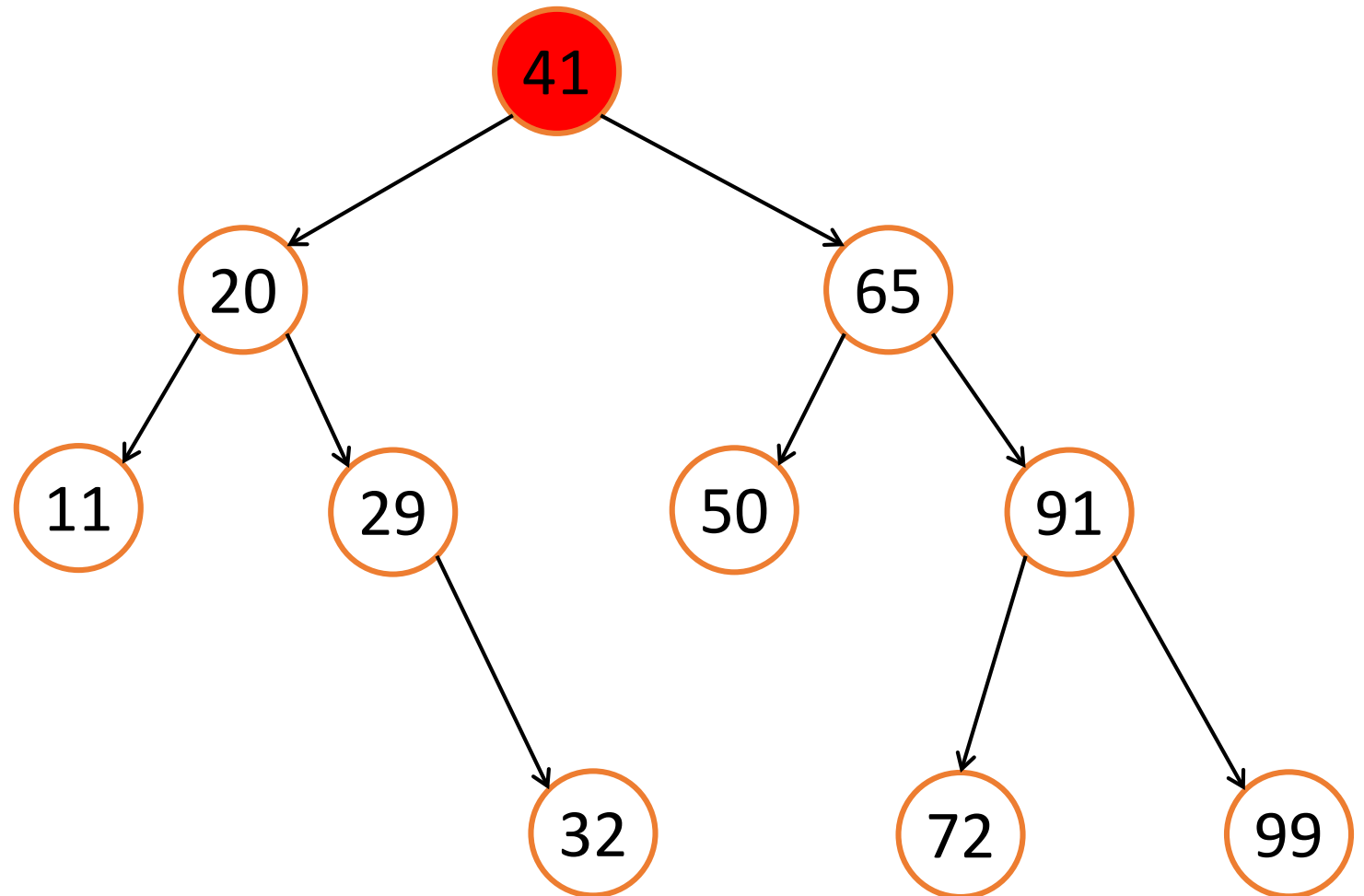
- $25 < 41$



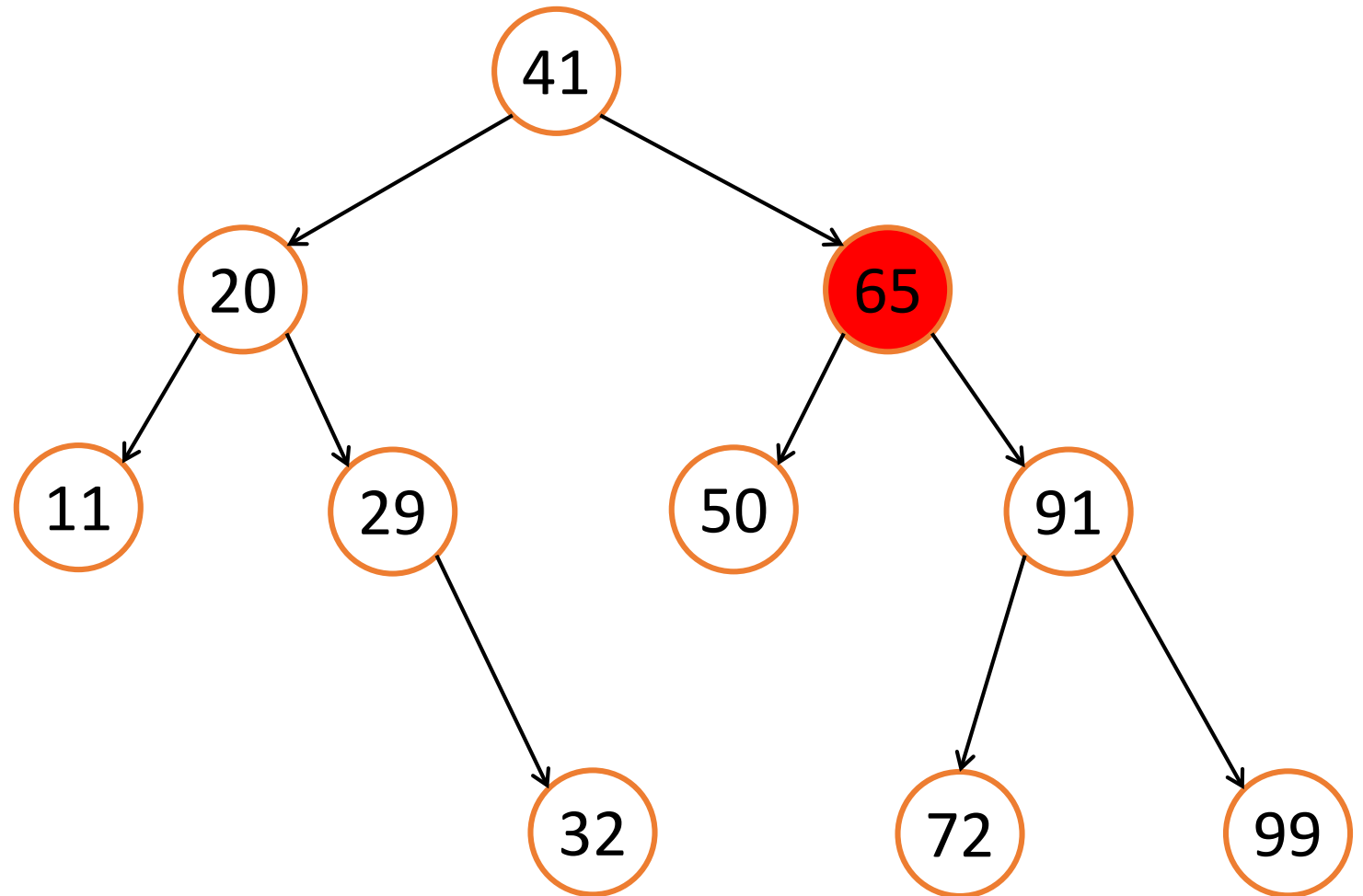
```
template <class T>
bool BinarySearchTree<T>::exist(T x) {
    TreeNode<T>* current = _root;

    while (current) {
        if (current->_item == x)
            return true;
        else if (x > current->_item )
            current = current->_right;
        else
            current = current->_left;
    }
    return false;
}
```

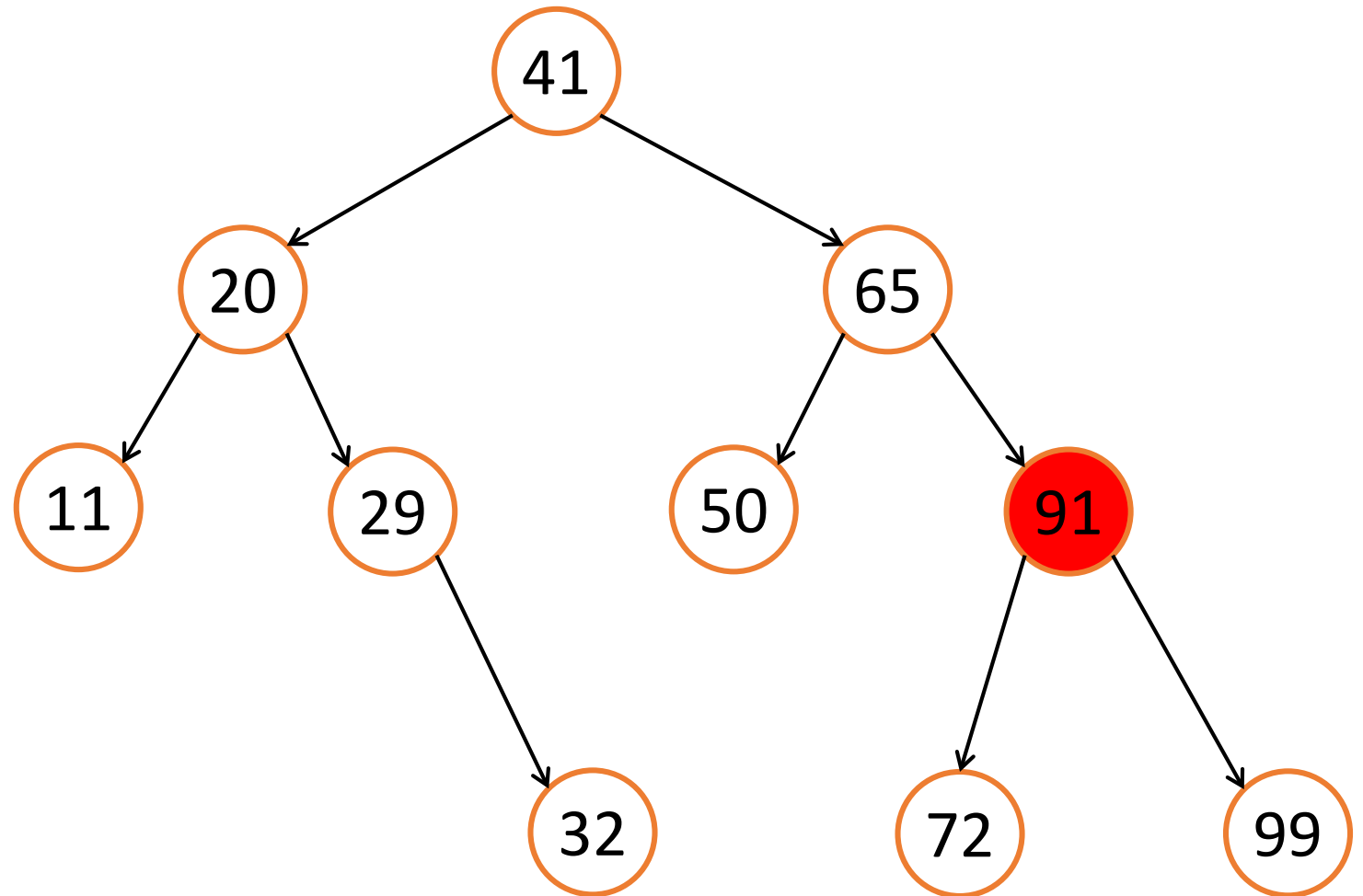
Example: exist(72)



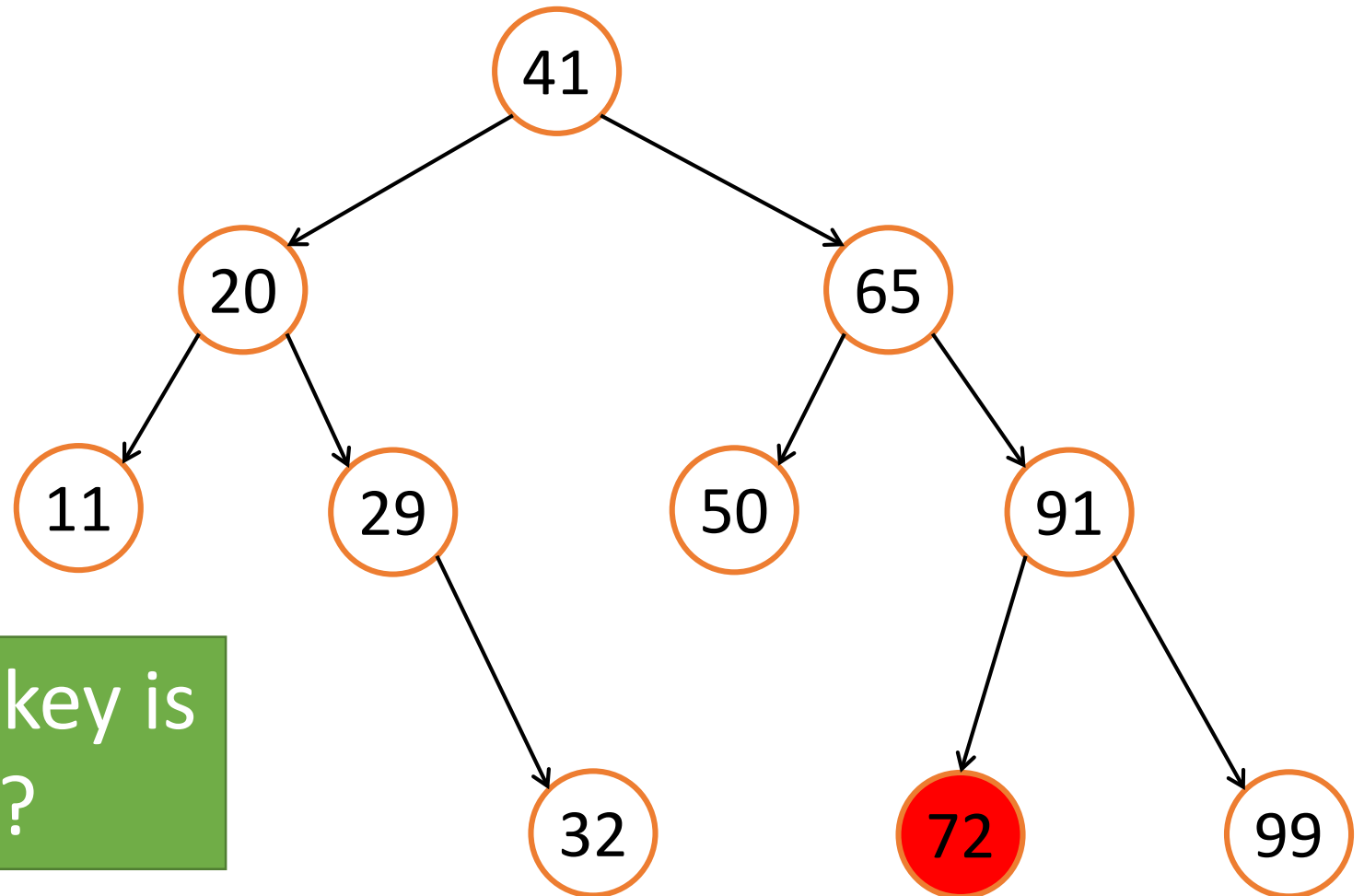
Example: exist(72)



Example: exist(72)



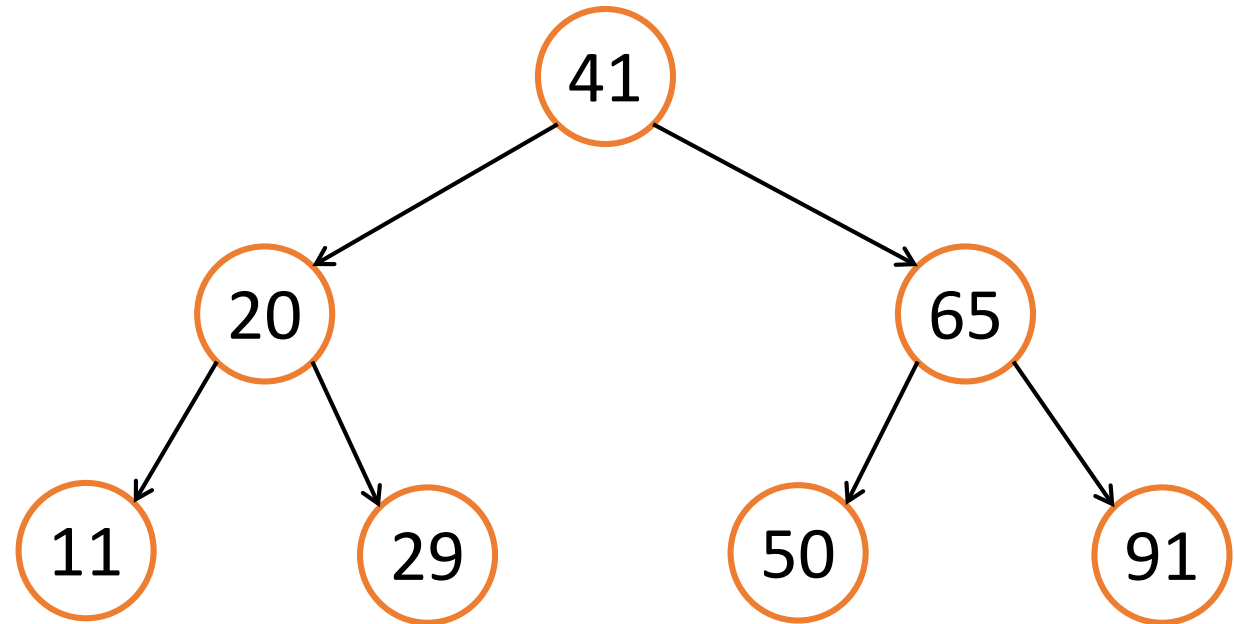
Example: exist(72)



What happen if the key is
NOT in the tree?

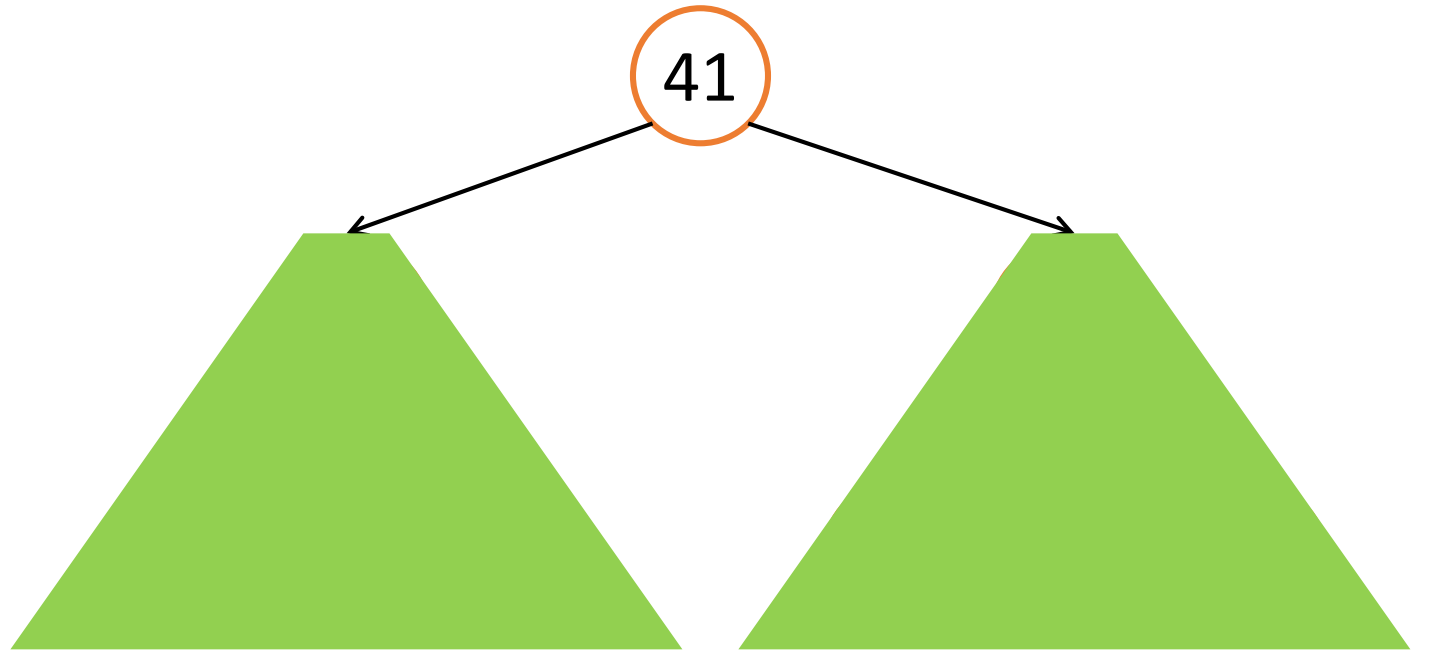
Inserting a Key

- Inserting 25



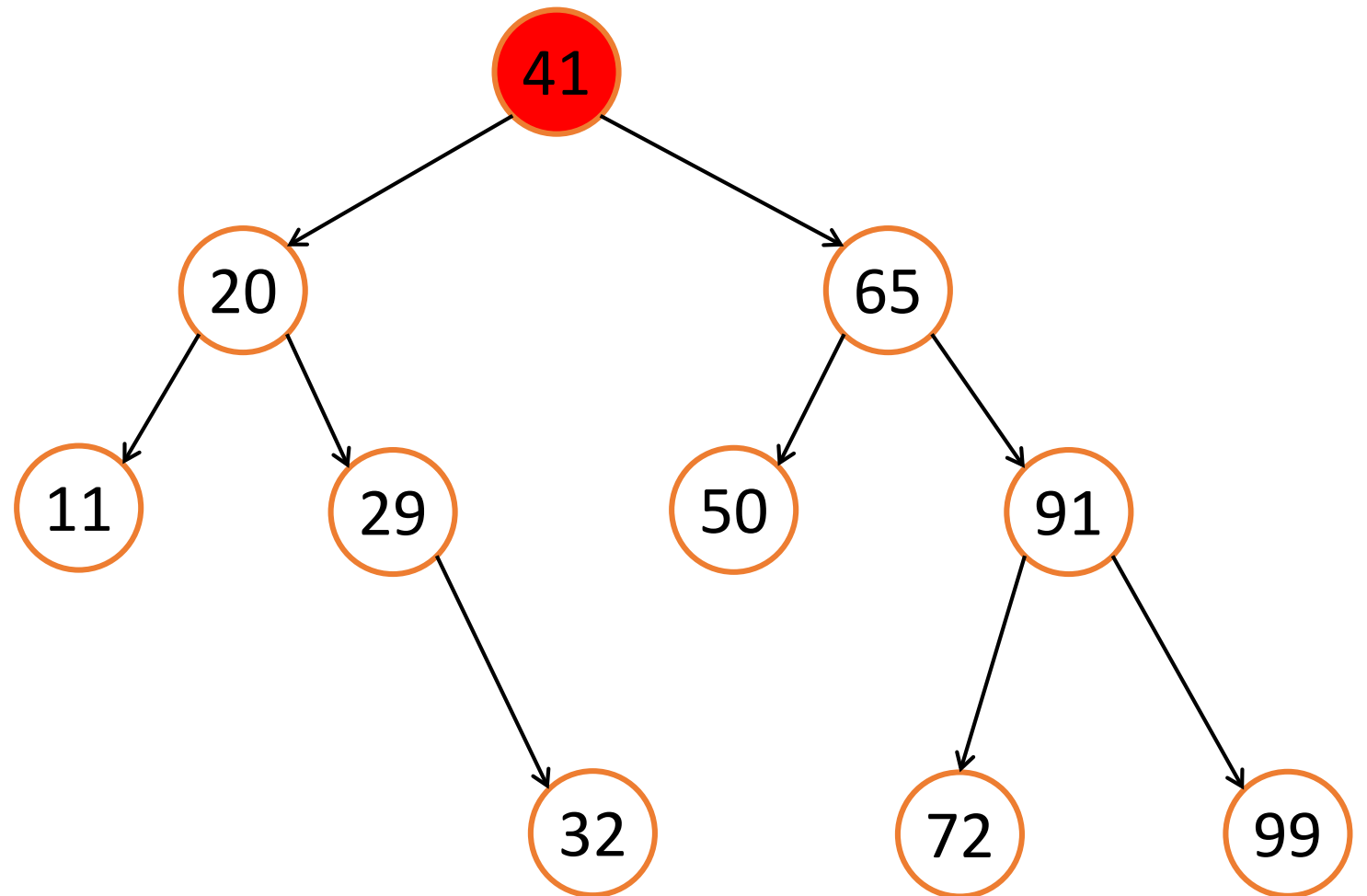
Inserting 25

- Go left or right?

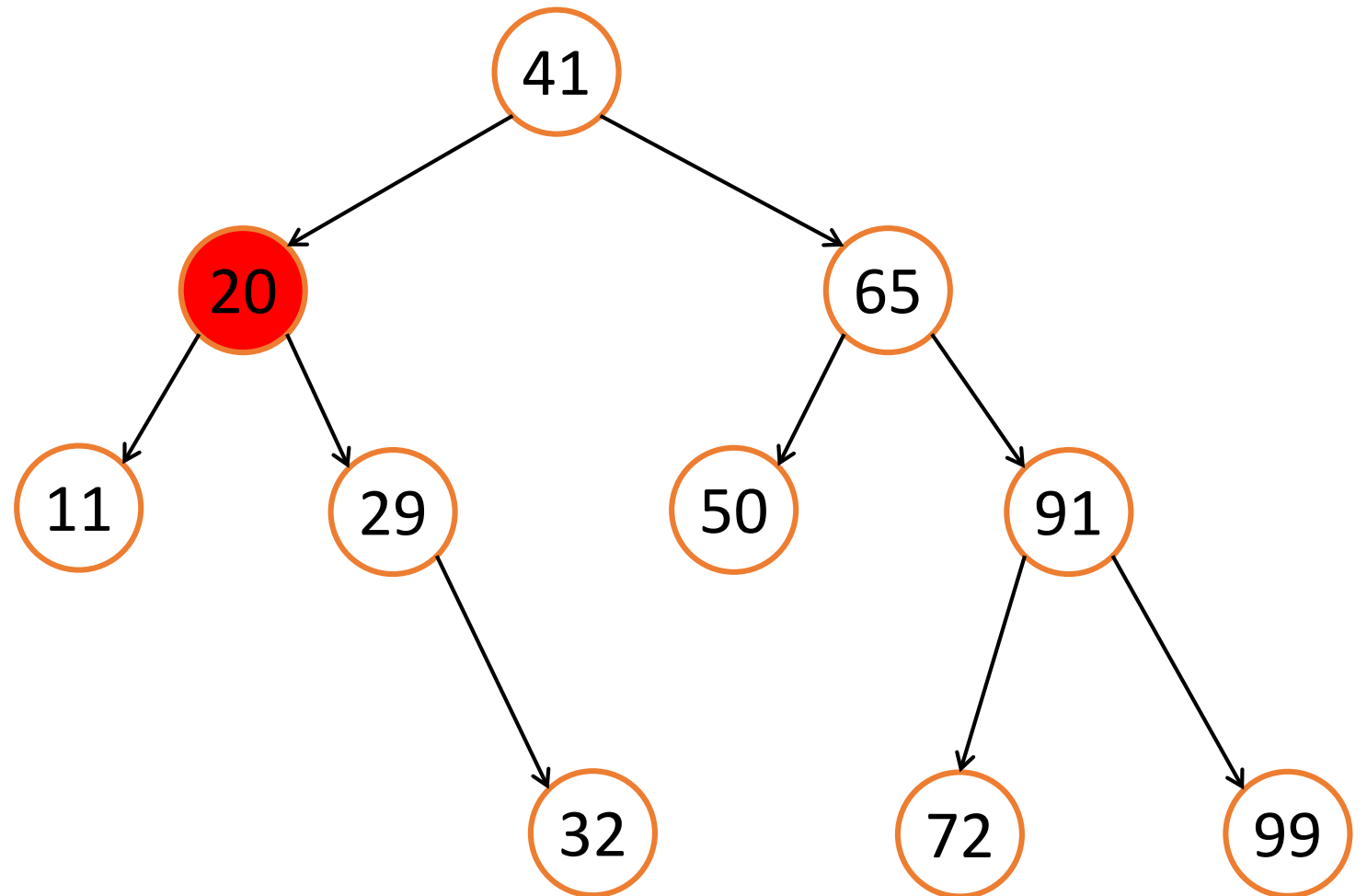


```
template <class T>
TreeNode<T>* BinarySearchTree<T>::insert
    (TreeNode<T>* current, T x)
{
    if (current->_item > x) {
        if (current->_left)
            current->_left = insert(current->_left, x);
        else
            current->_left = new TreeNode<T>(x);
    } else if (x > current->_item) {
        // try it yourself
    }
    else return current;
}
```

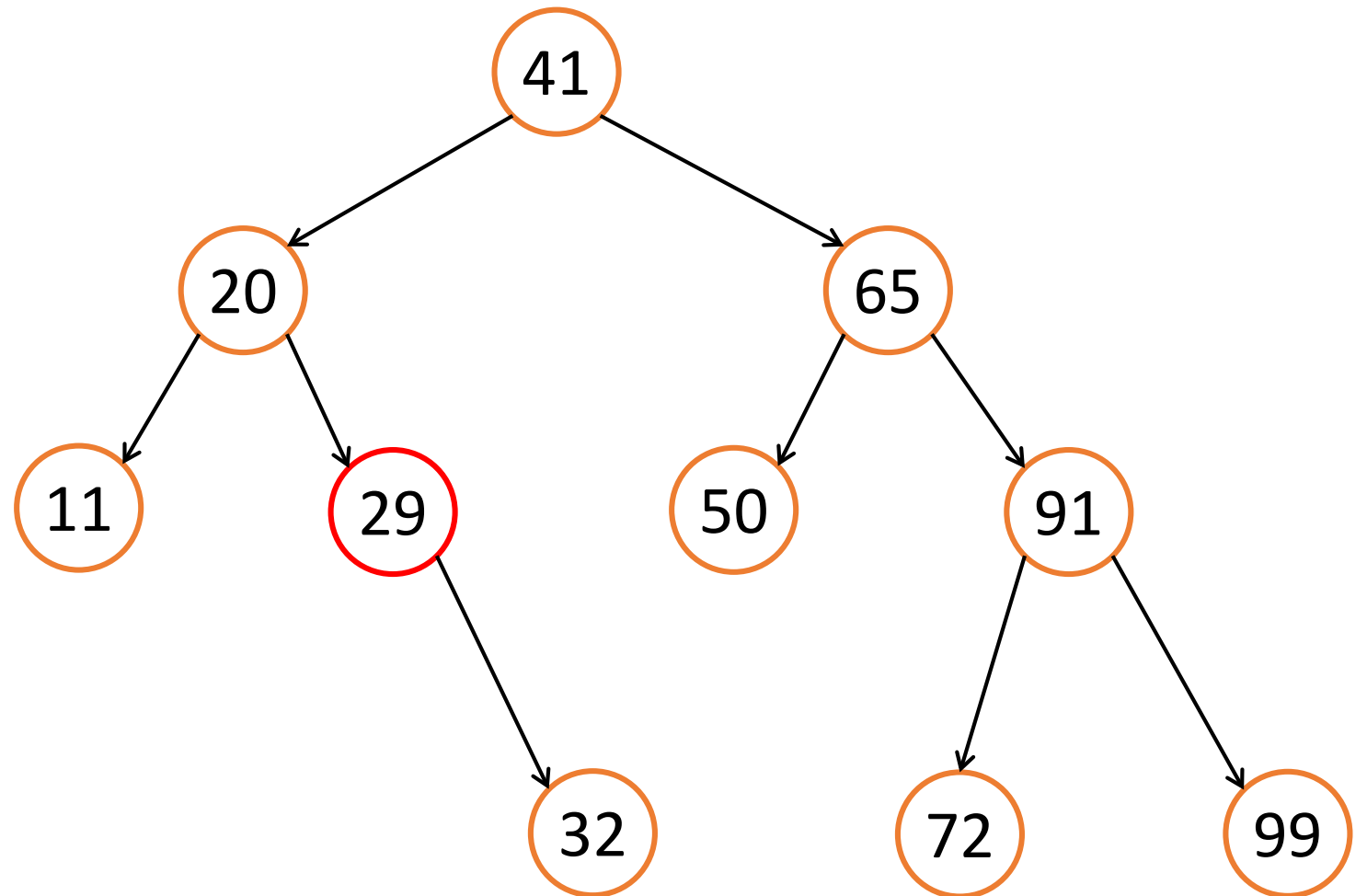
Insert(27)



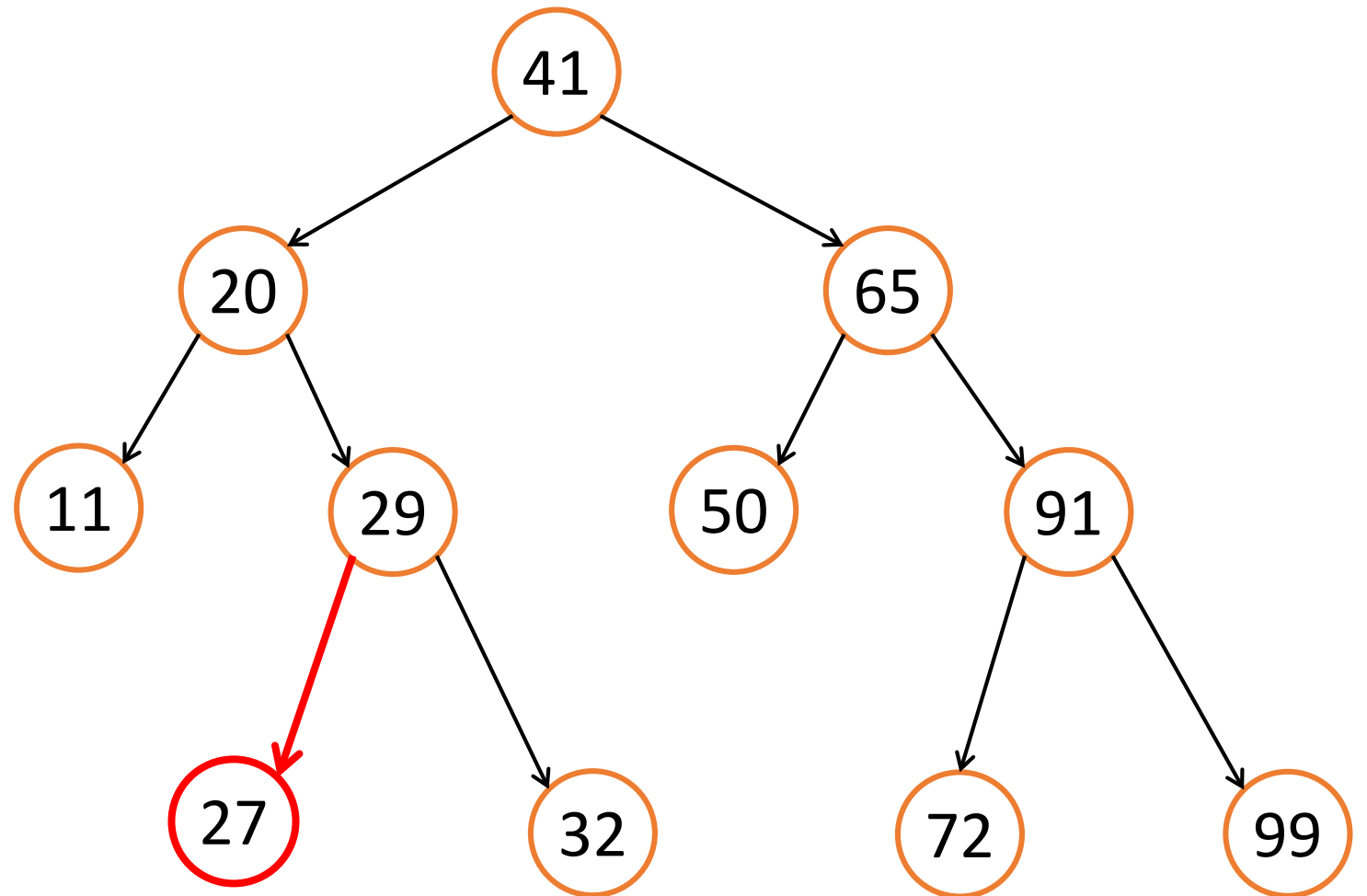
Insert(27)



Insert(27)

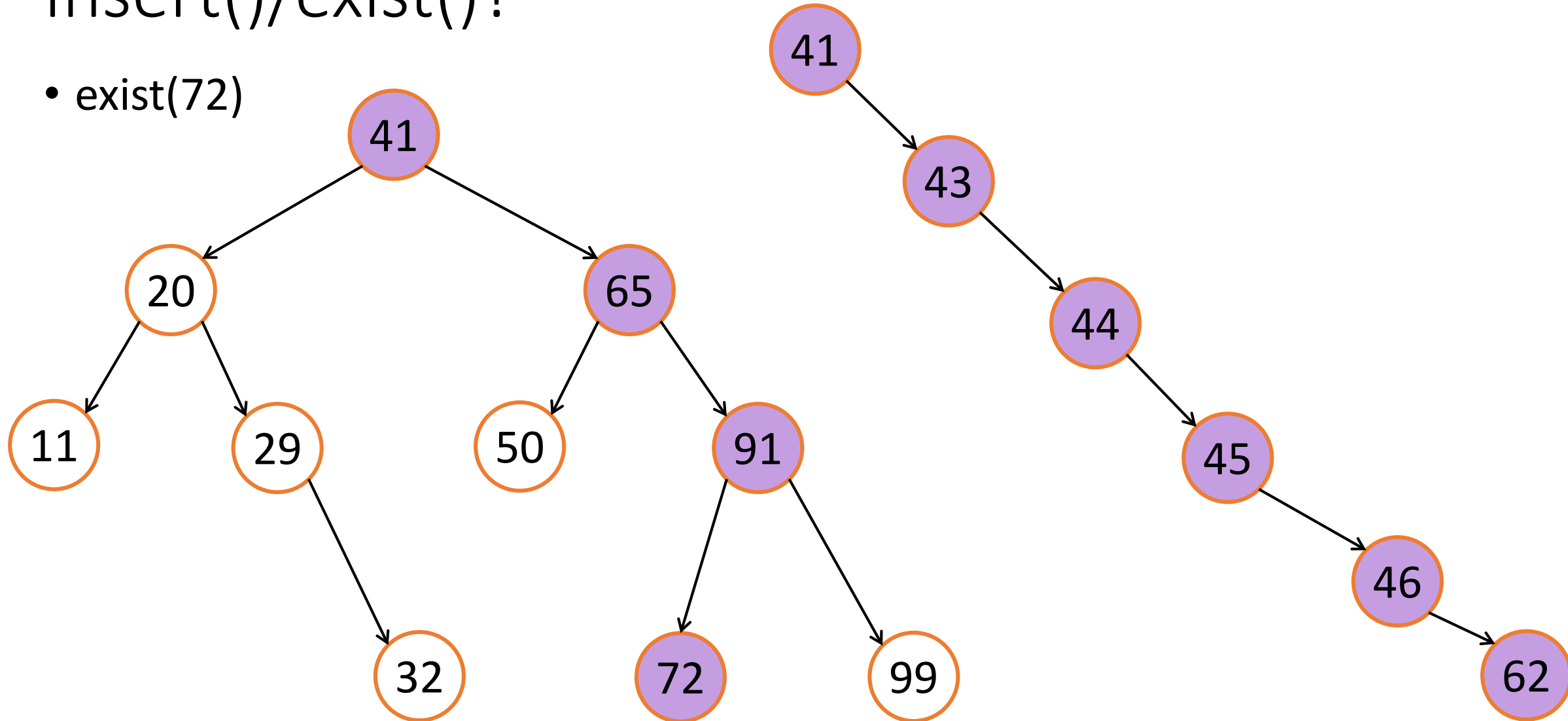


Insert(27)



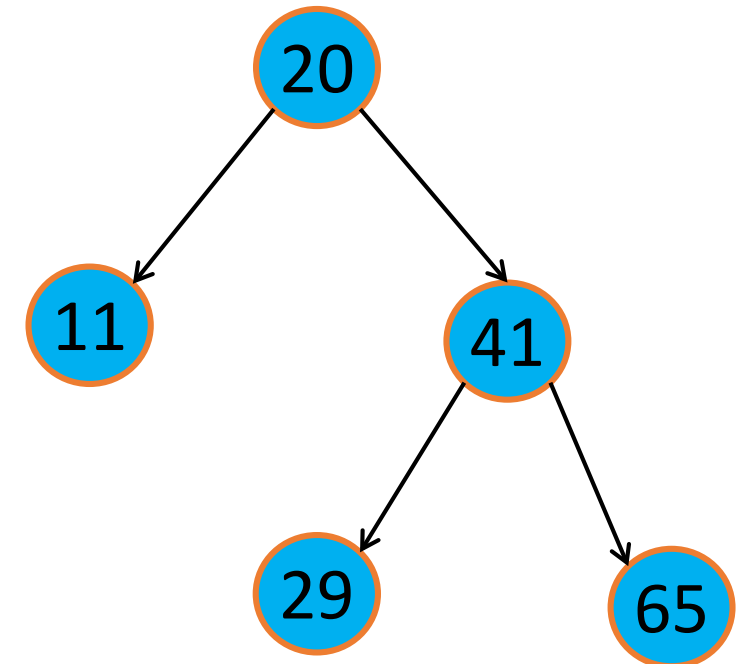
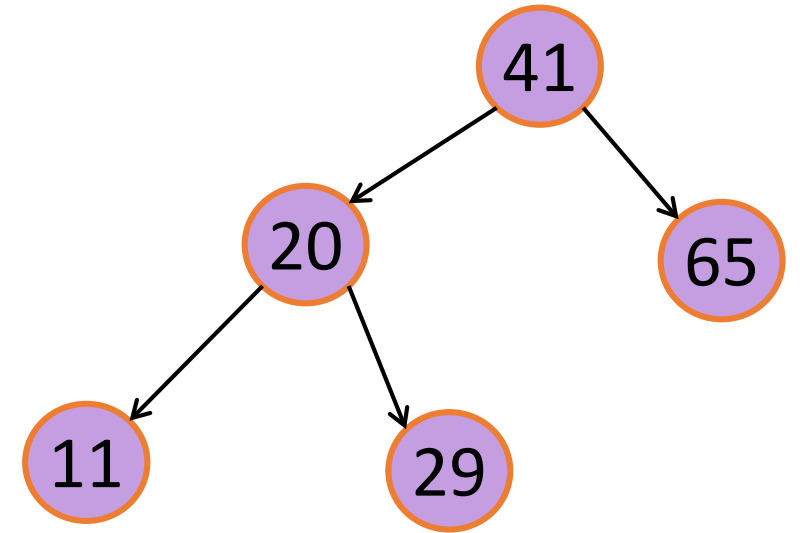
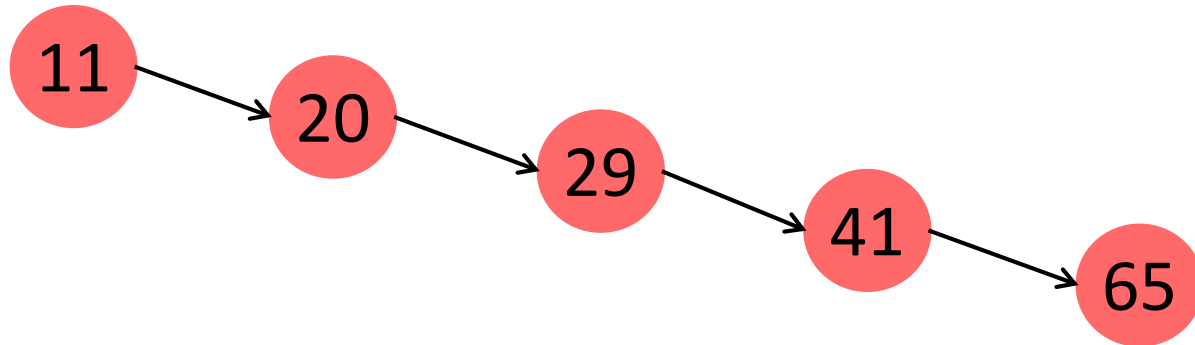
What is the Time Complexity for Insert()/exist()?

- exist(72)



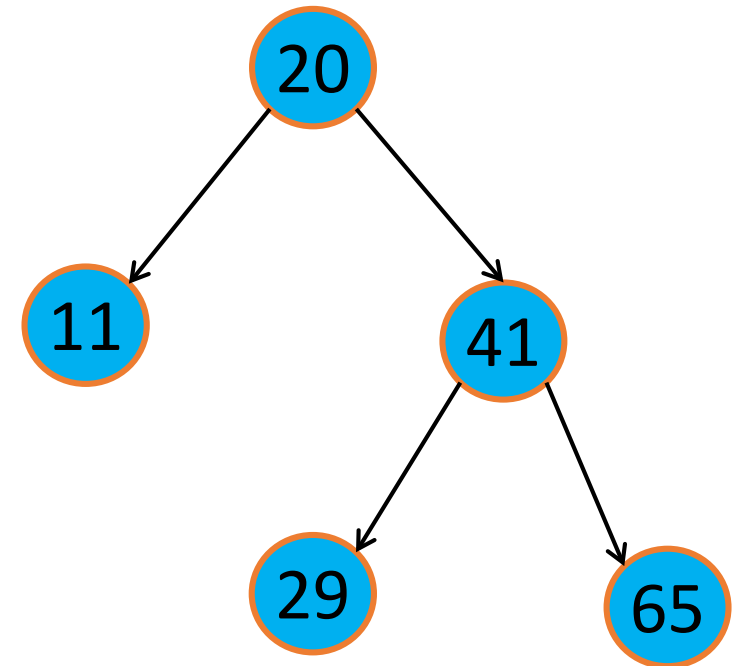
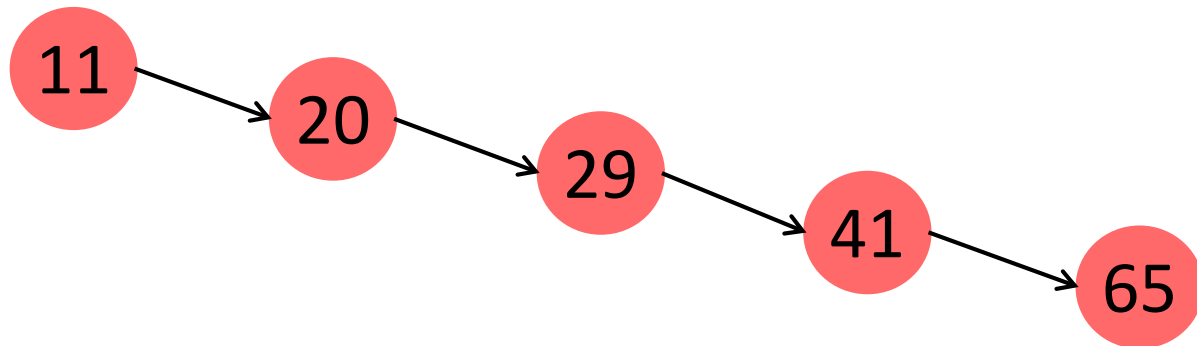
Tree Shapes

- Trees come in many shapes
 - same keys \neq same shape
 - performance depends on shape
- What determines shape?
 - Order of insertions



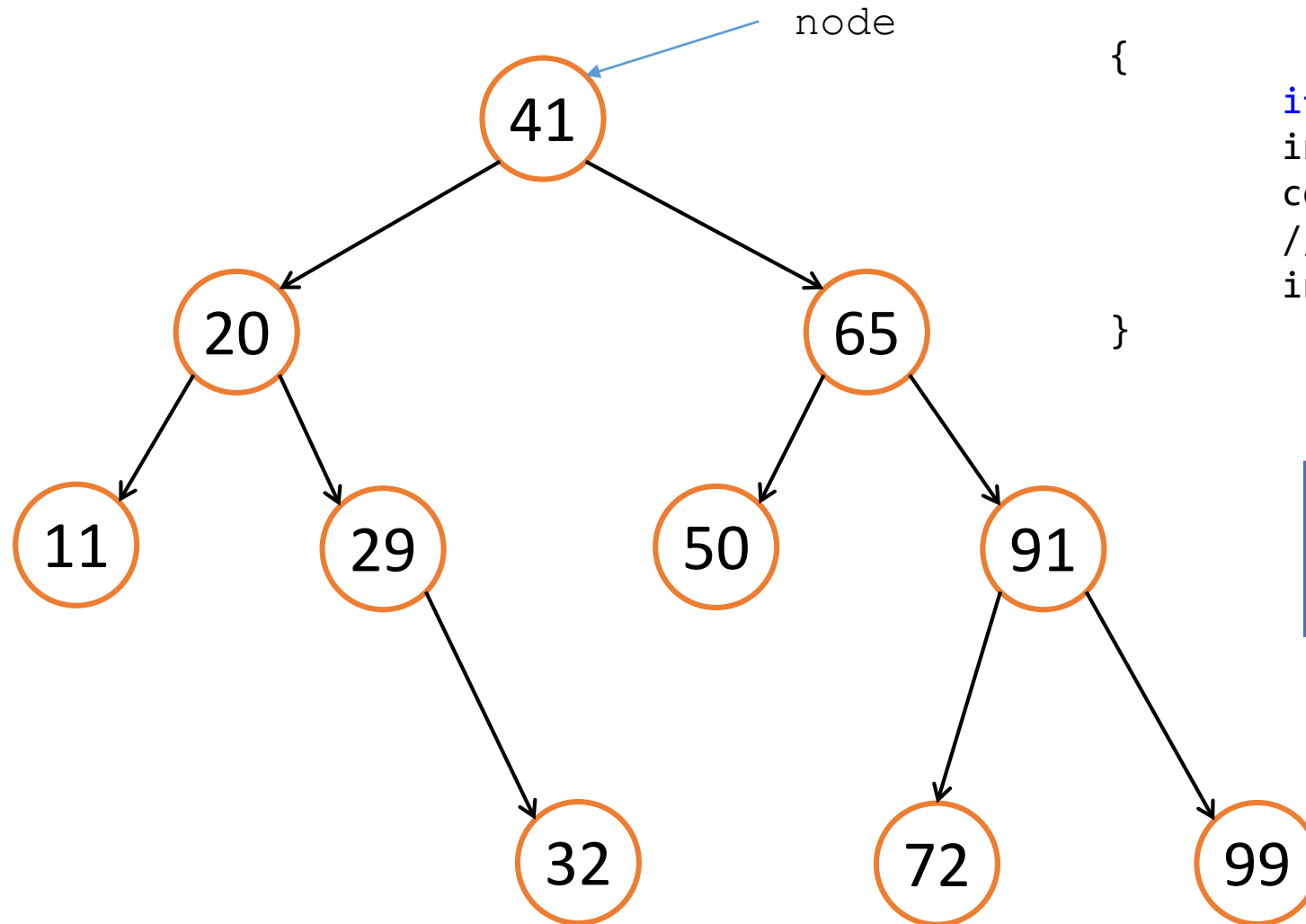
Time Complexity

- Depending on the height of the tree
- A tree with n nodes
 - Height = $O(n)$ or $O(\log n)$
- Come back next week!



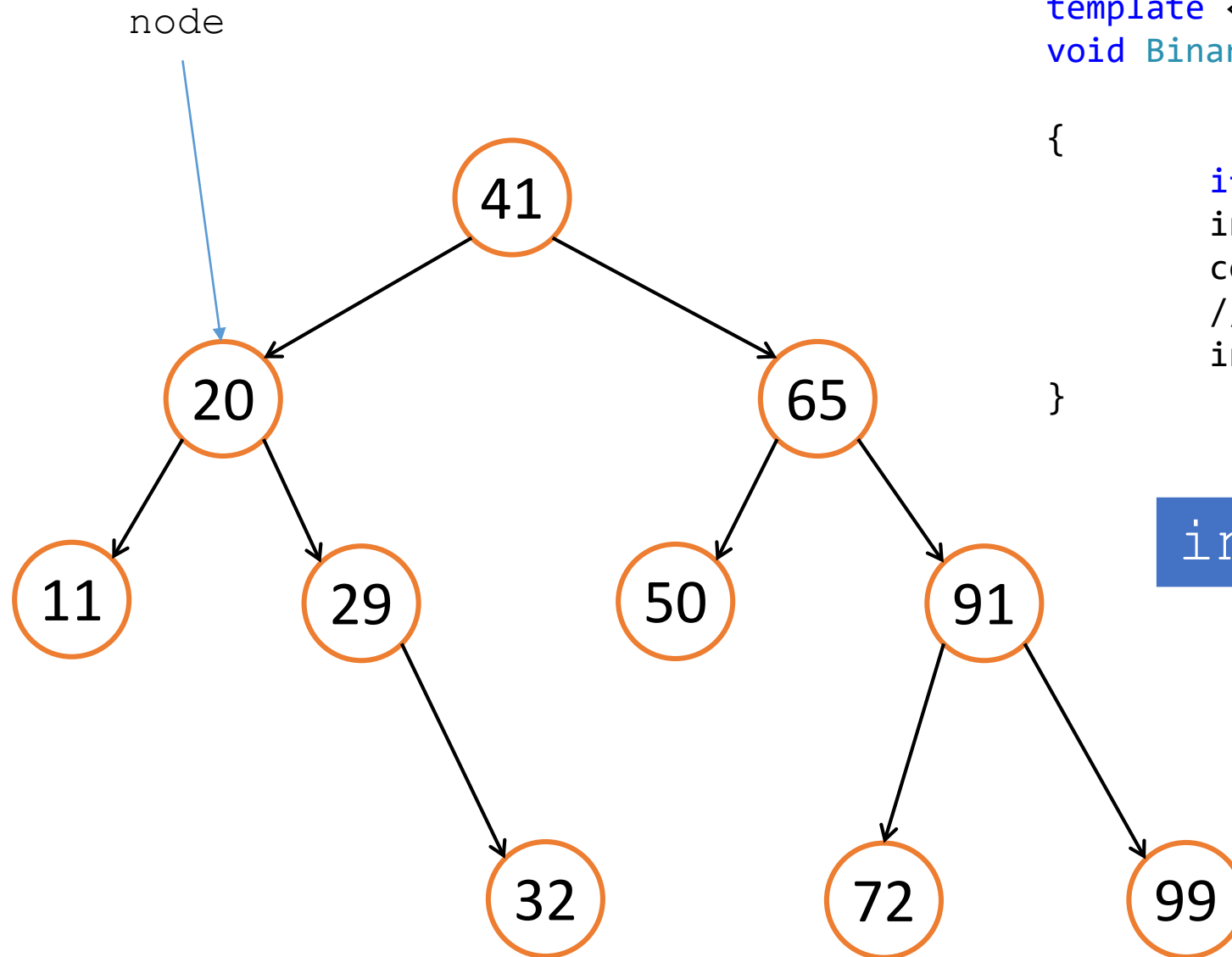
In-order-traversal(v)

```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```



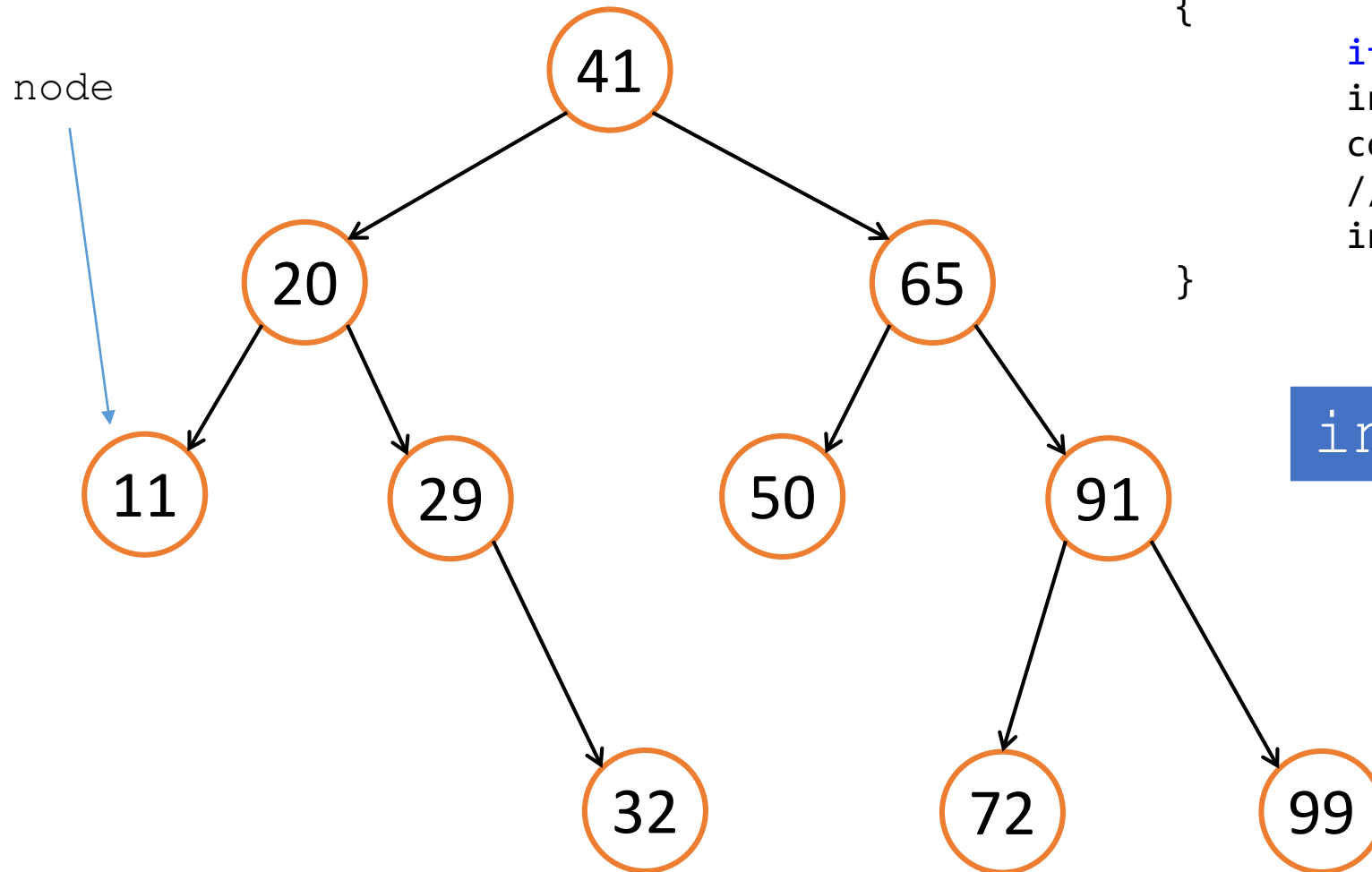
```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

Start by calling
`inOrder(_root)`



```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

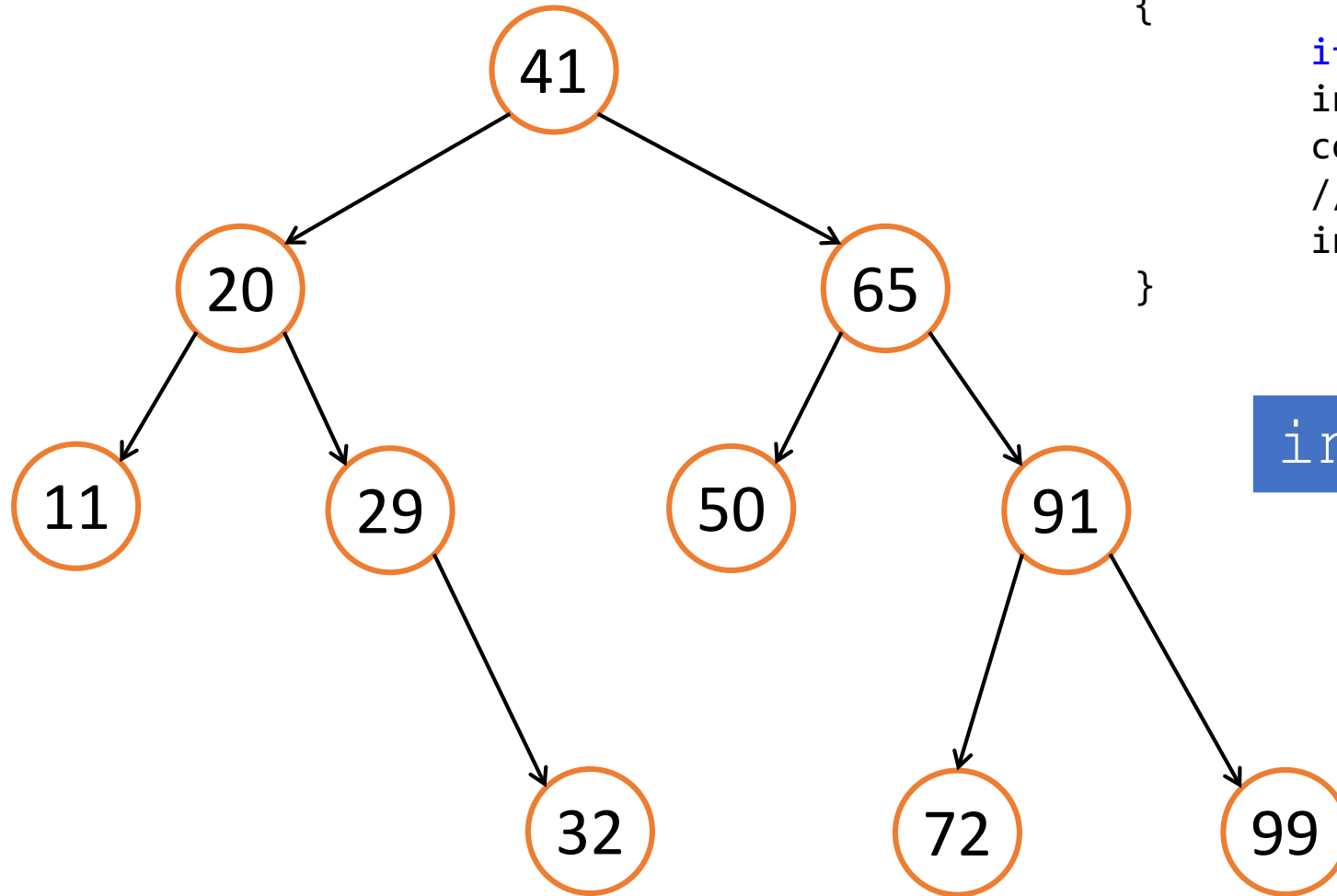
`inOrder(node->left)`



```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

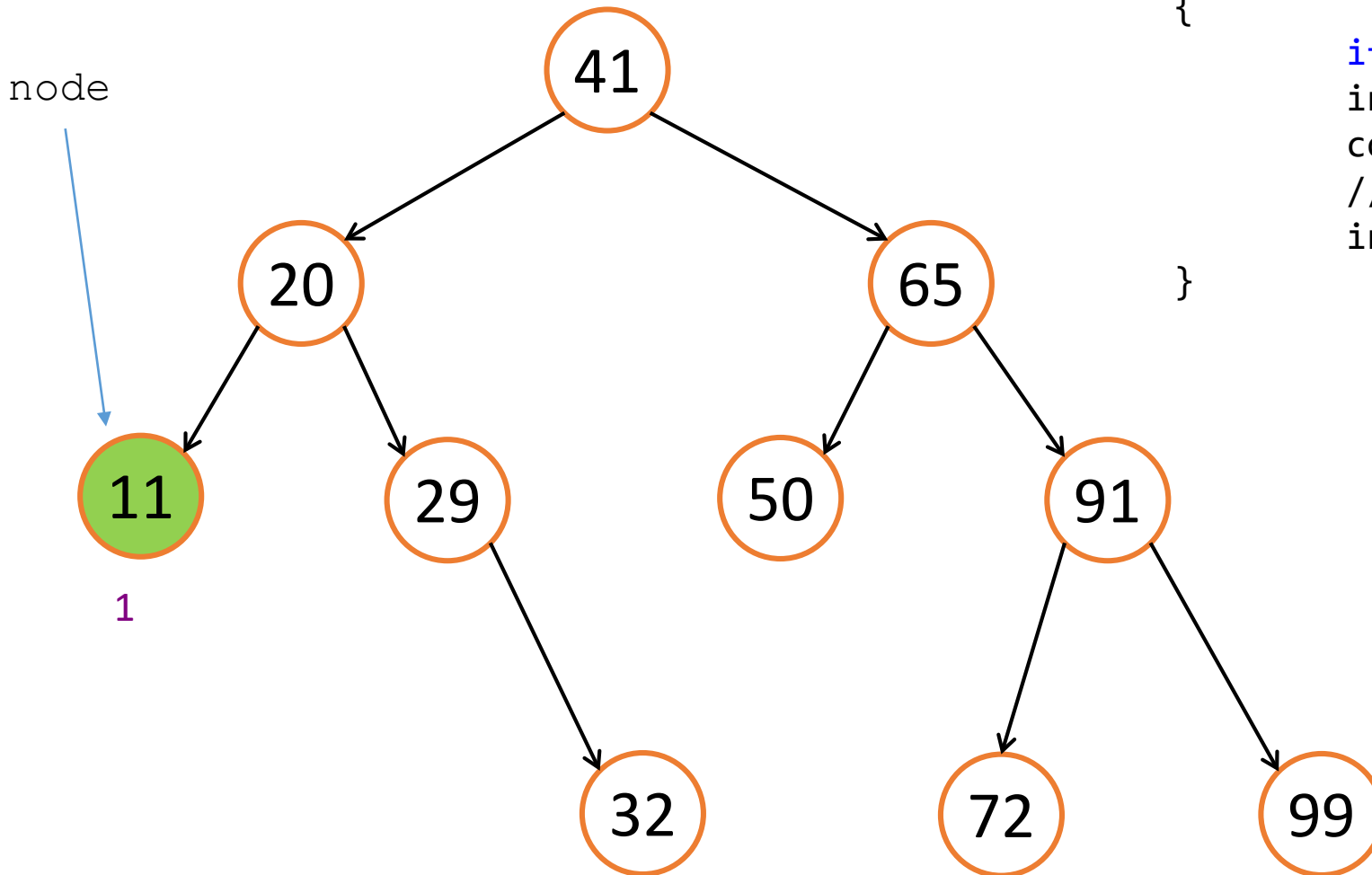
`inOrder(node->left)`

node



```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

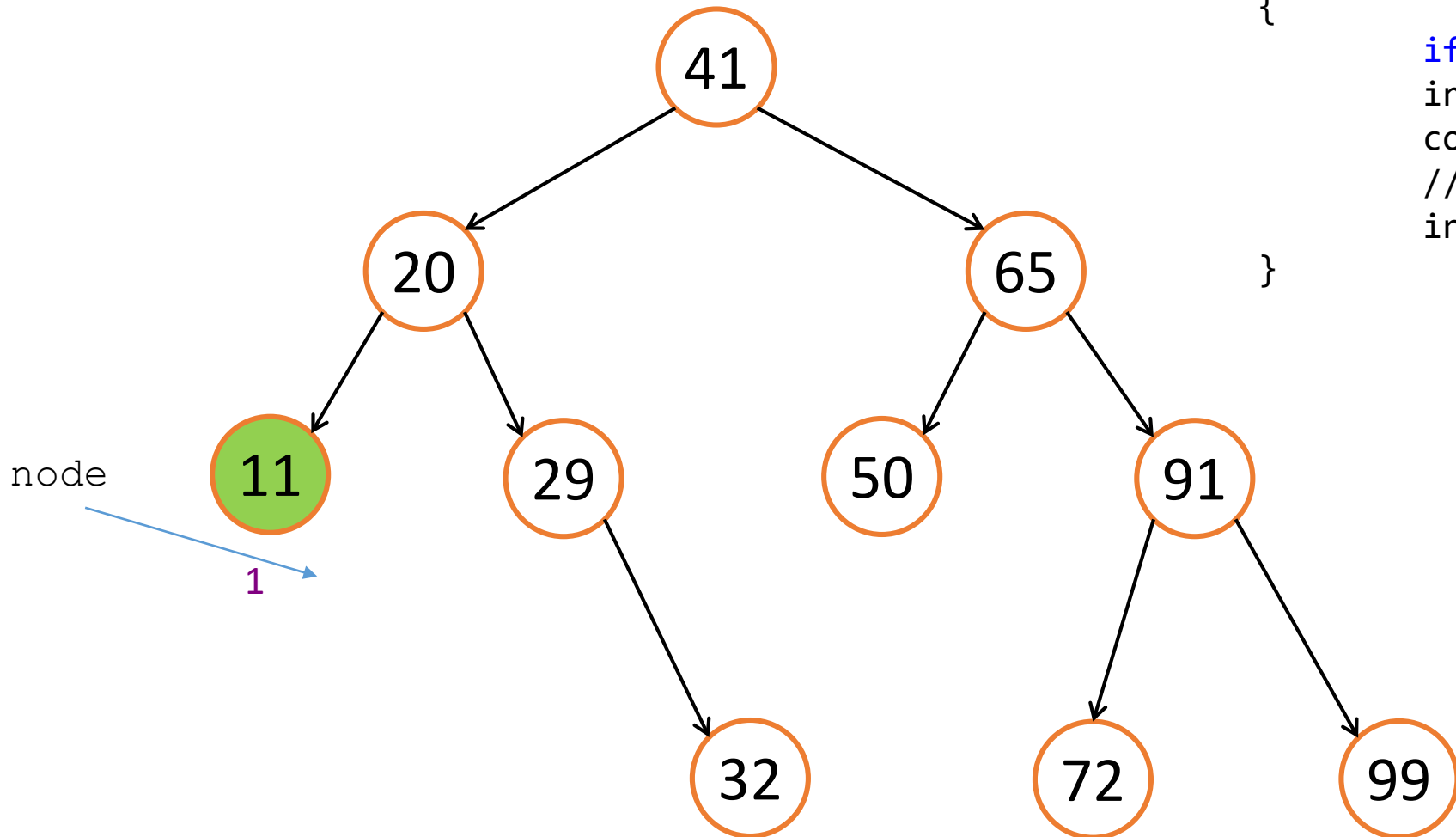
`inOrder(node->left)`



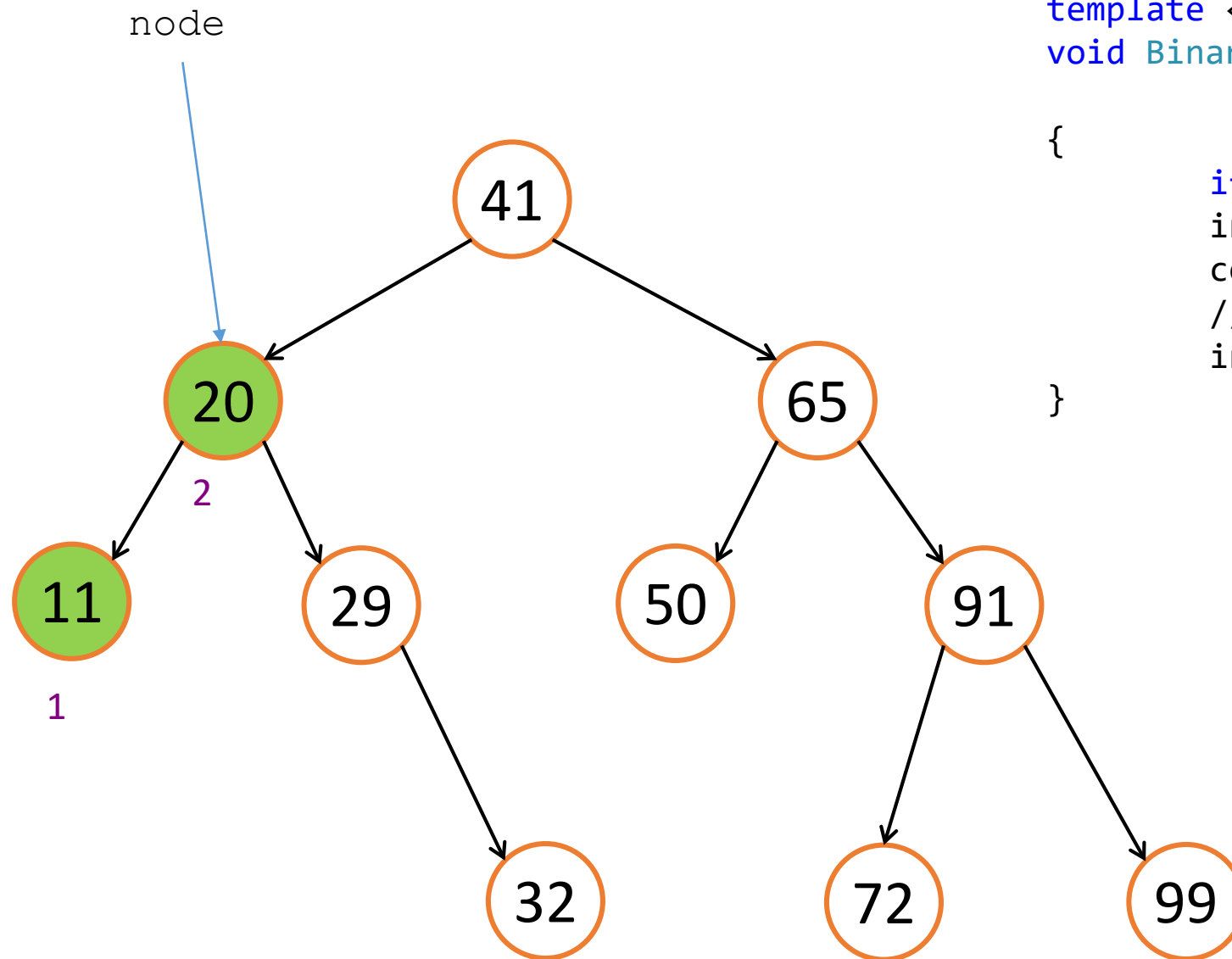
```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

```
cout << node->_item << " ";
```

```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```



`inOrder(node->_right)`



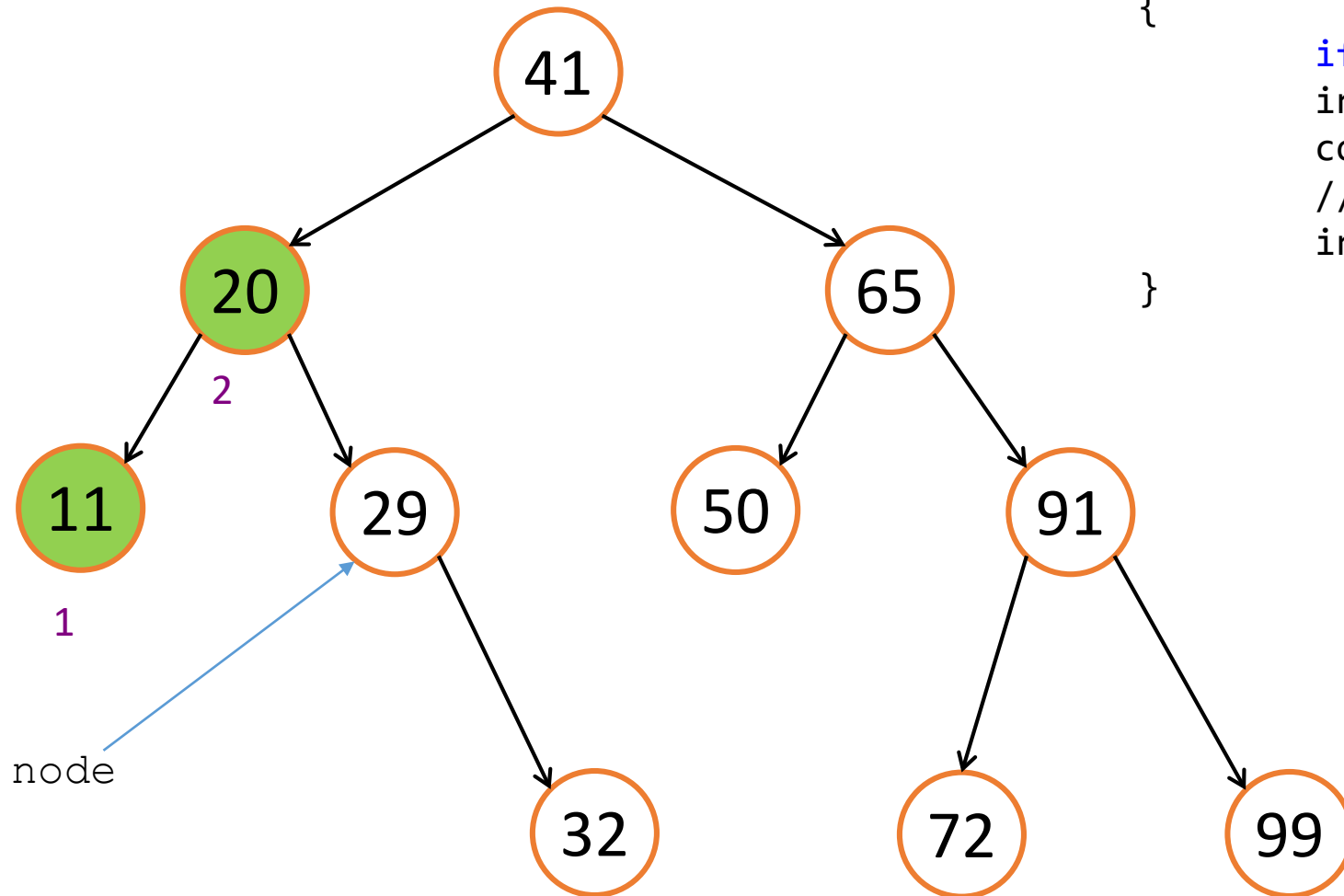
```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

```
cout << node->_item << " ";
```

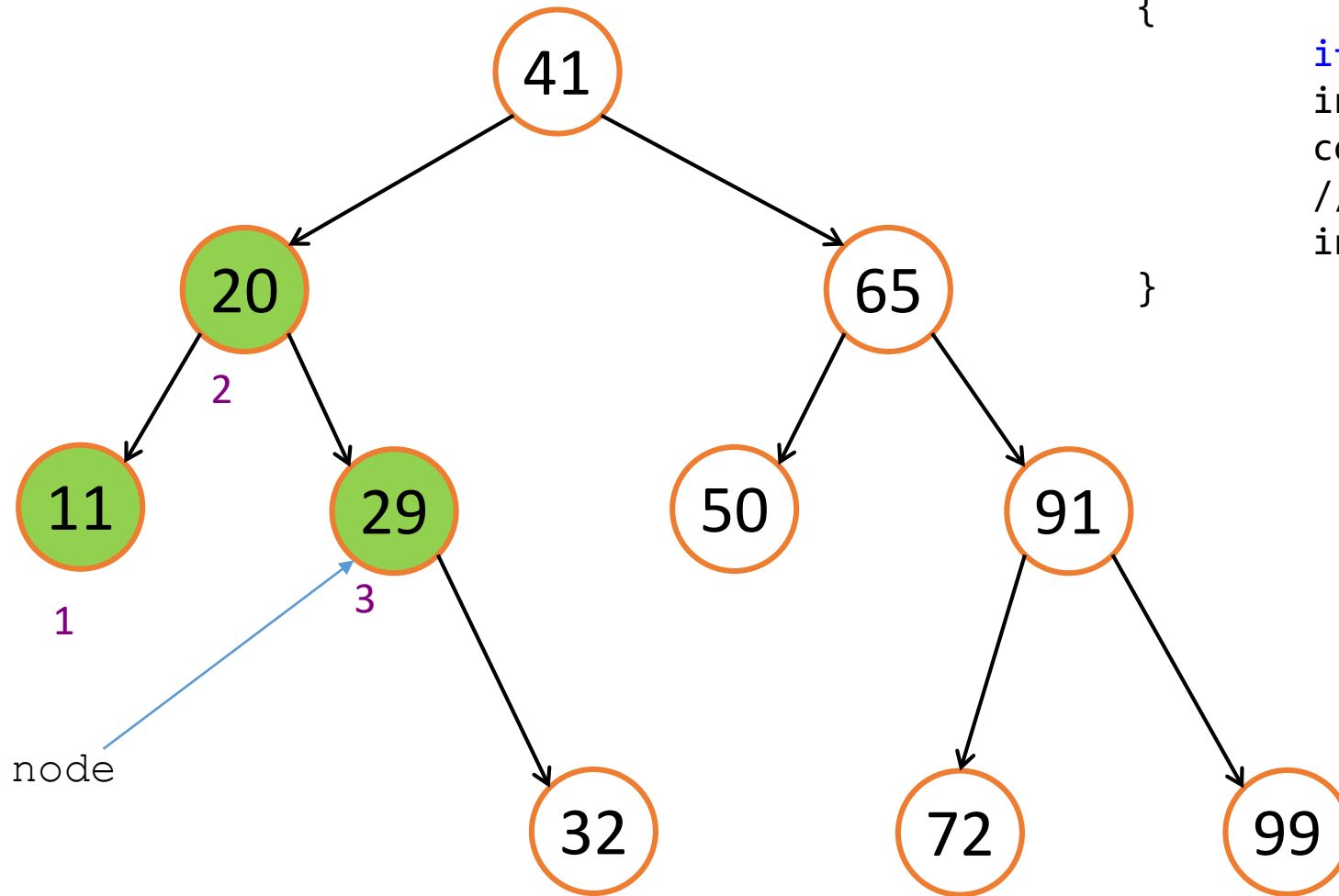
```

template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}

```



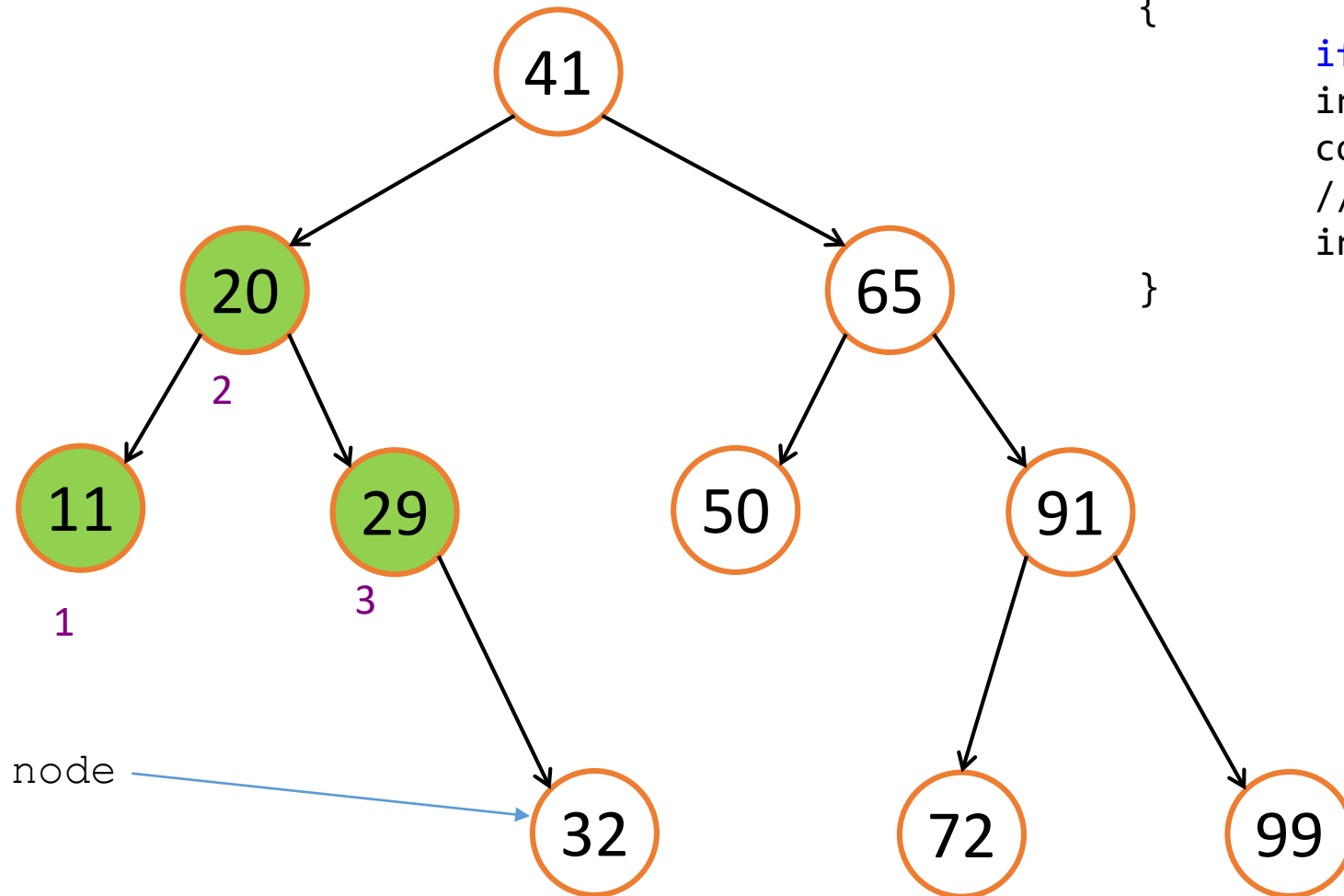
inOrder(node->_right)

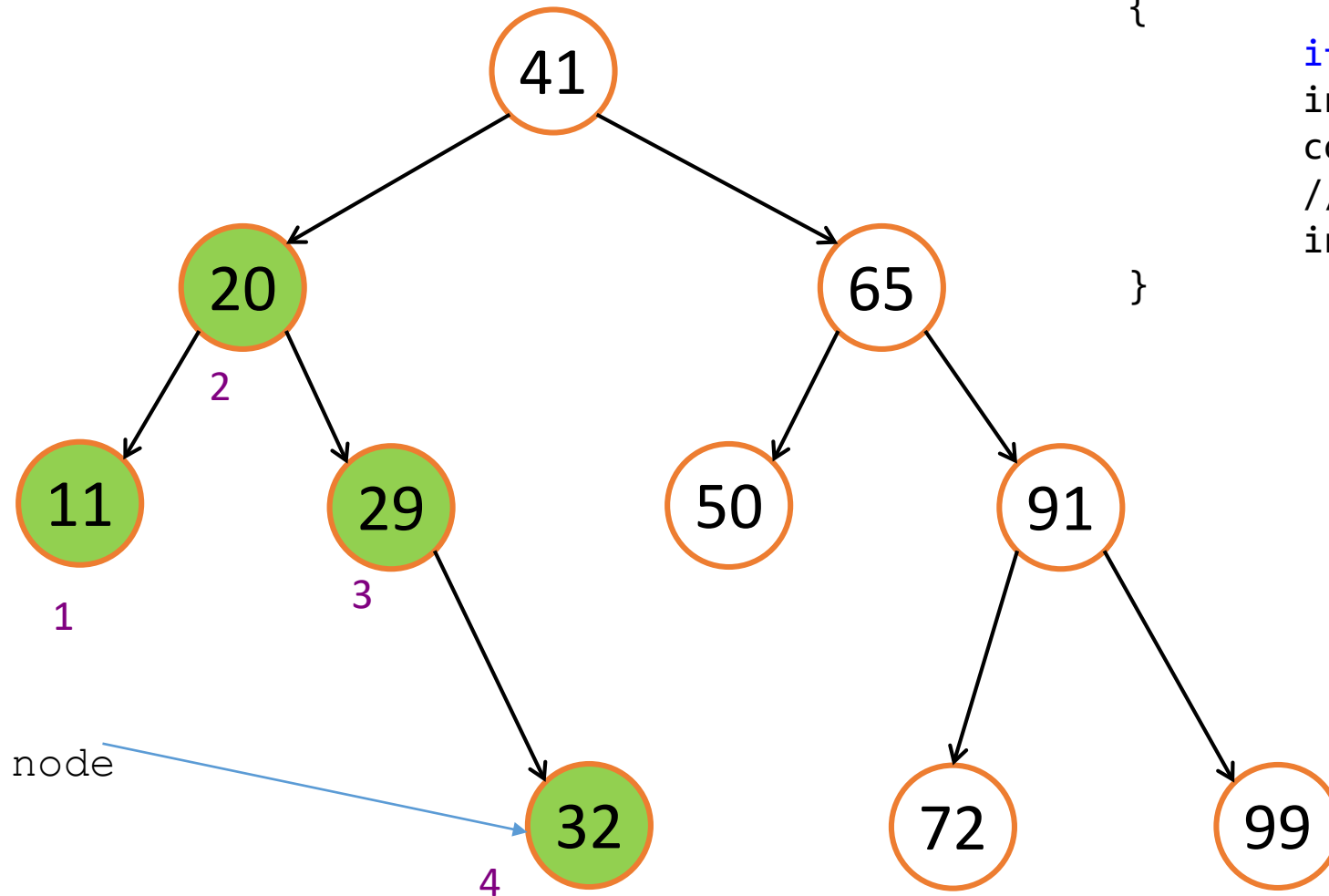


```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

```
inOrder(node->_left) (null)
cout << node->_item << " ";
```

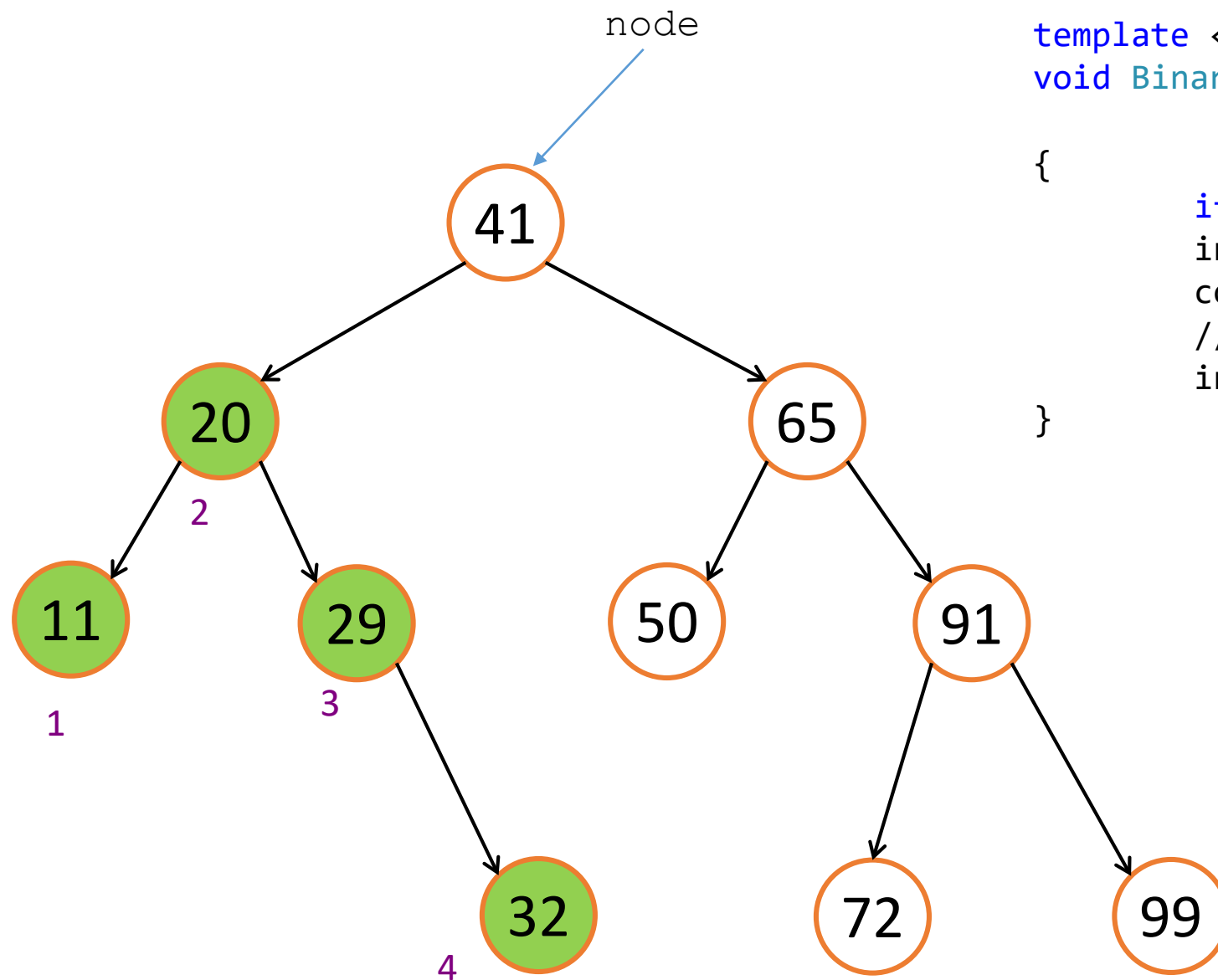
```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```





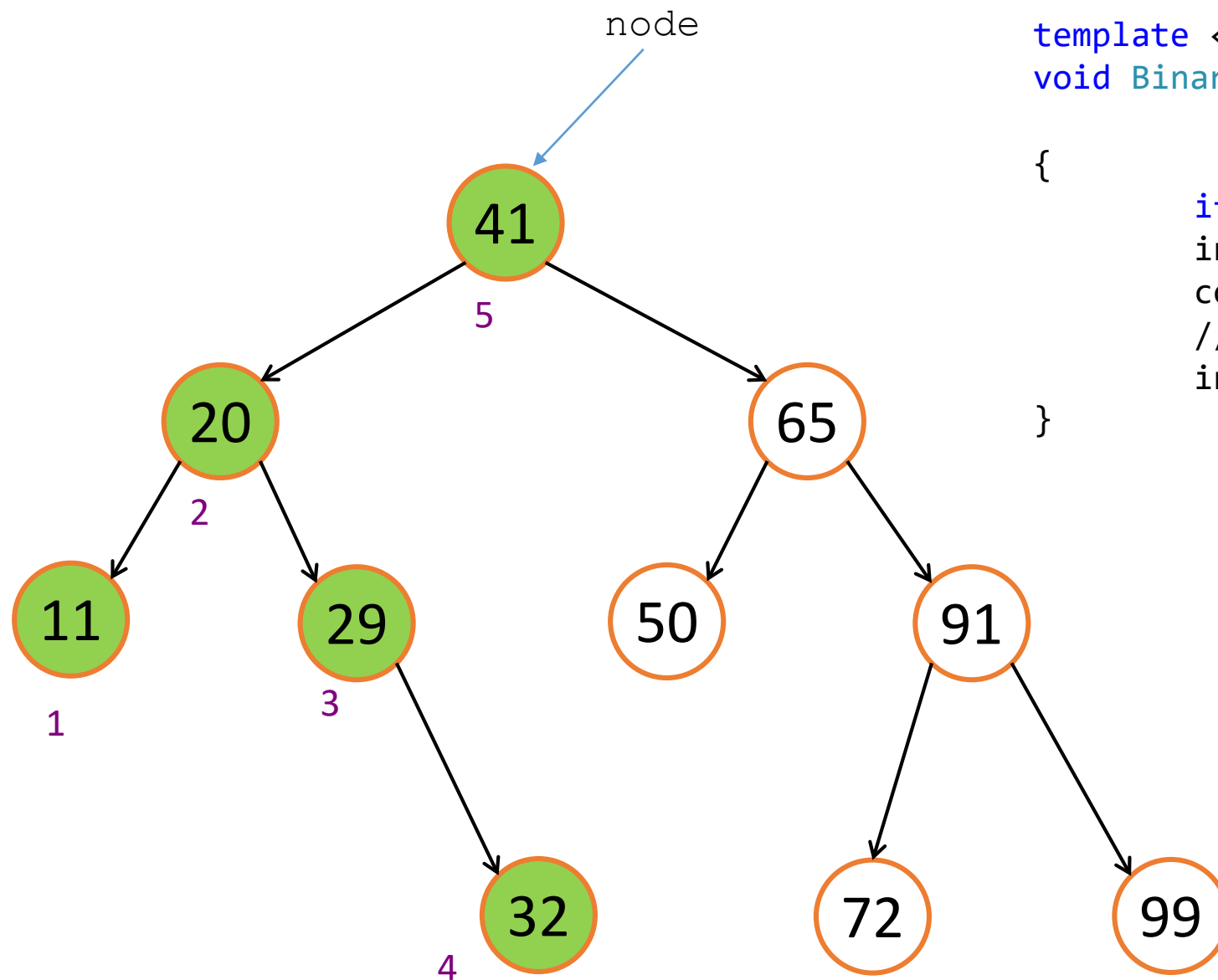
```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

```
inOrder(node->_left) (null)
cout << node->_item << " ";
inOrder(node->_right) (null)
```



```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

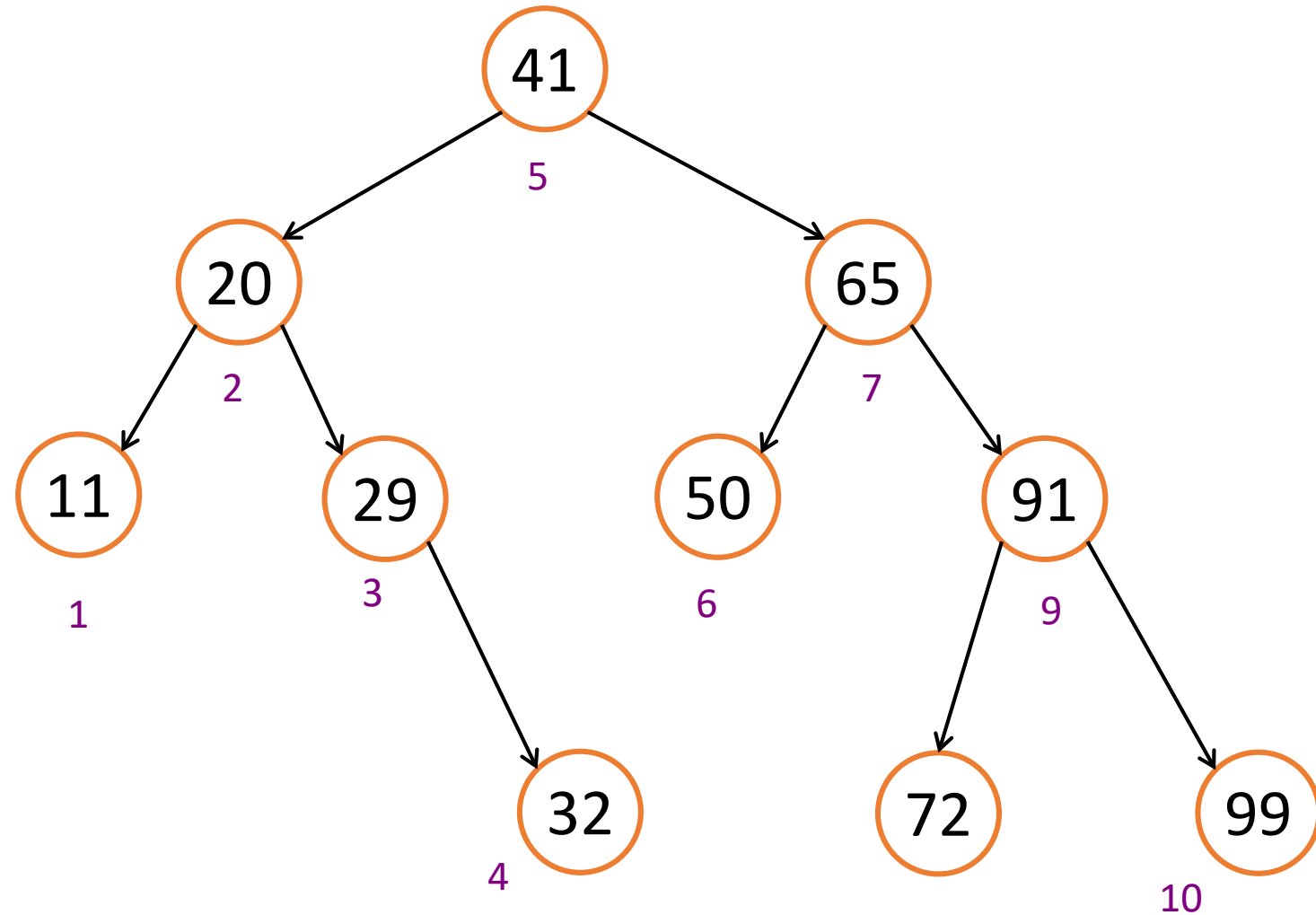
return
return
return



```
template <class T>
void BinarySearchTree<T>::inOrder
    (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    // or do something about it
    inOrder(node->_right);
}
```

```
cout << node->_item << " ";
```

In-order



11 20 29 32 41 50 65 72 91 99

- Notice something?

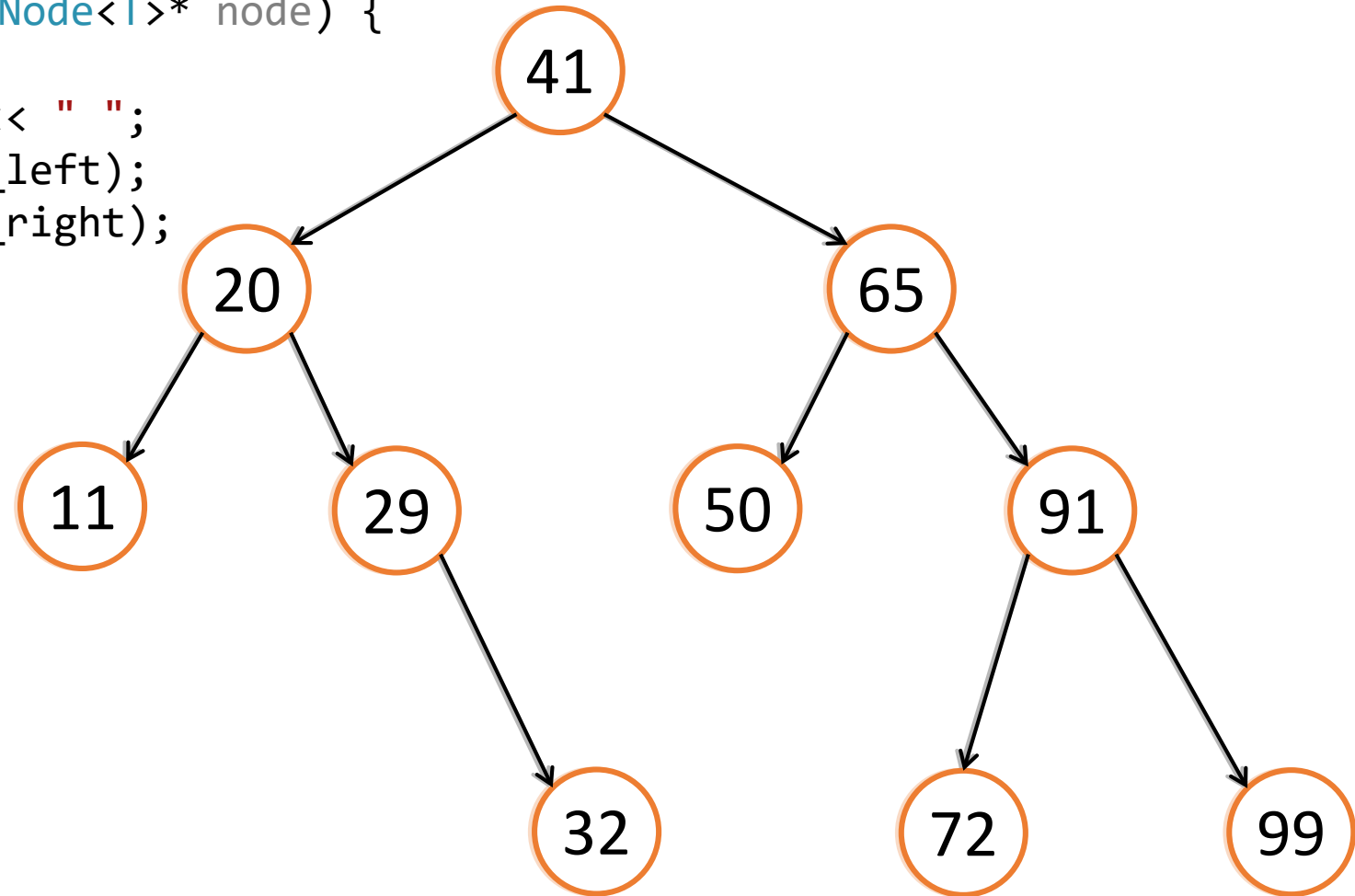
In-order-traversal(v): $O(n)$

```
template <class T>
void BinarySearchTree<T>::inOrder
                                   (TreeNode<T>* node)
{
    if (!node) return;
    inOrder(node->_left);
    cout << node->_item << " ";
    inOrder(node->_right);
}
```

Pre-order-traversal(v): $O(n)$

```
template <class T>
void BinarySearchTree<T>::
    preOrderPrint(TreeNode<T>* node) {
    if (!node) return;
    cout << node->_item << " ";
    preOrderPrint(node->_left);
    preOrderPrint(node->_right);
}
```

```
template <class T>
void BinarySearchTree<T>::
    preOrderPrint(TreeNode<T>* node) {
    if (!node) return;
    cout << node->_item << " ";
    preOrderPrint(node->_left);
    preOrderPrint(node->_right);
}
```

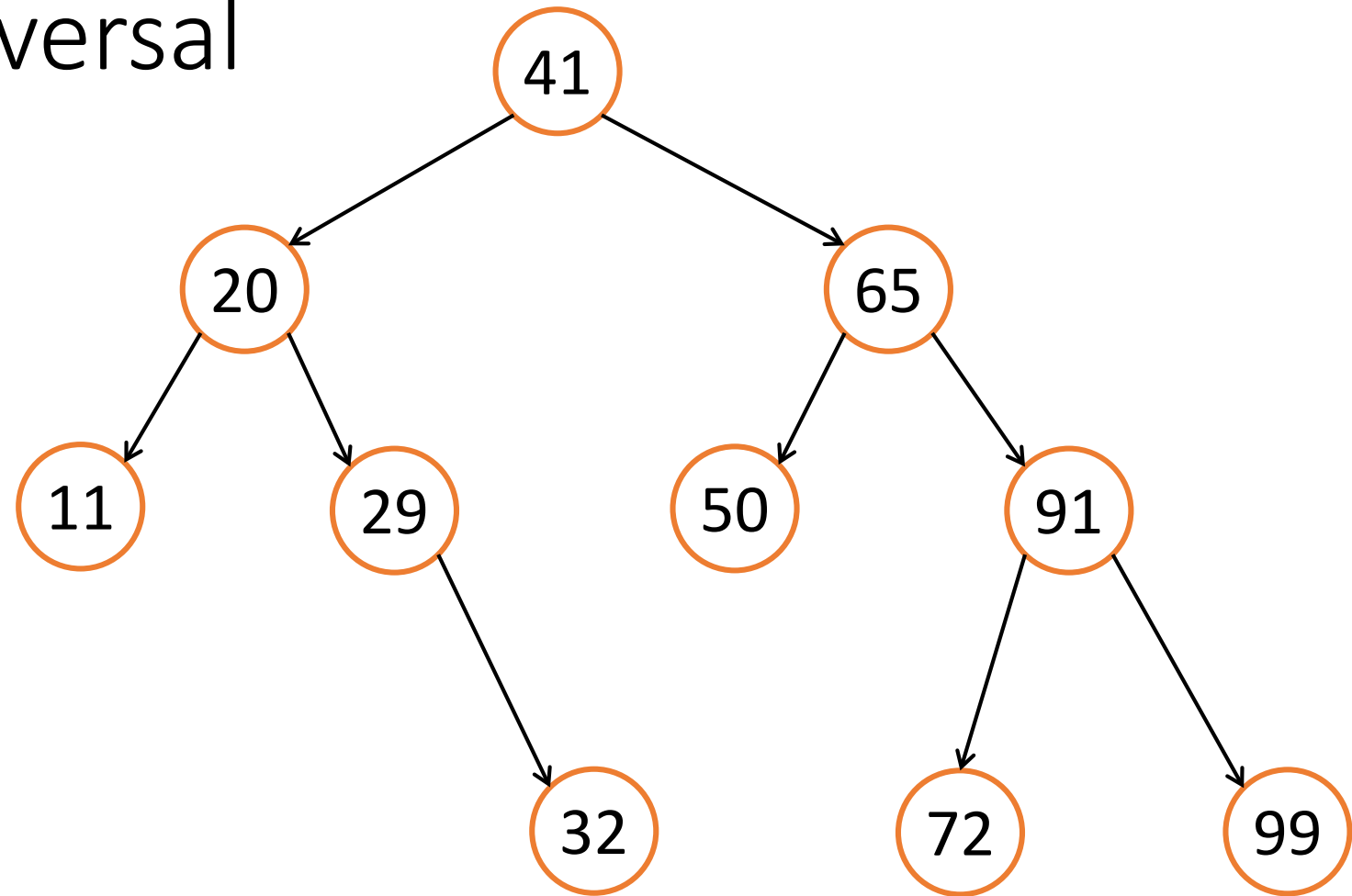


41 20 11 29 32 65 50 91 72 99

Post-order-traversal(v): $O(n)$

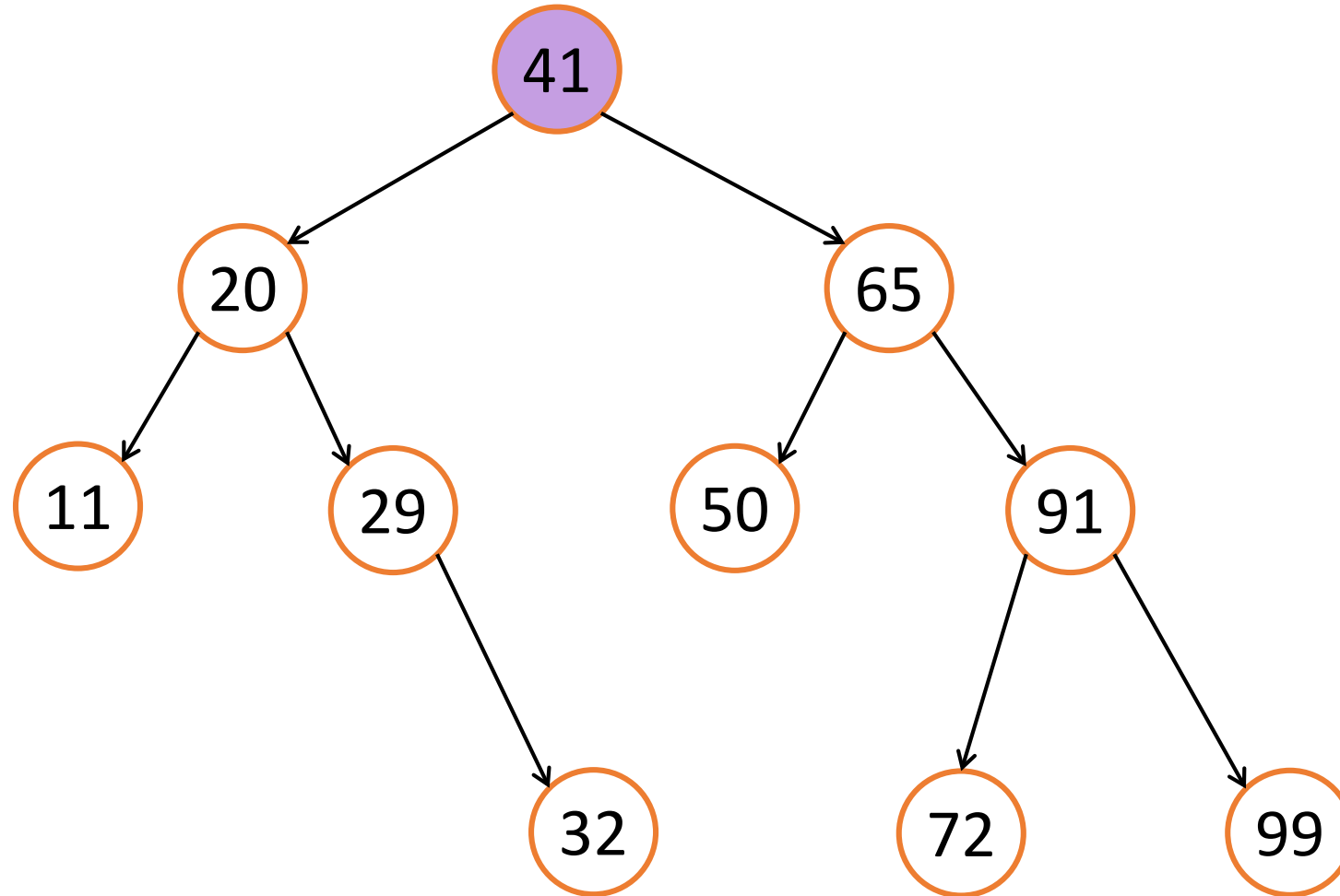
```
template <class T>
void BinarySearchTree<T>::
    postOrderPrint(TreeNode<T>* node) {
    if (!node) return;
    postOrderPrint(node->_left);
    postOrderPrint(node->_right);
    cout << node->_item << " ";
}
```

Post-order Traversal



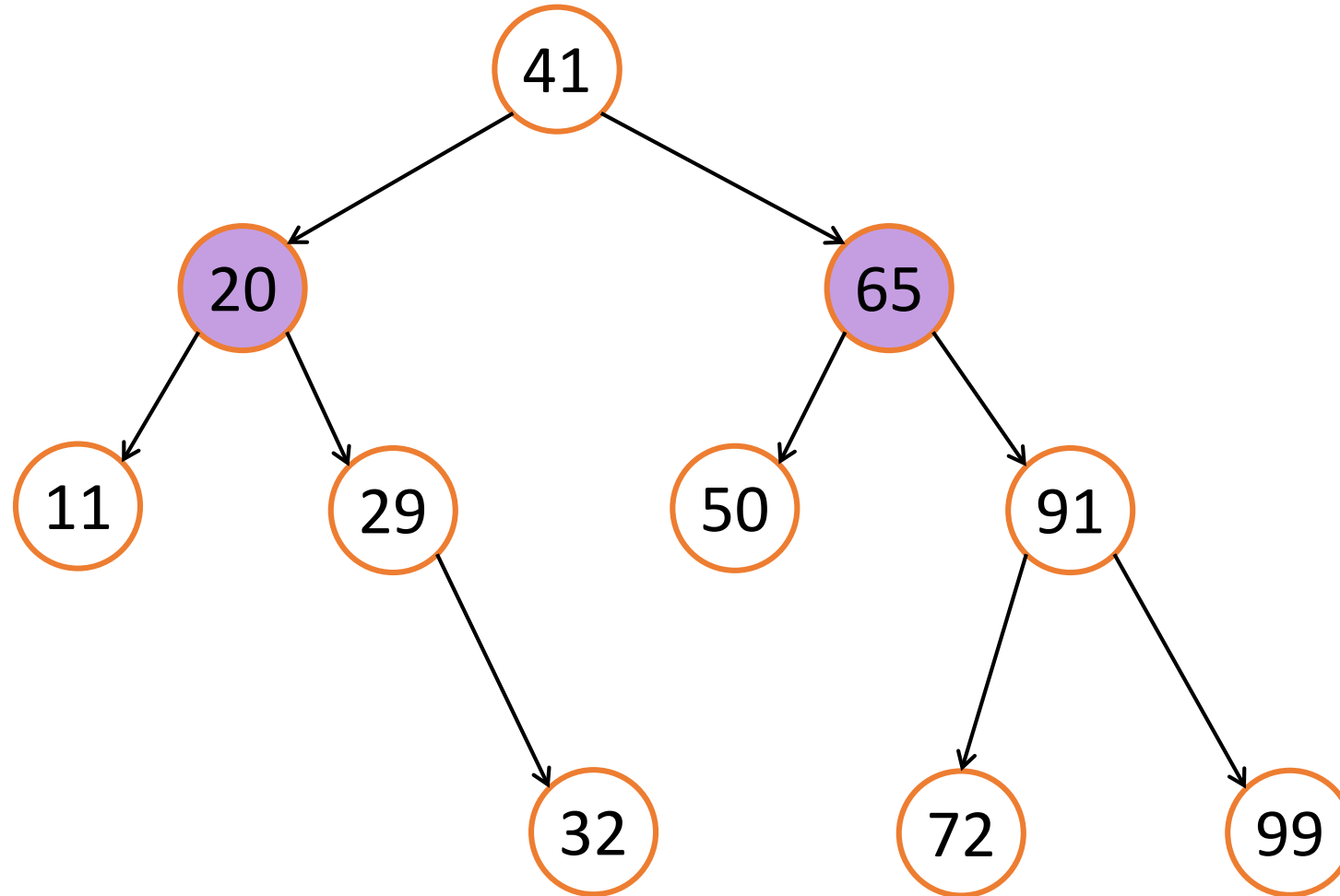
11 32 29 20 50 72 99 91 65 41

Level-order Traversal



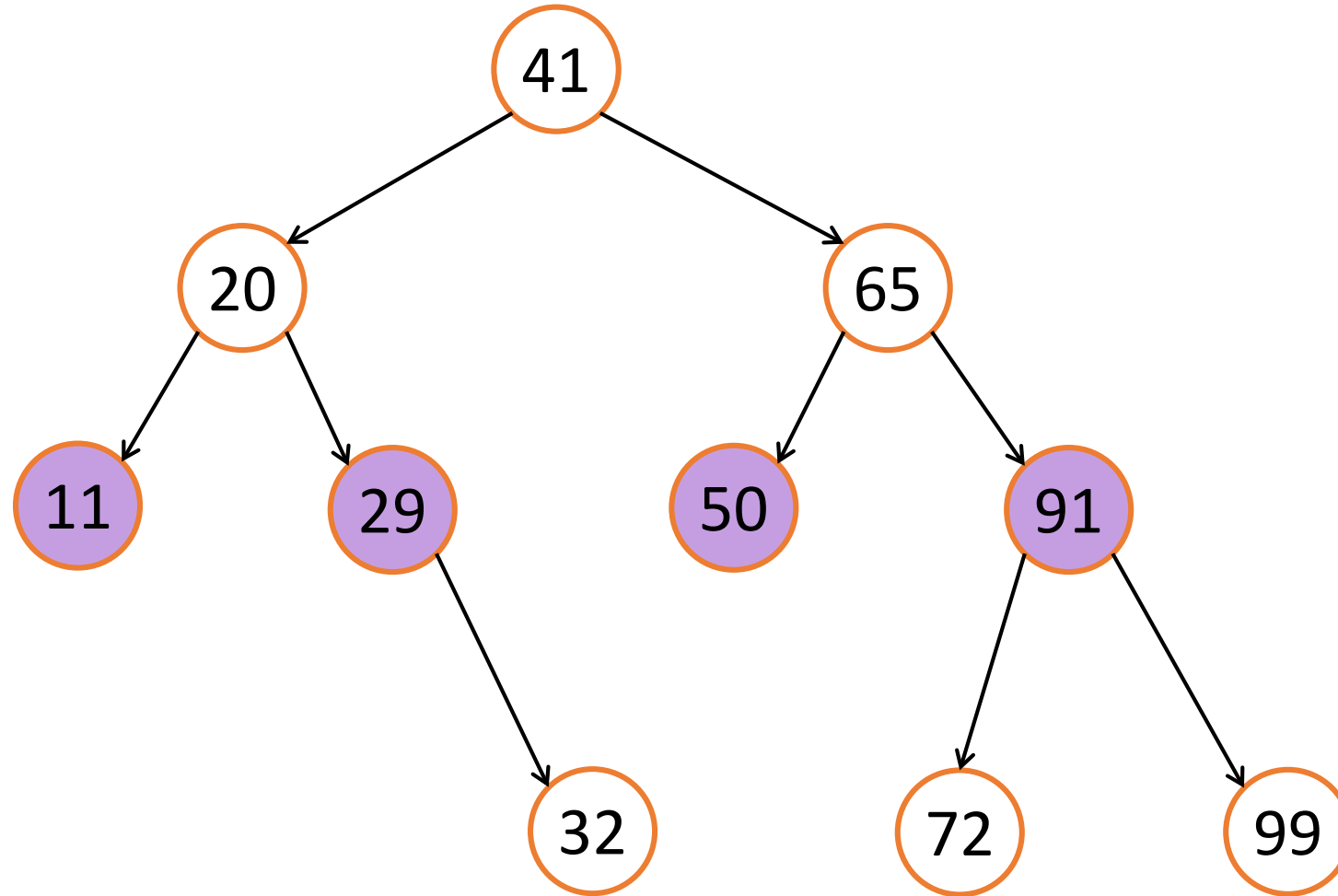
41

Level-order Traversal



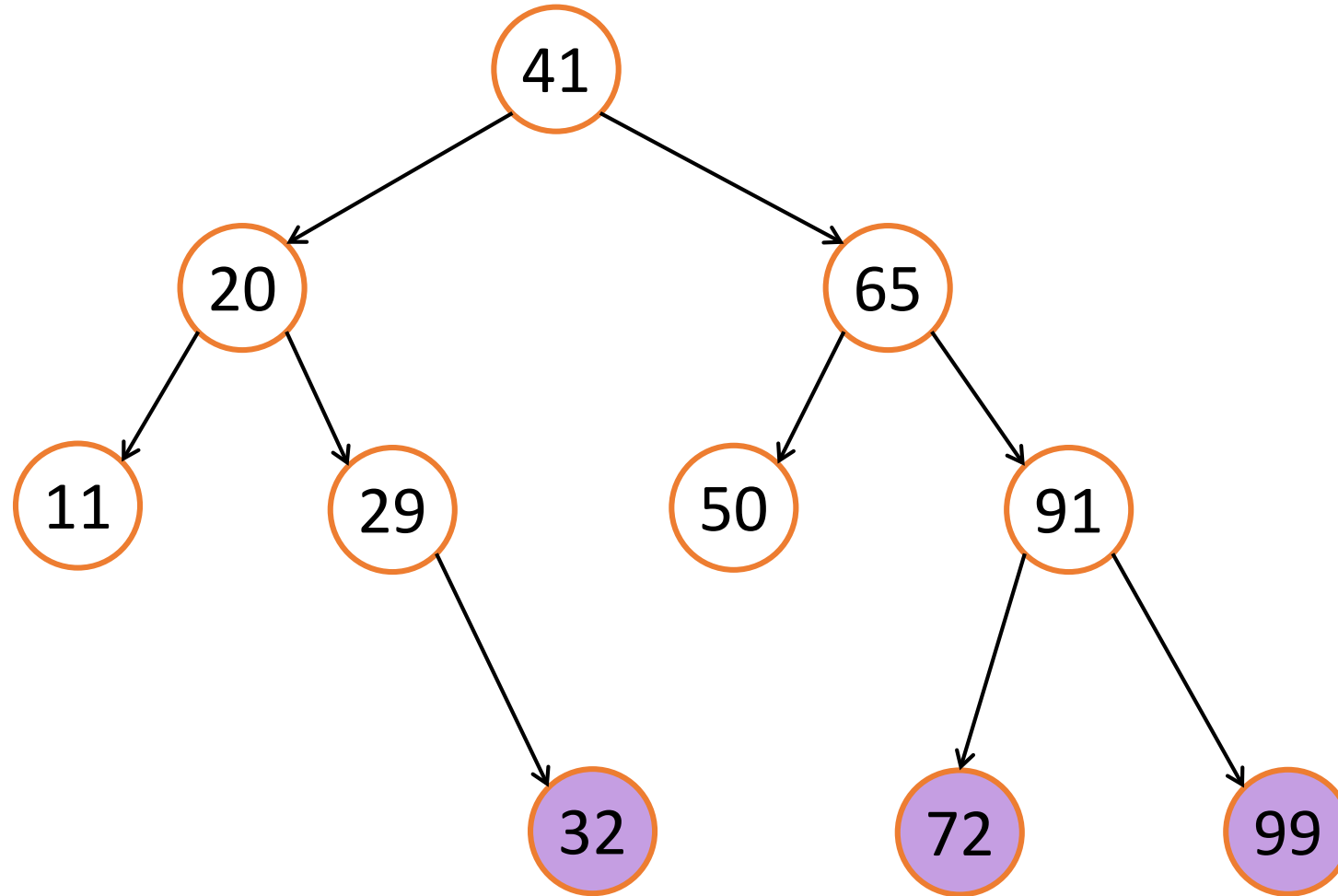
41 20 65

Level-order Traversal



41 20 65 11 29 50 91

Level-order Traversal



41 20 65 11 29 50 91 32 72 99

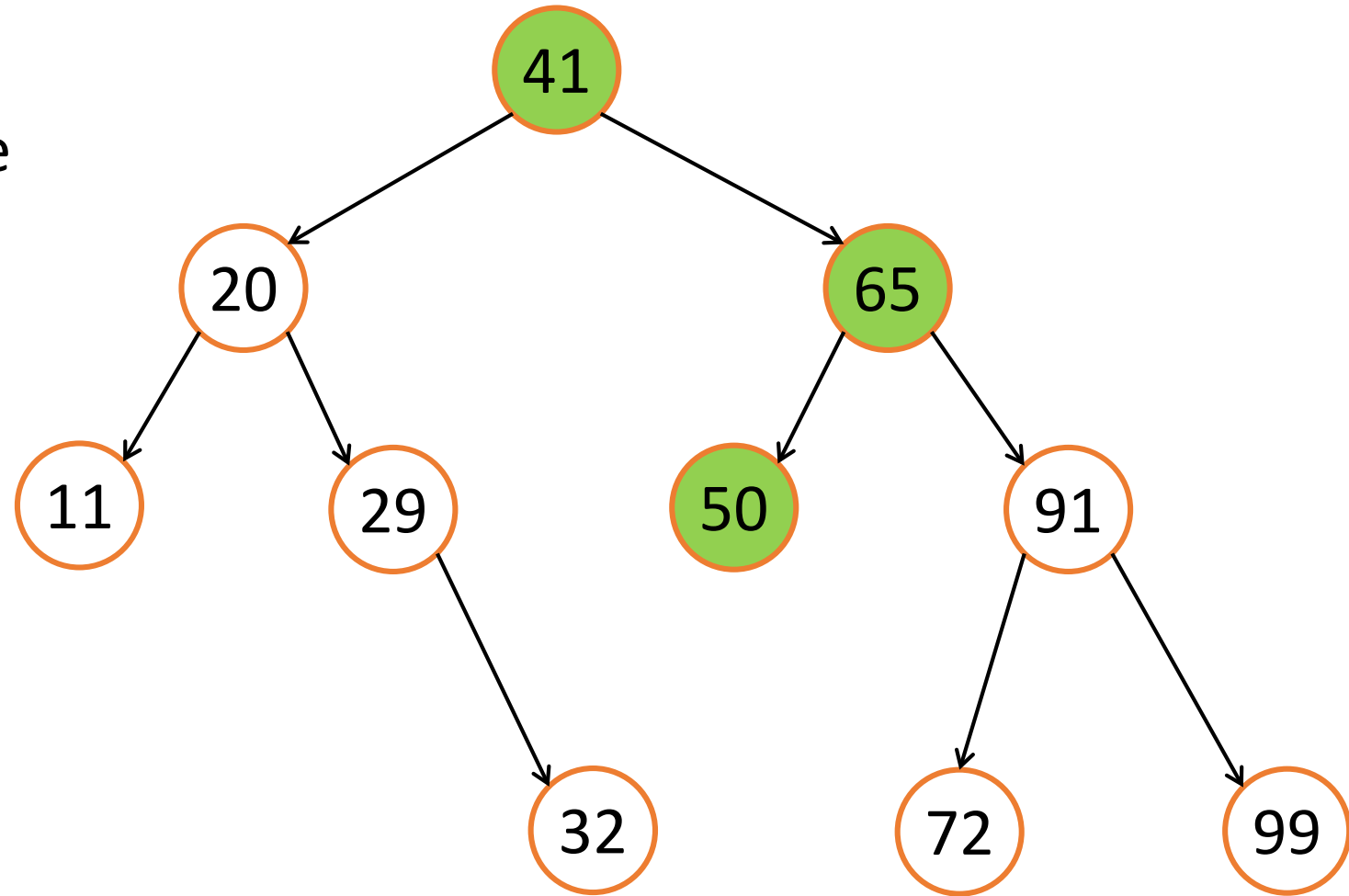
Application: Airport Scheduling

6:35	7:00	7:19	8:21	12:21	14:23	14:42			
------	------	------	------	-------	-------	-------	--	--	--

- What is the next plane after 8:30?

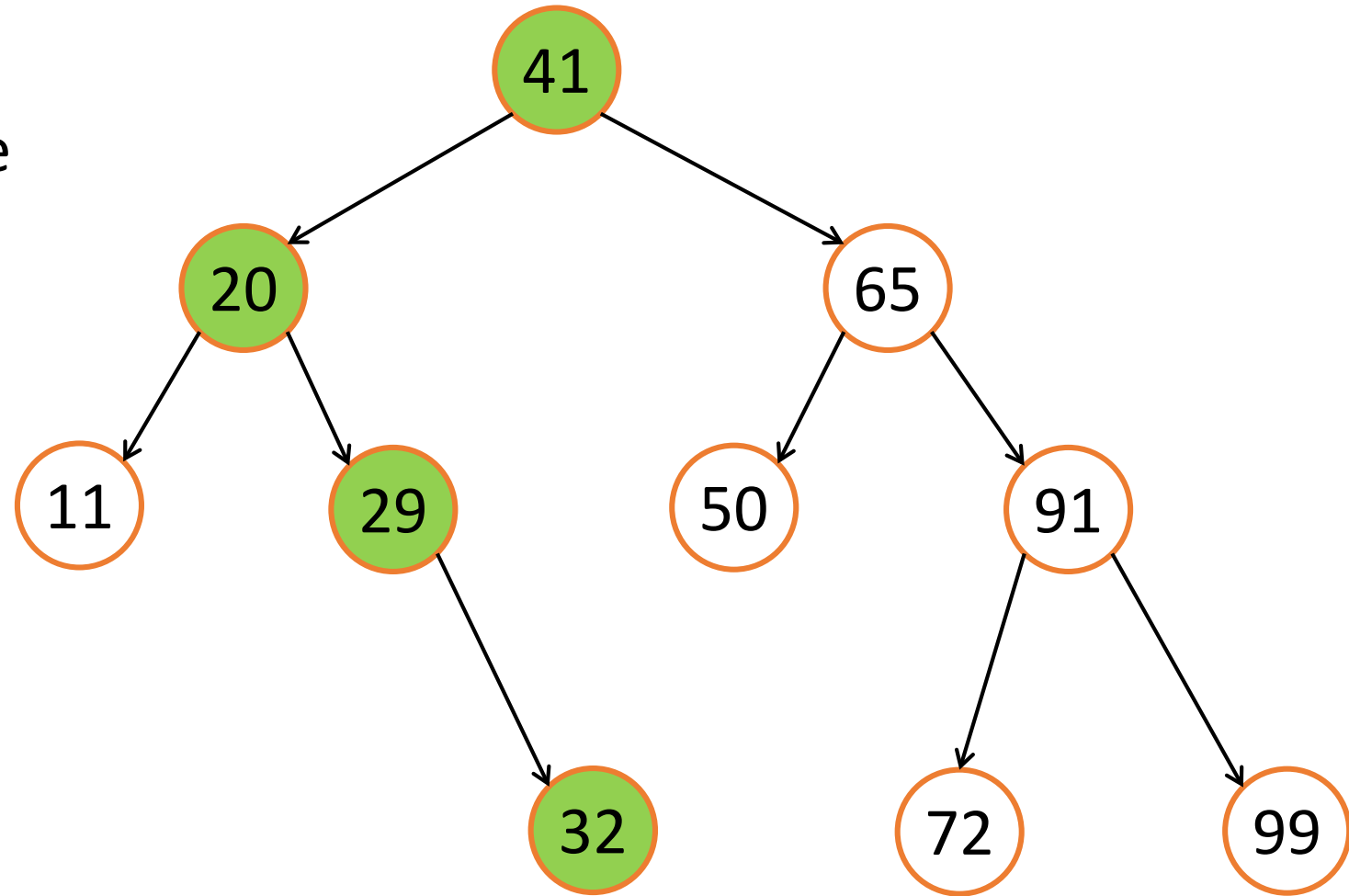
Successor Queries

- `successor(42)`
- When 42 is NOT in the tree



Successor Queries

- `successor(33)`
- When 33 is NOT in the tree

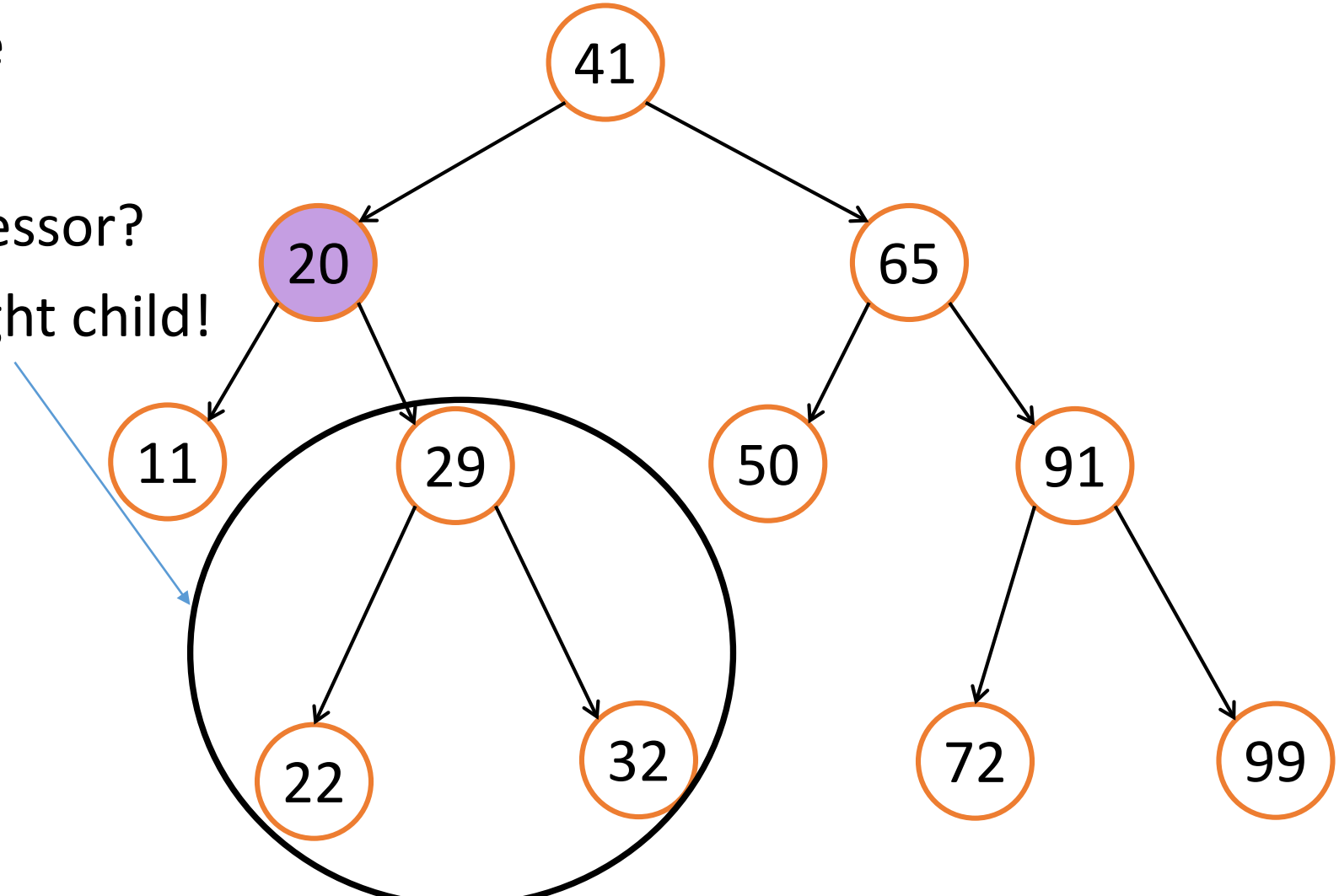


Basic strategy: Successor(key)

1. Search for key in the tree.
2. If (result > key), then return result.
3. If (result <= key), then?

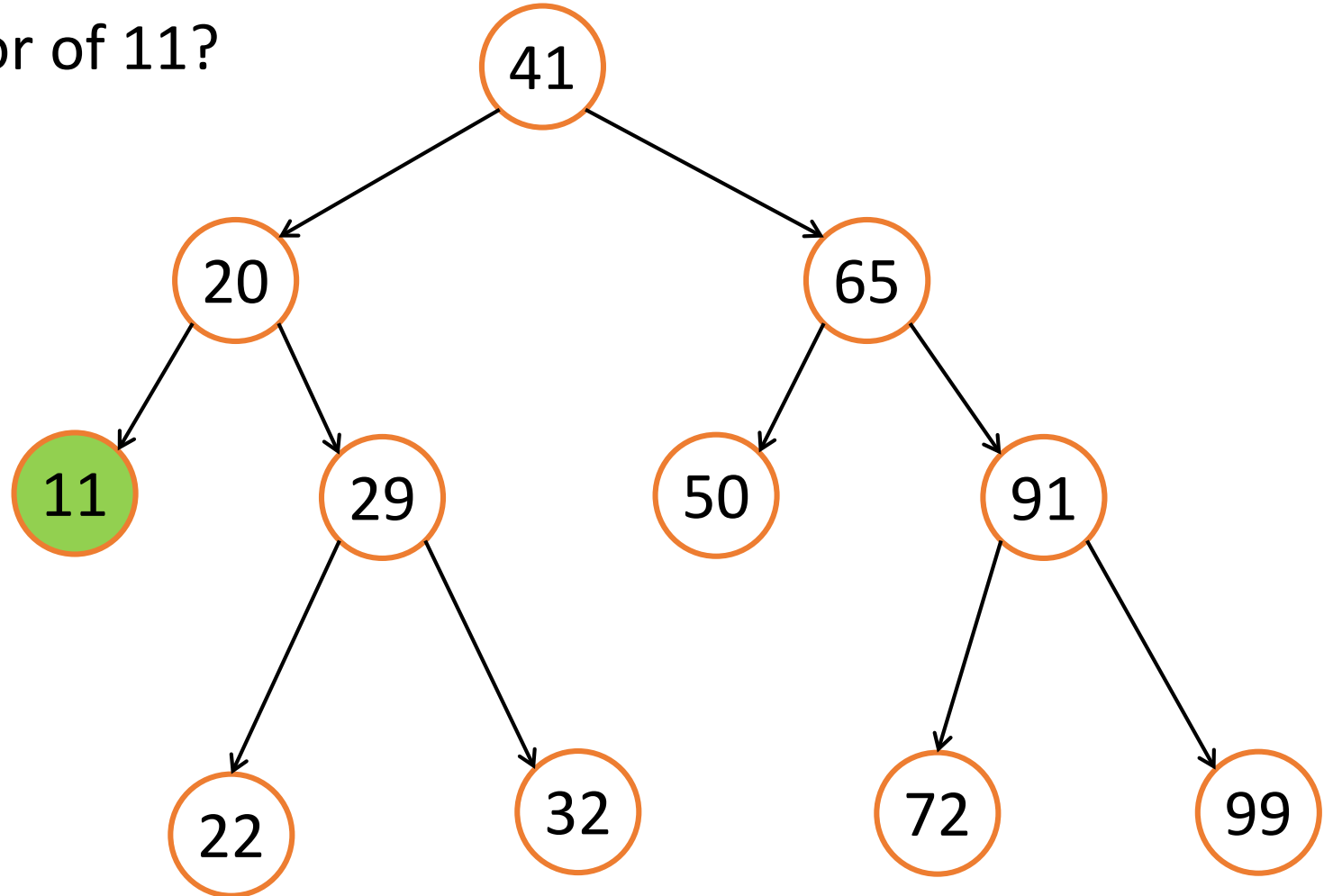
Successor Queries

- If the key is IN the tree
- s.g. `successor(20)`
- Which one is the successor?
- The minimum of its right child!
- Why?



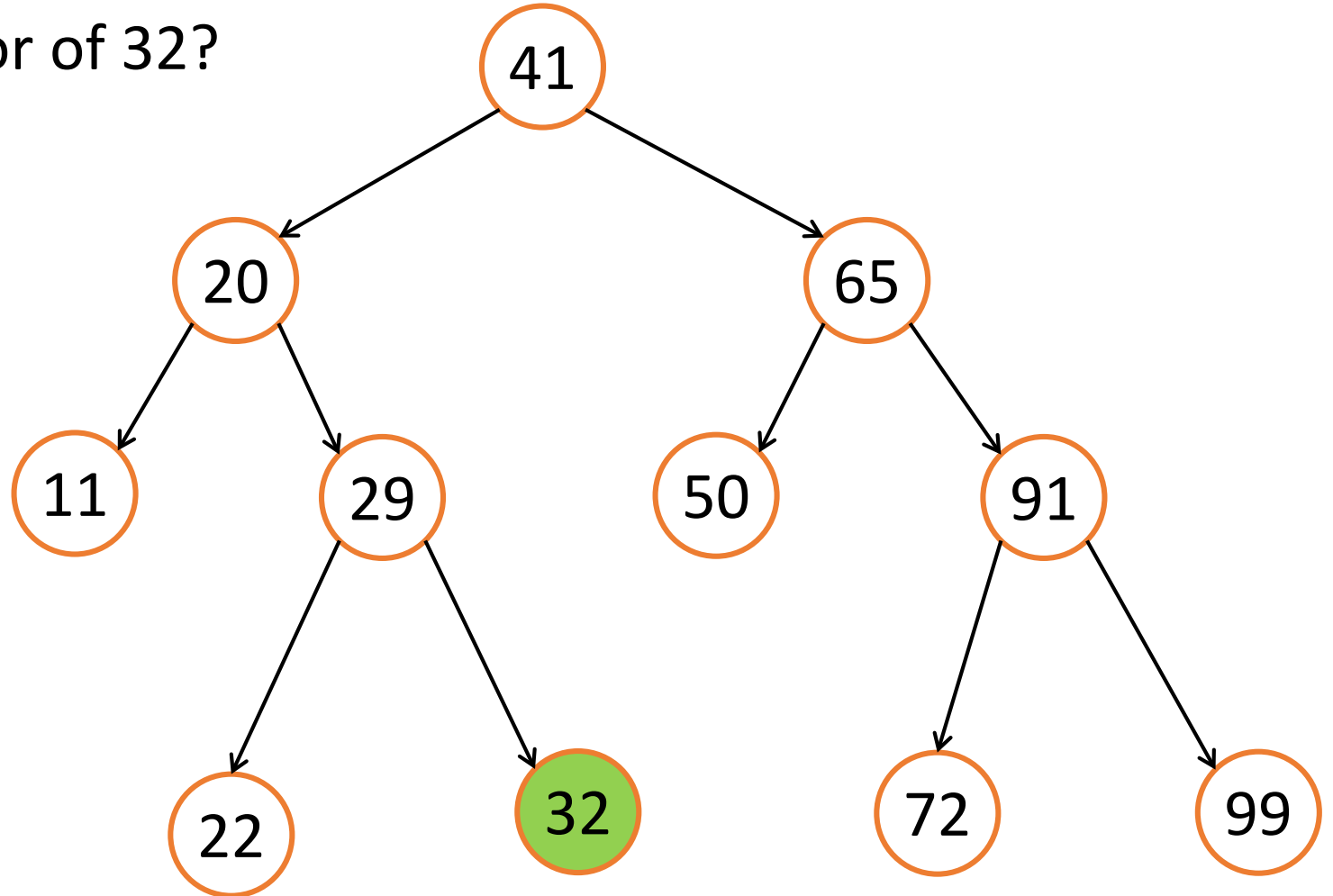
Successor Queries: No Right Child?

- Which one is the successor of 11?

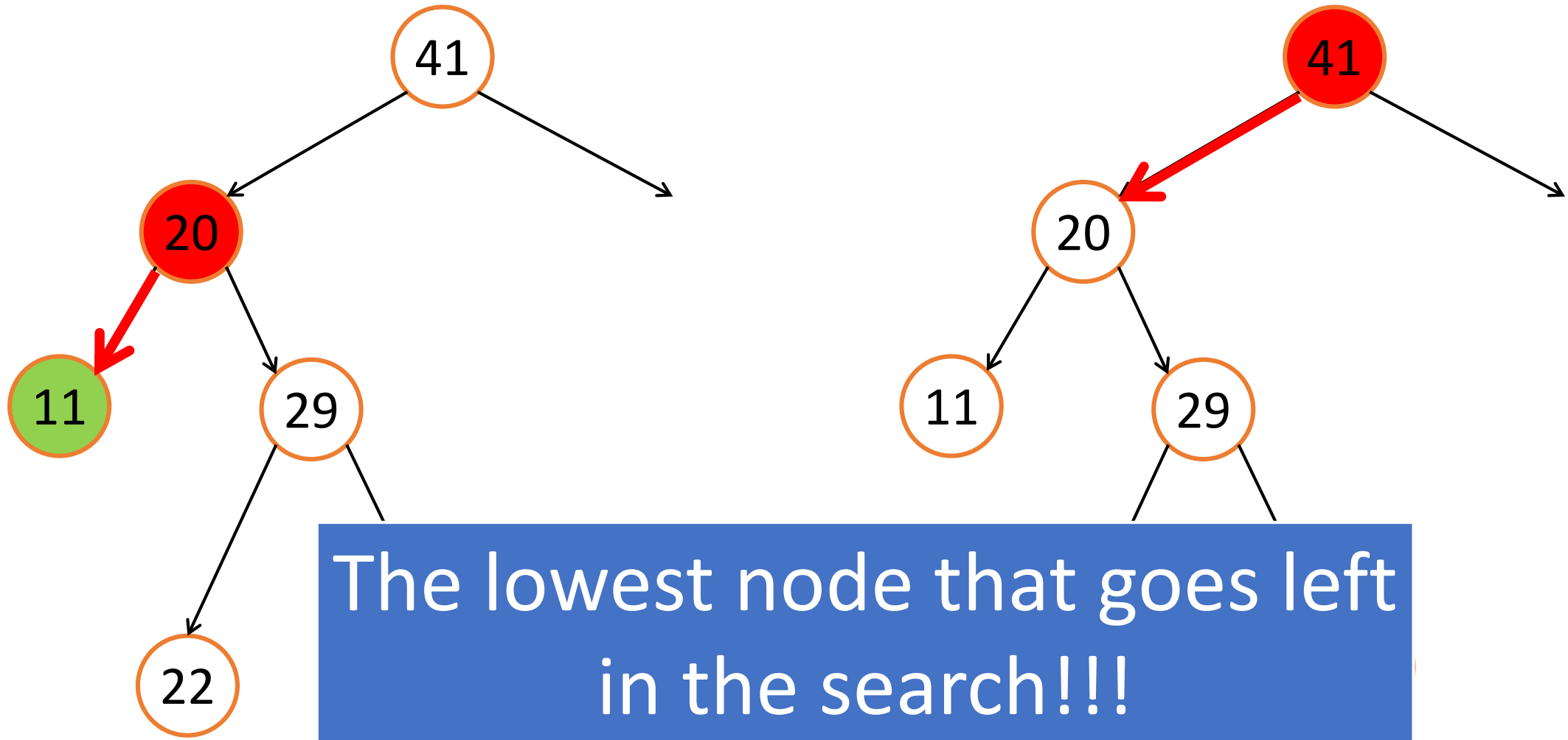


Successor Queries: No Right Child?

- Which one is the successor of 32?

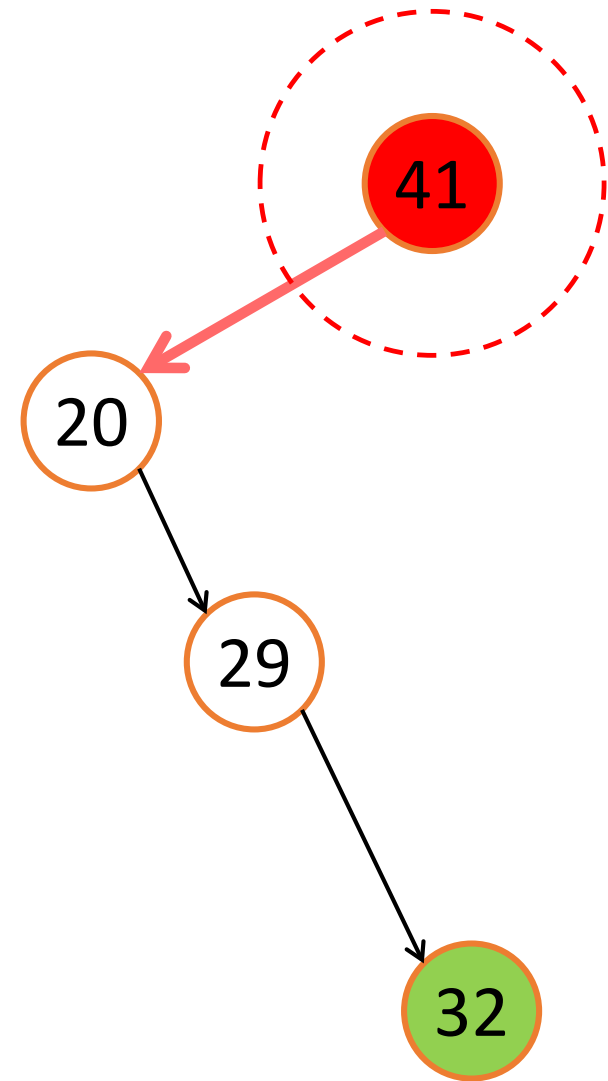


What is the common pattern?



Successor Queries

- Follow the search to find the node for x
- If the node has a right child
 - return the max of its right subtree
- Follow the search path from the root, find the last parent that goes left to find x



Works for The key NOT in the Tree

- `successor(33)`

