# CSCI 2270: Data Structures

Recitation #8 (Section 101)

# Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **From this week onwards**
  - **Office Hours - 12pm to 2pm on Mondays in ECAE 128**
  - **Office Hours - 12:30pm to 2:30pm on Fridays in ECAE 128**

# Office Hours Friday

# Midterm 1

- **Highest amongst all the sections.** Kudos to you guys!
  - Avg: 79.26
  - Median: 83

- Meta-level Analysis on the coding questions
  - **Q7 : Array Halving**
    i. Checking whether you understand call-by-reference
  - **Q8: Insertion in a doubly LL**
    i. Checking whether you understand & can handle LL
  - **Q9: Swapping minimum node and the head node in LL**
    i. Checking whether you can handle pointers properly

# Logistics

- There will probably be another make up midterm towards the end of the semester.
  - It is not mandatory. You can take it if you wish to improve your grade.
  - I would advise anyone who gets <65 in Midterm 1 to take the make up midterm exam.

# Logistics

- How many of you are still interested in seeing your midterm 1 paper?
  - Only a handful of students came on Monday.

# Logistics

- How many of you are still interested in seeing your midterm 1 paper?
  - Only a handful of students came on Monday.

- I have arranged **special/extra office hours on Friday from 2:30pm-3:30pm in ECAE 129** for those who want to see their midterm 1 papers.
  - If this doesn't work for you, send me an email to schedule a meeting.

- So, this Friday
  12:30pm - 2:30pm in ECAE 128 - Actual office hours (for help with the assignment)
  2:30pm - 3:30pm in ECAE 129 - Extra office hours for seeing your midterm paper

# Logistics

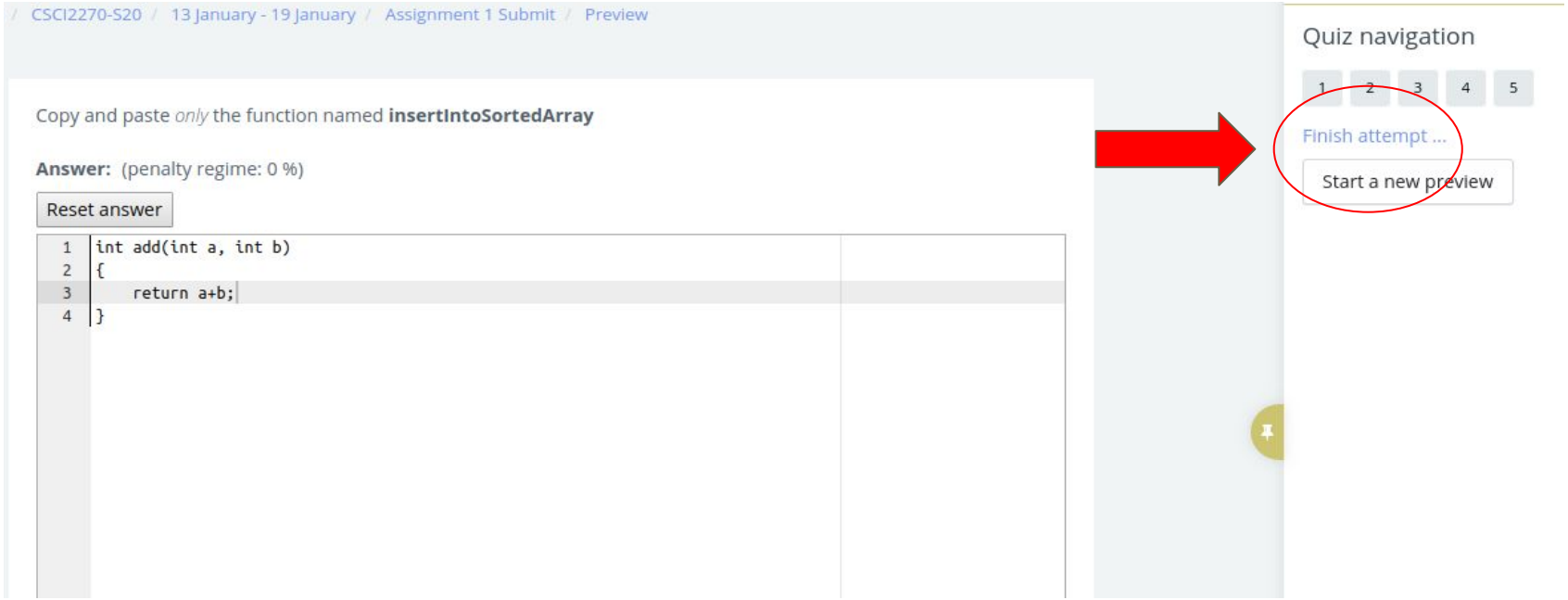
- Thank you for filling out FCQs (all anonymous)

- Few noteworthy feedbacks:
  - “Something that my TA did last semester that was extremely helpful was in addition to office hours, he had one on one appointments with his students who needed extra help. If my TA did that, it would be very helpful.”
    - I am more than happy to do that. **Just send me an email.** I have already done that with a couple of students who reached out to me and wanted help.
  - “Provide more office hours and encourage us to go more”
    - Since the office hours are on two different days now, hopefully I will see more of you.
  - “Go through the material slower and less rushed” (most common)
    - How many of you feel the exact same way?
  - “Talk a little less”
    - Well….

# Logistics

- **Assignment 6 is due on Sunday, March 8 2020, 11:59 PM.**
  **GOOD LUCK!**

# Please click on “Finish Attempt” after you are done!



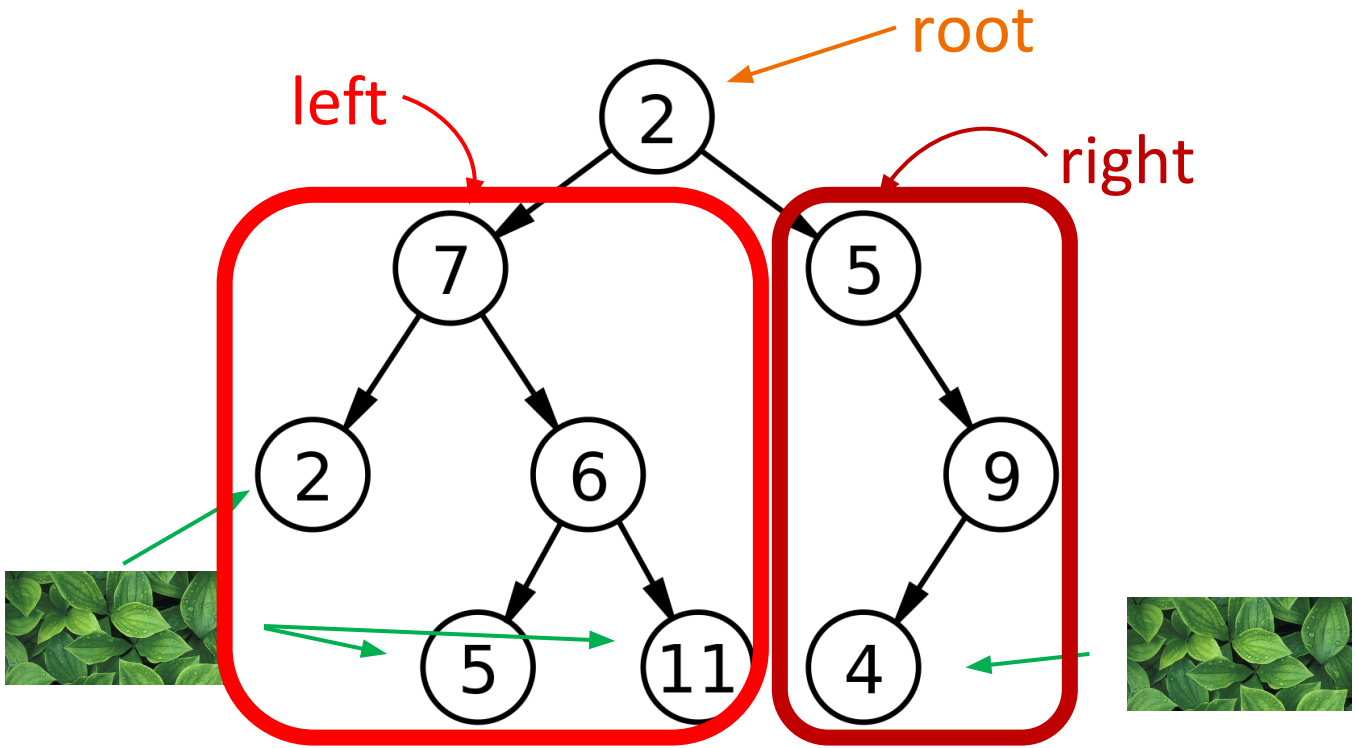
# Any questions on Logistics?

# Today's Agenda

- Review (20 ~ 30 mins)
  - Binary Search Trees (BST)
  - Search in a BST
  - Insertion in a BST
  - Search complexity in a BST
  - Deletion in a BST

- Exercise
  - **Silver Problem** - Implement "deleteTree" function which deletes all the nodes of the tree.
  - **Gold Problem** - Check if a given Binary tree is a BST or not

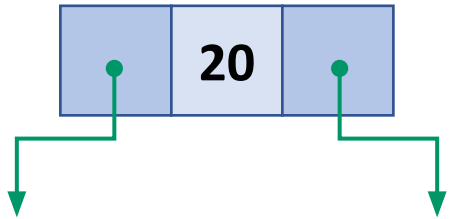
# Binary Trees




Tree rooted at left child is called left subtree. (Same for right)

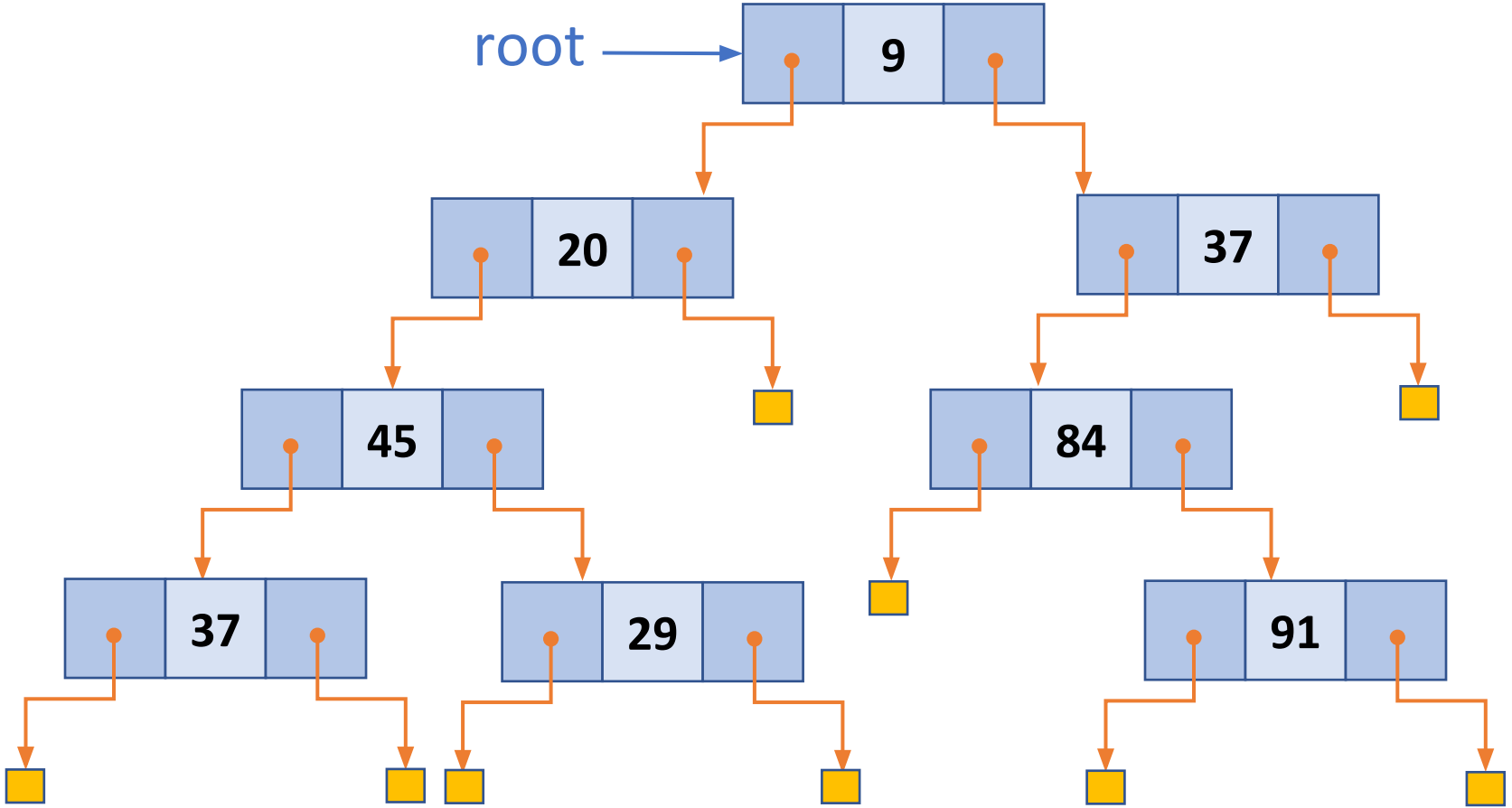
# Implementation of Binary Trees

- What does each node look like?

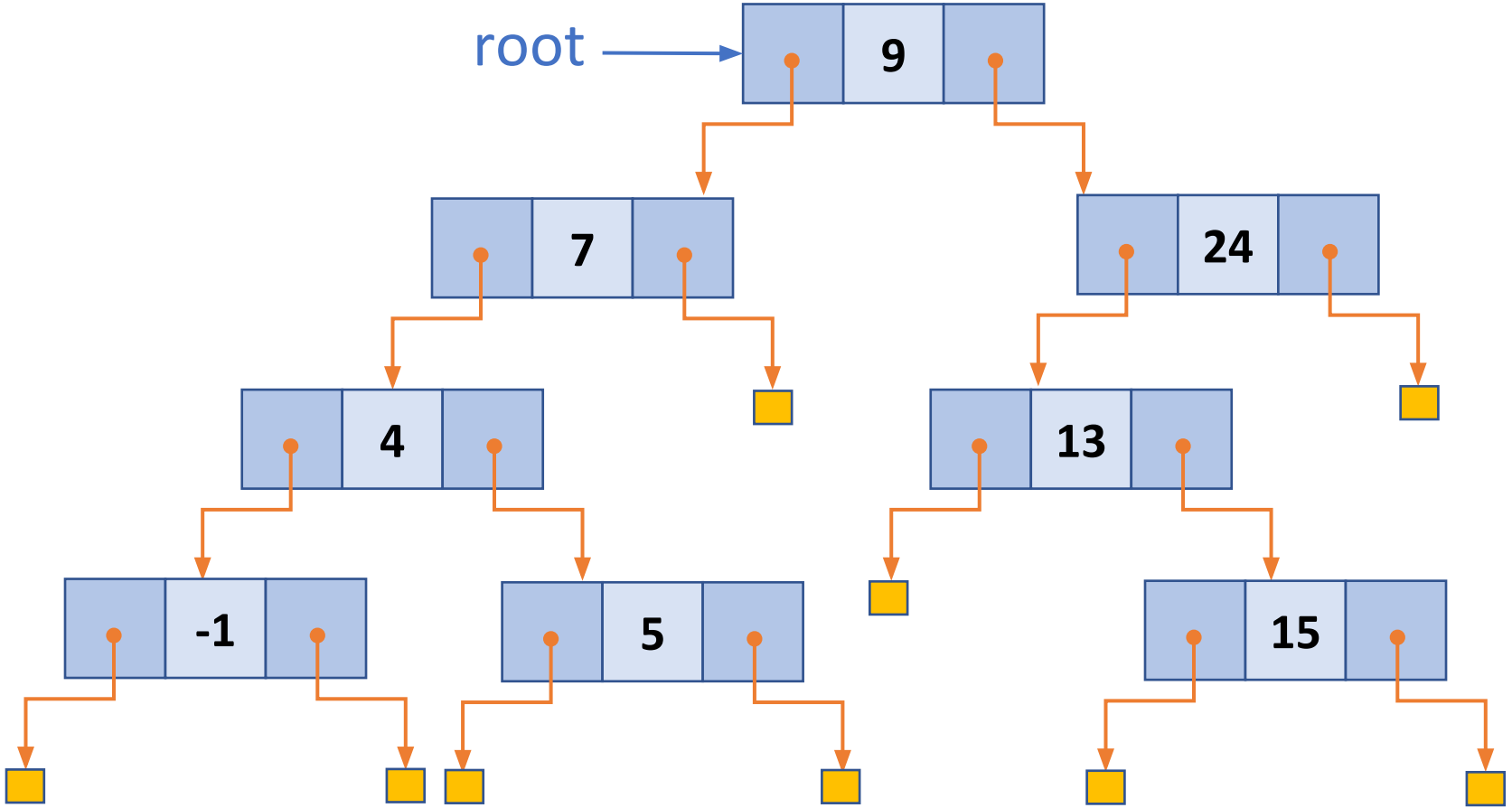



```
struct Node {
   int data;
   Node * right;
   Node * left;
}
```

# Implementation of Binary Trees

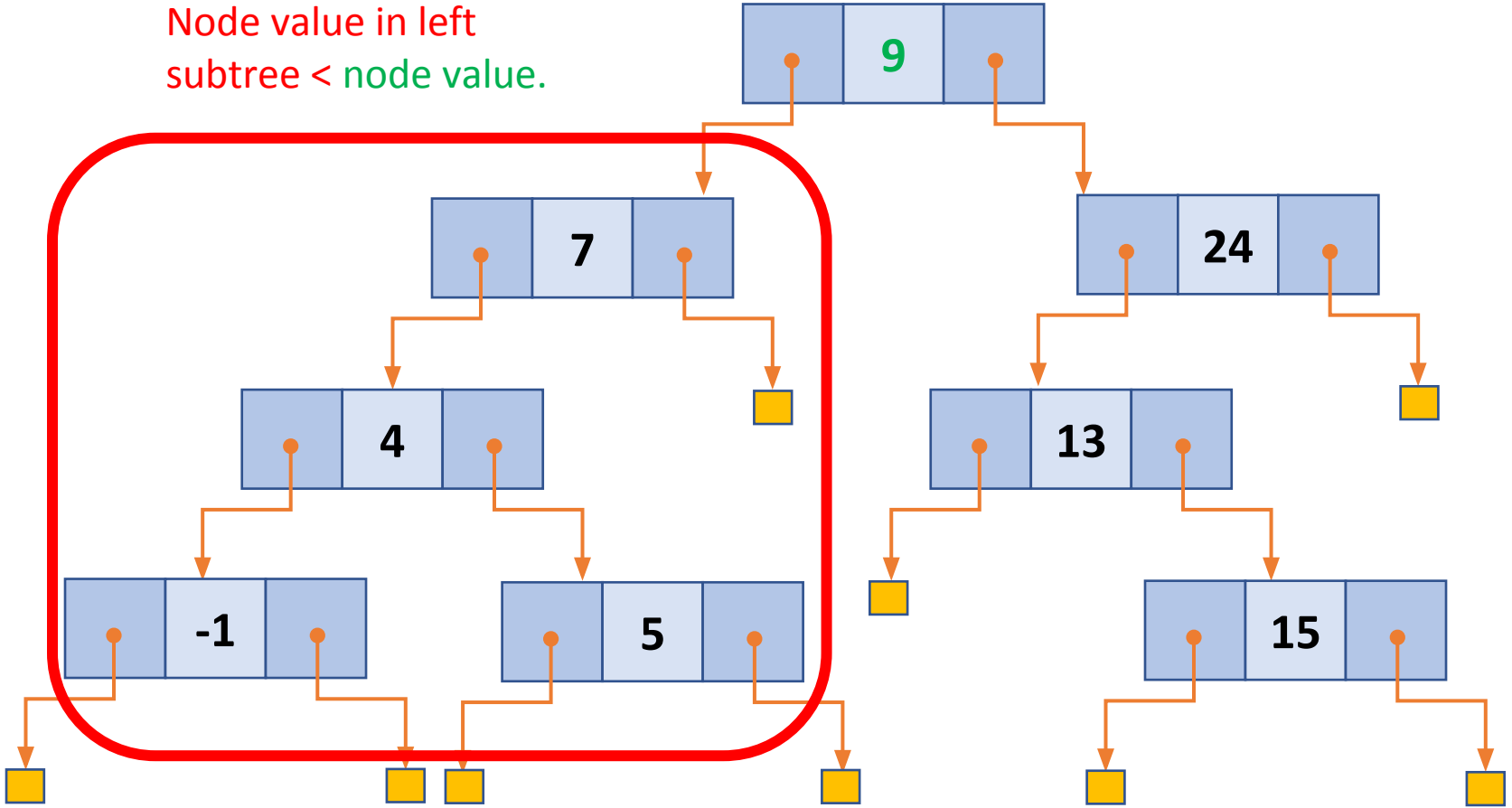


# What is a Binary Search Tree?



# BST: For each node …

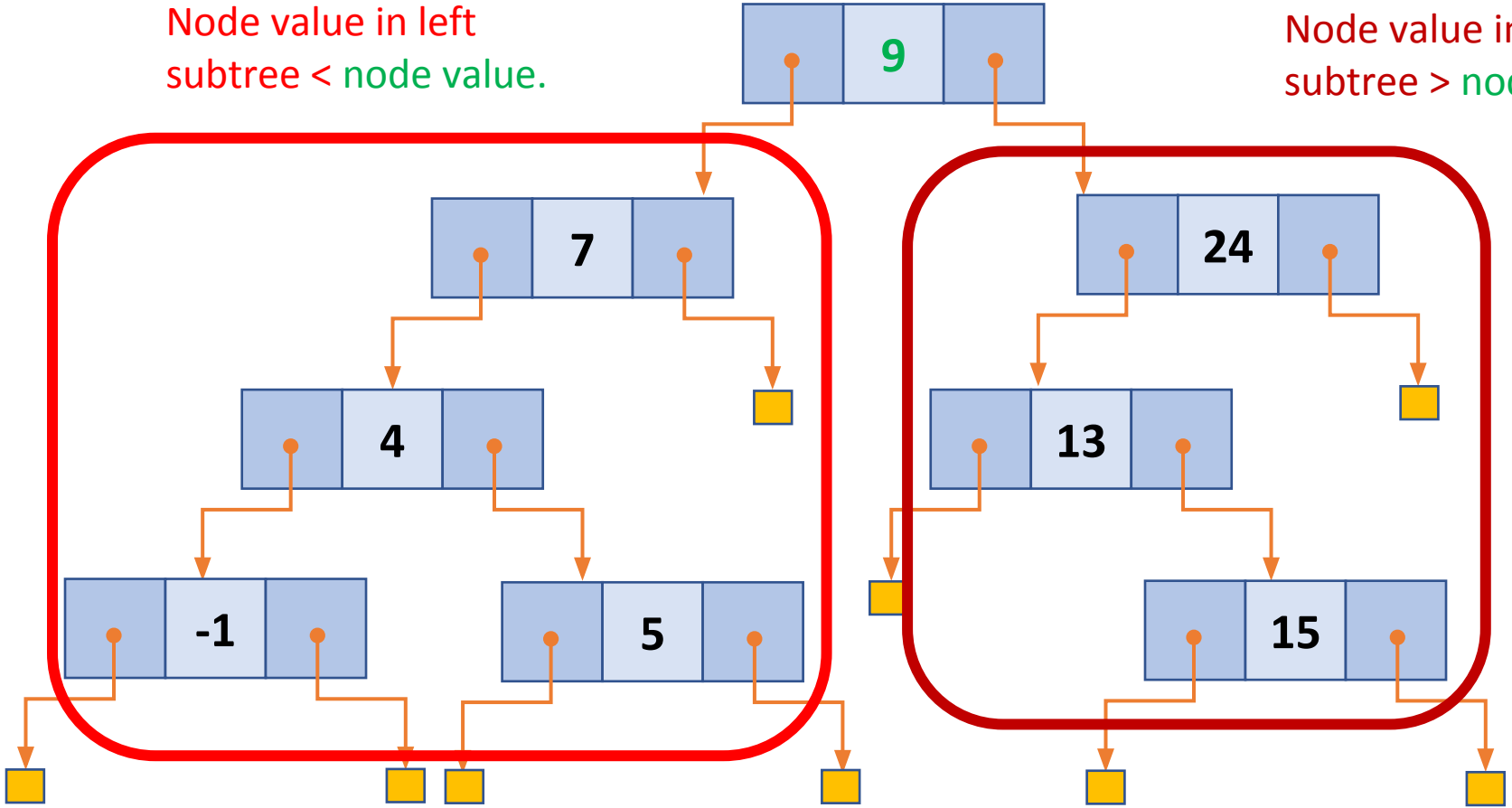
Node value in left subtree < node value.



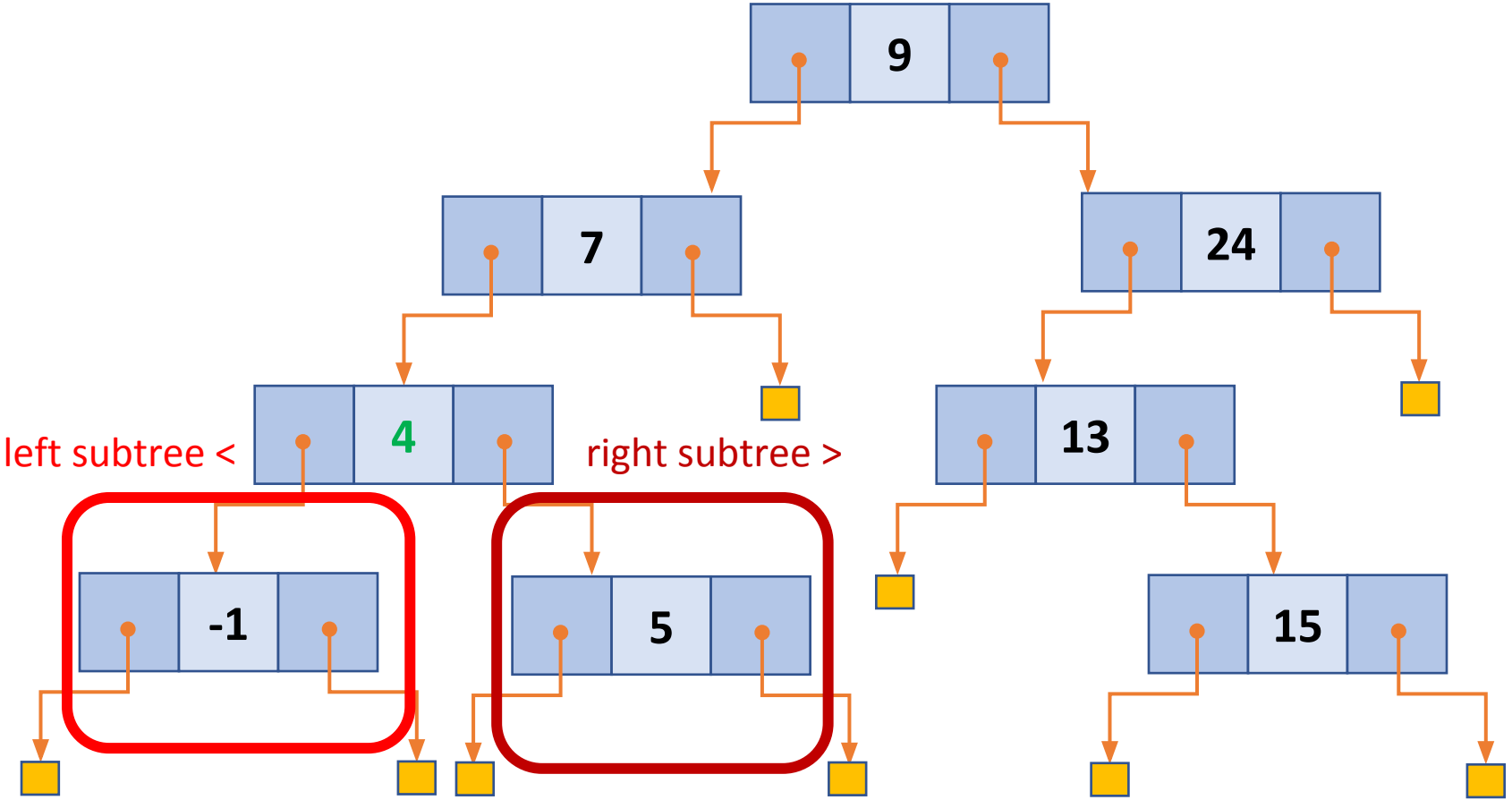
# BST: For each node …

Node value in left subtree < node value.

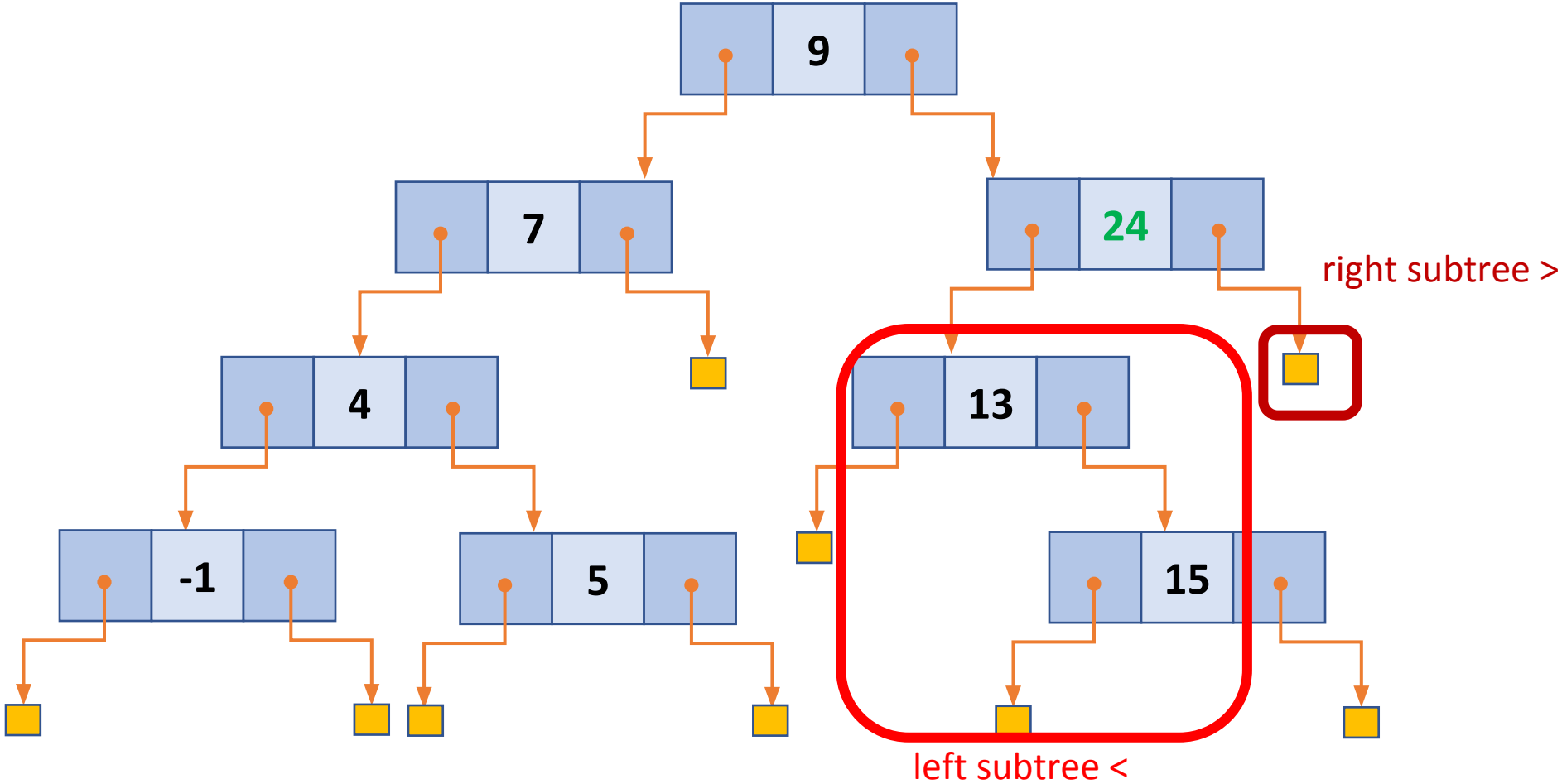
Node value in right subtree > node value.



# BST: For each node …



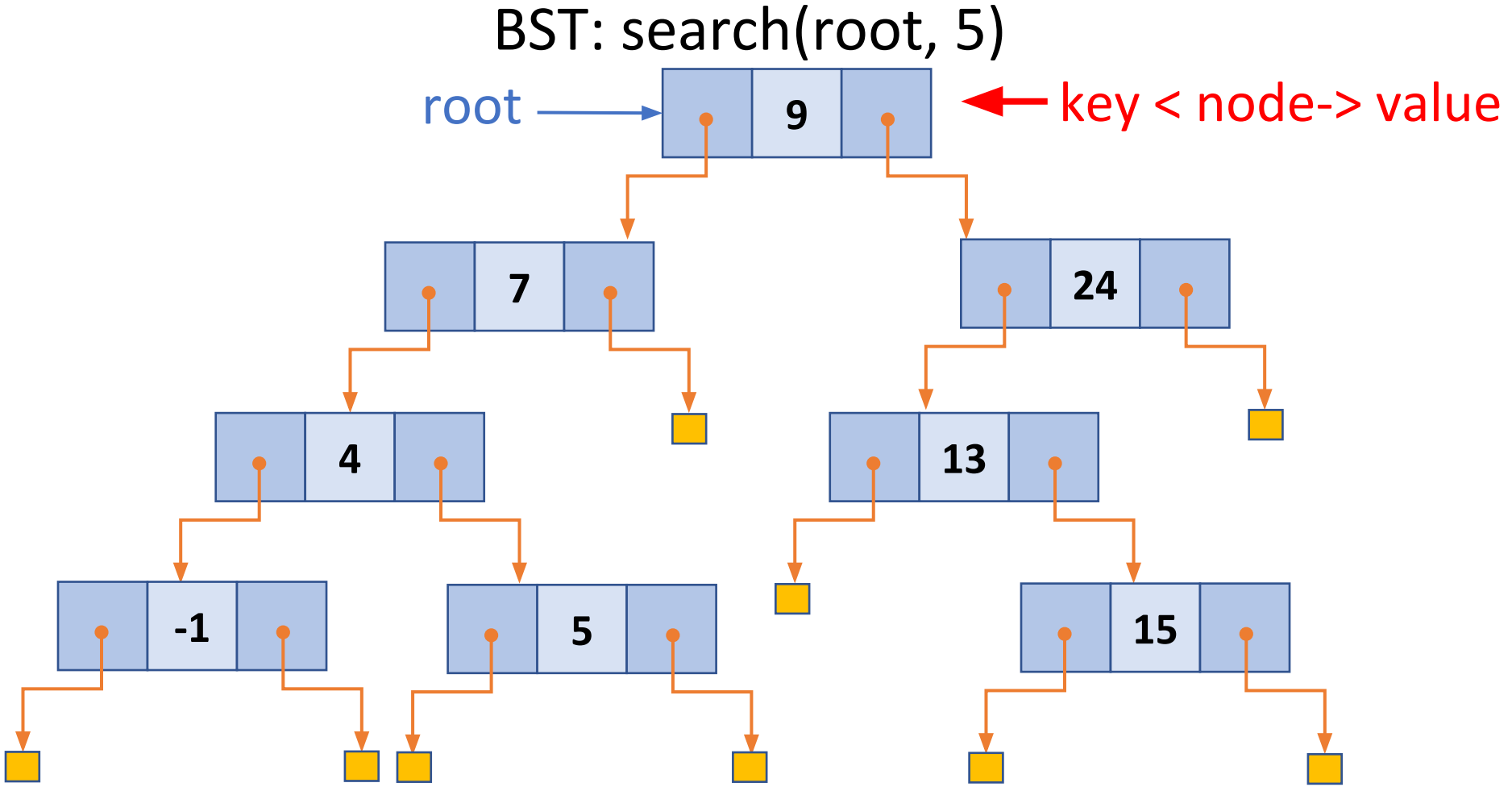
# BST: For each node …



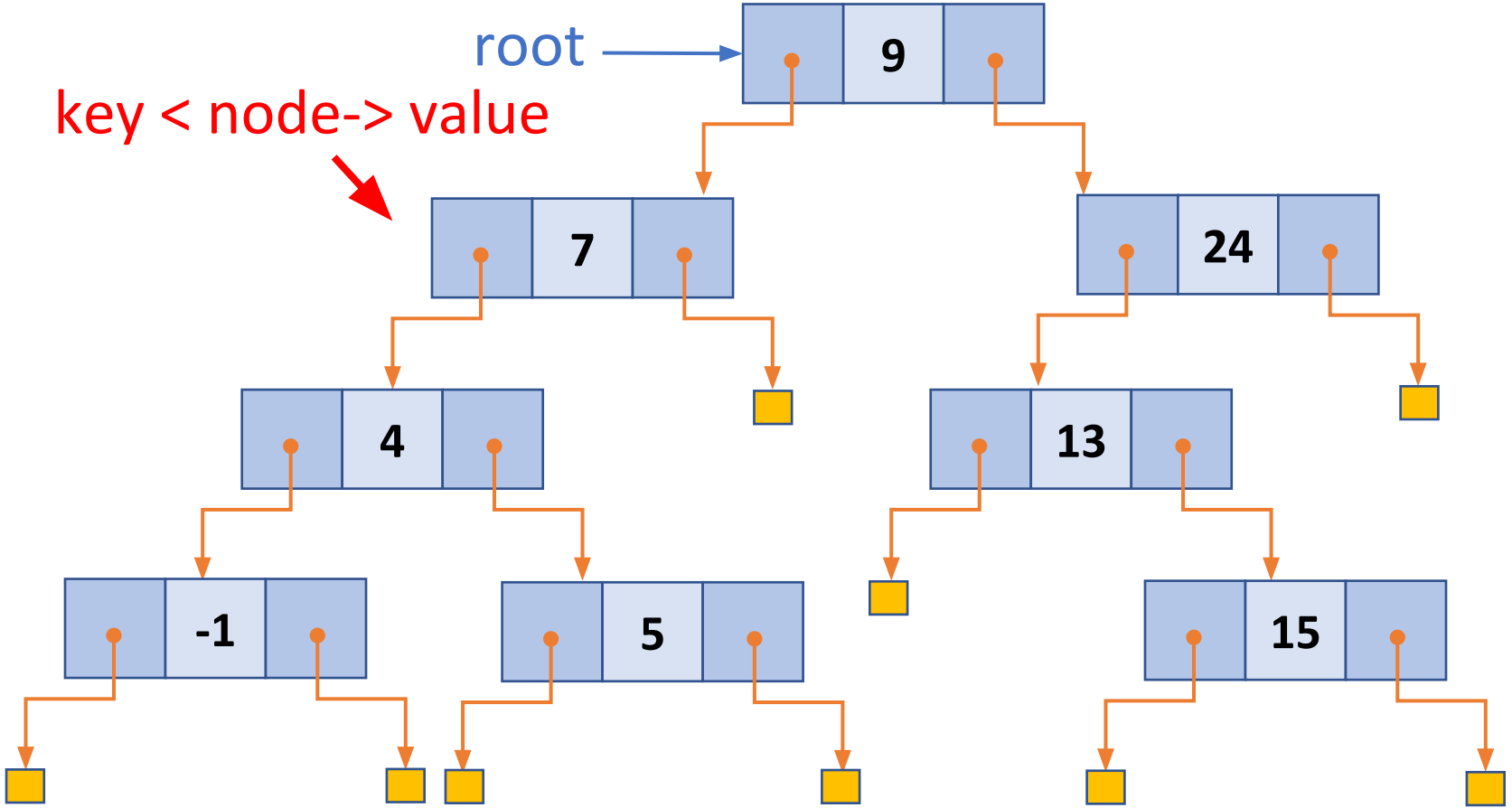
# BST: Searching a Node

```
Node * search(Node * node, int search) {
 /* Base case: leaf node */
 if (node == NULL) return node;
 /* Base case: Node found! */
 if (node->value == search_key) return node;
 /* Search in the left subtree */
 if (search_key < node->value) return search(node->left);
 /* Search in the right subtree */
 if (search_key > node->value) return
search(node->right);
}
```

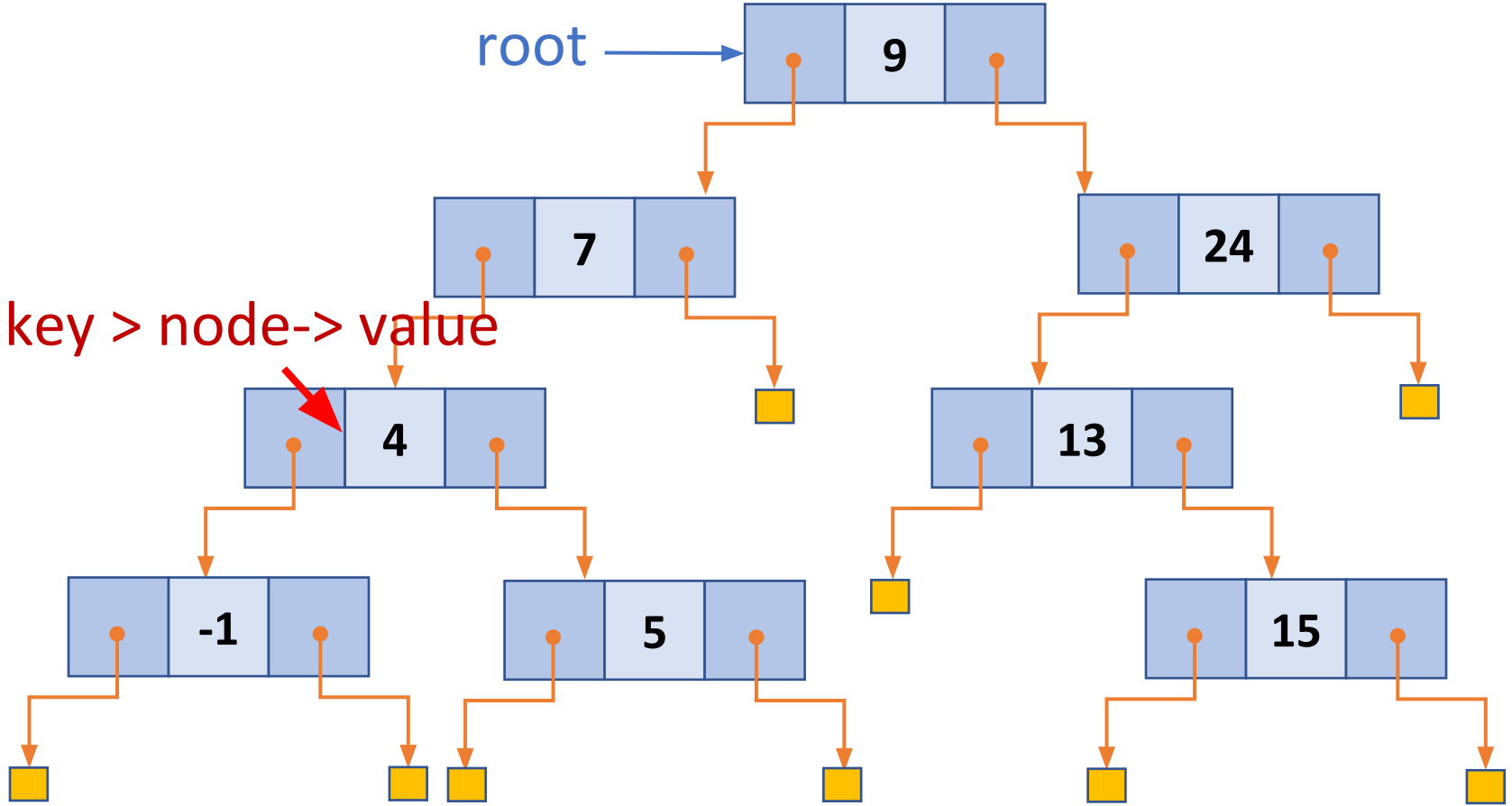
**search(root, 5);**


BST: search(root, 5)
root
9
key < node-> value
7
24
4
13
-1
5
15

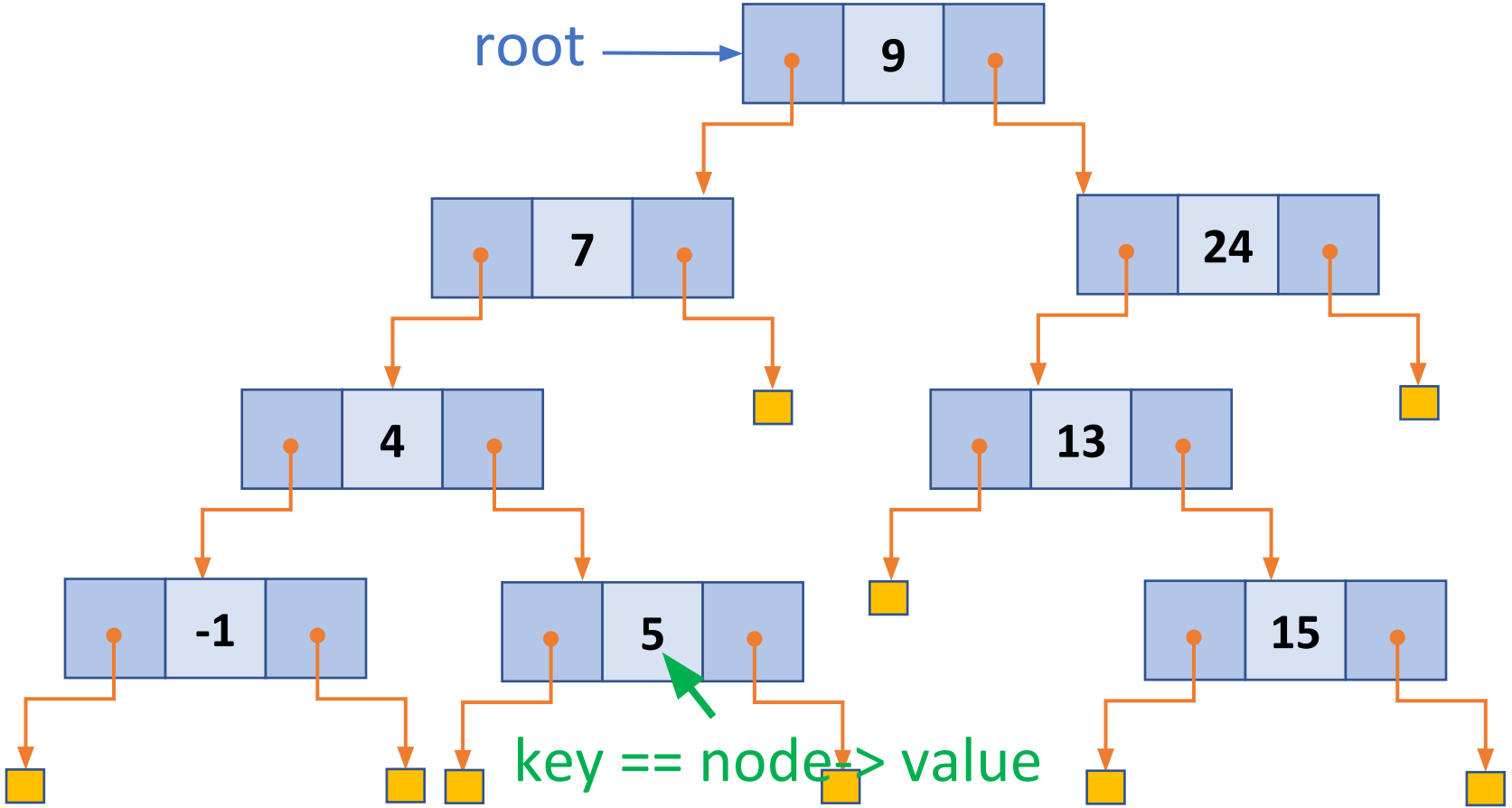
# BST: search(root, 5)



# BST: search(root, 5)



# BST: search(root, 5)



# Any Questions?

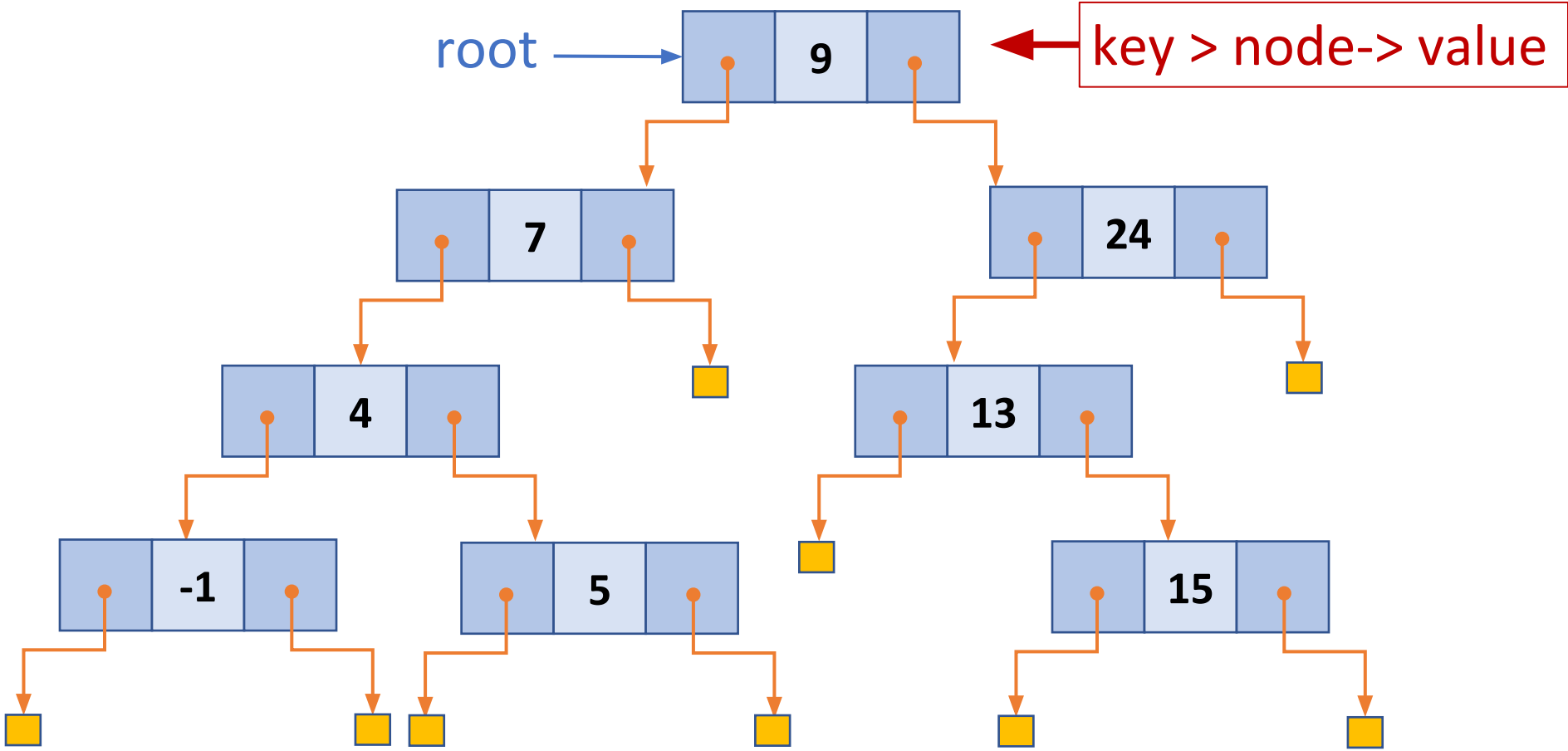
# BST: Inserting a new node

- Traverse the tree using a recursive call.

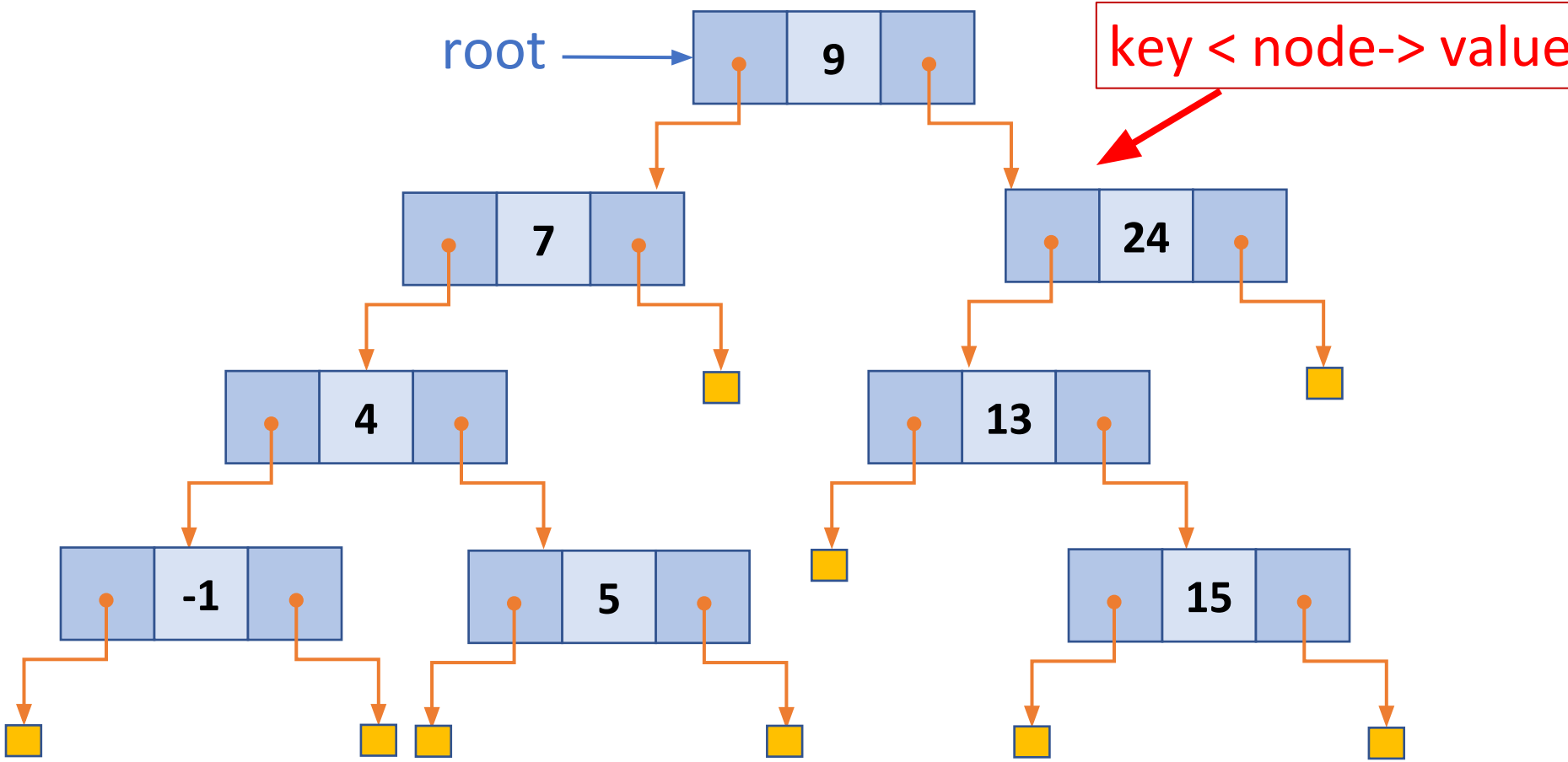
```
Node * addNode(Node * node, int key) {
 /* Base case: leaf node */
 if (node == NULL) return new Node(key);
 /* Search in the left subtree */
 if (search_key < node->value)
     node->left = addNode(node->left, key);
 /* Search in the right subtree */
 if (search_key > node->value)
     node->right = addNode(node->right, key);
 return node;
}
```

**addNode(root, 14);**

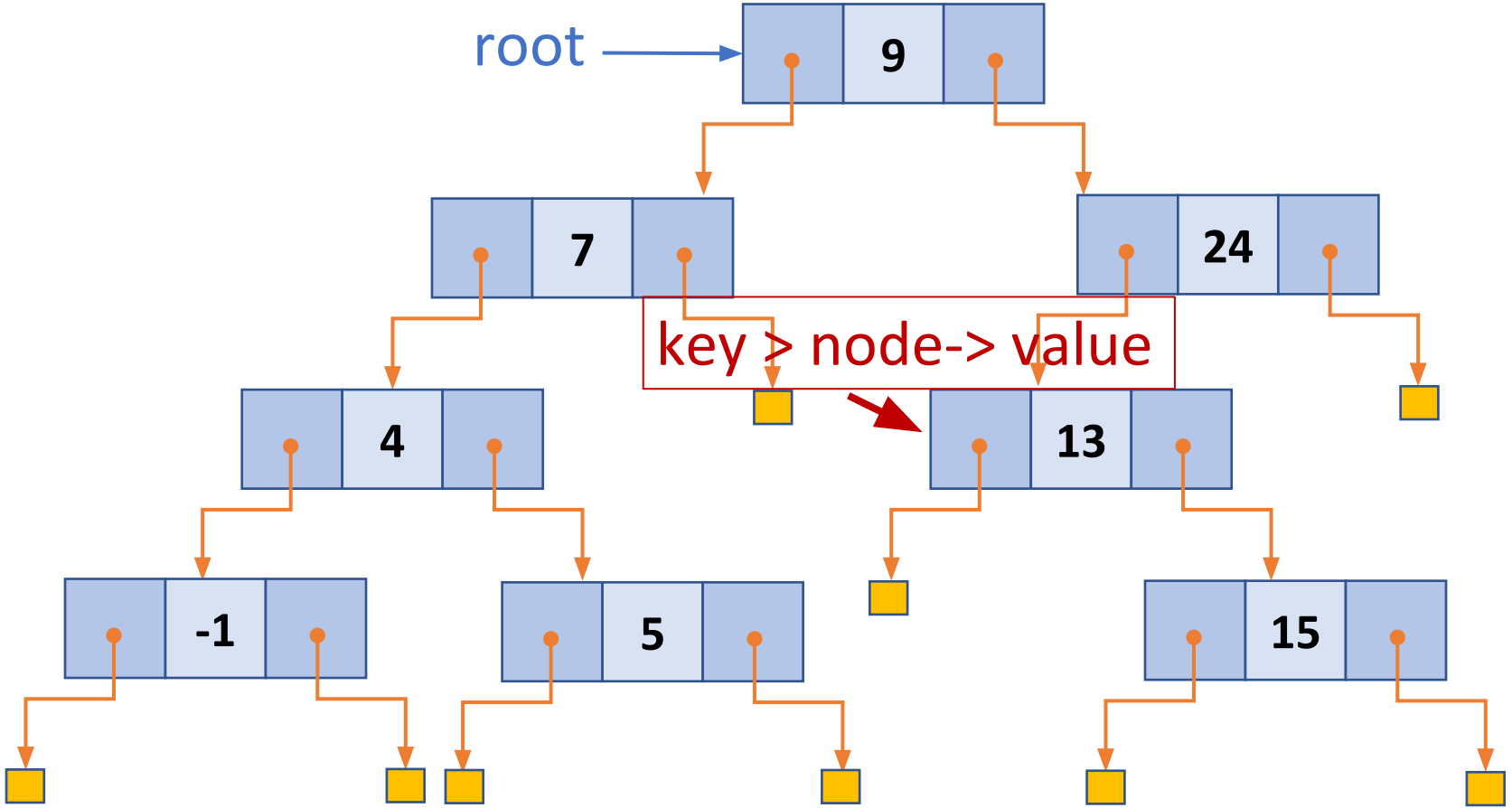
# BST: addNode(root, 14)



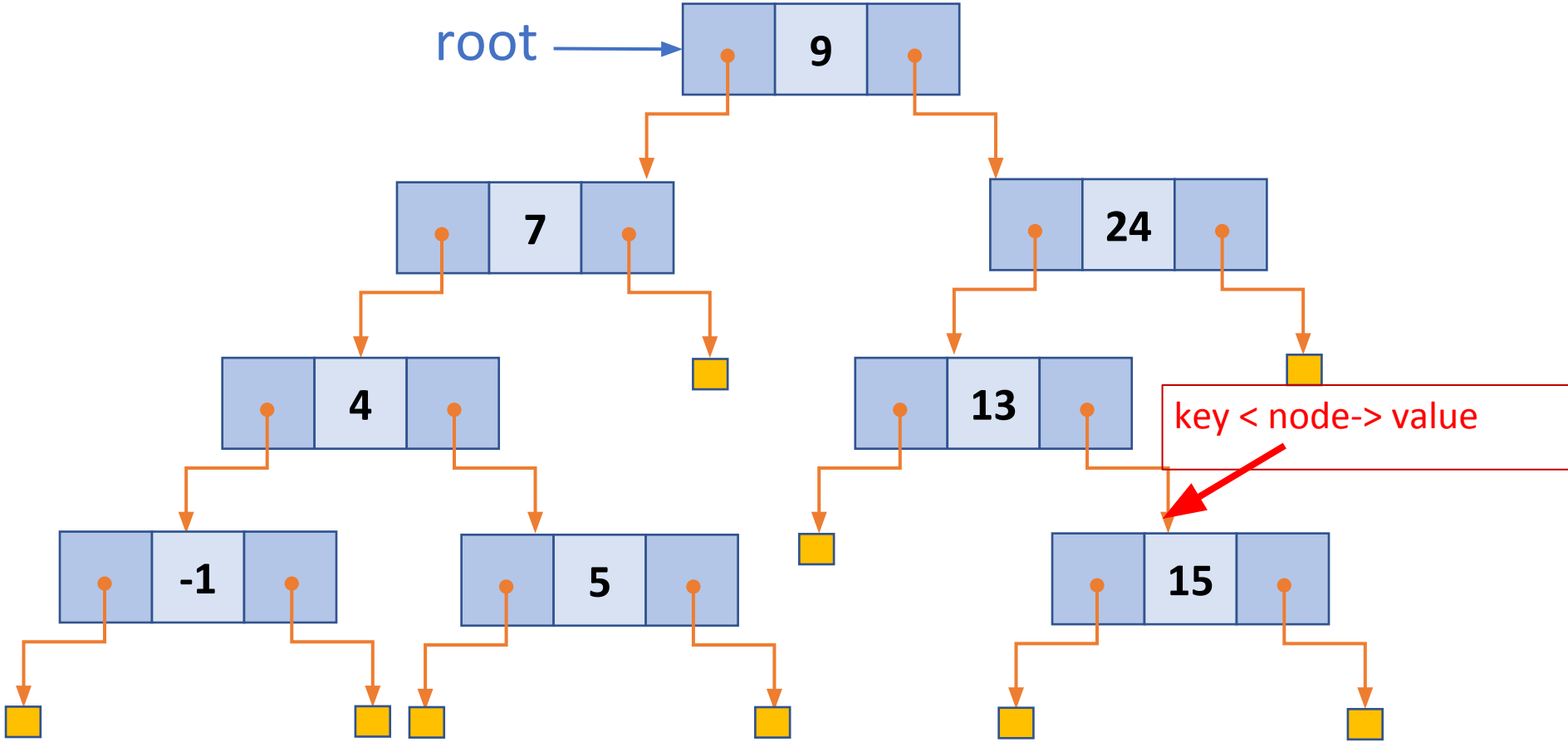
# BST: addNode(root, 14)



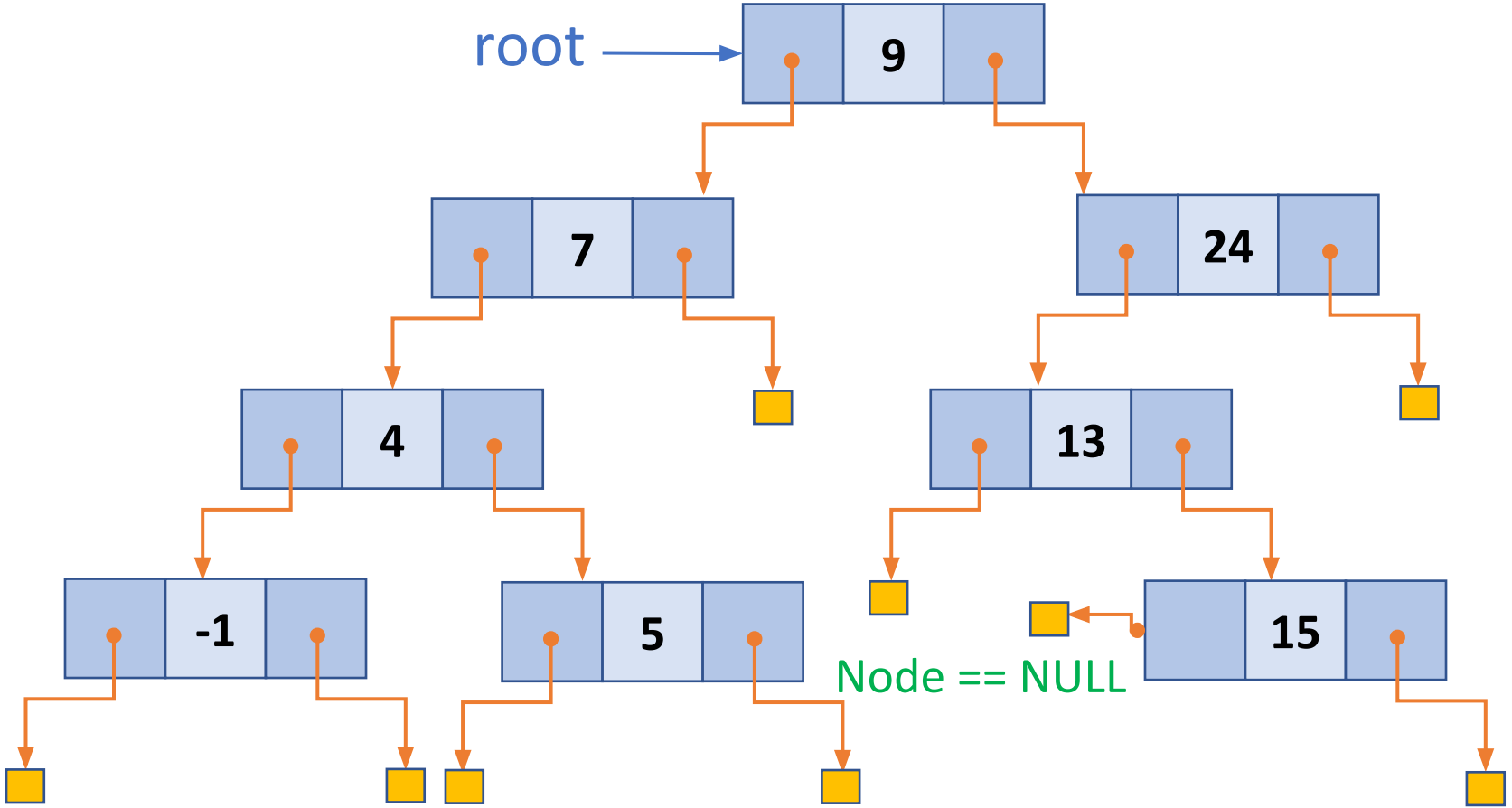
# BST: addNode(root, 14)



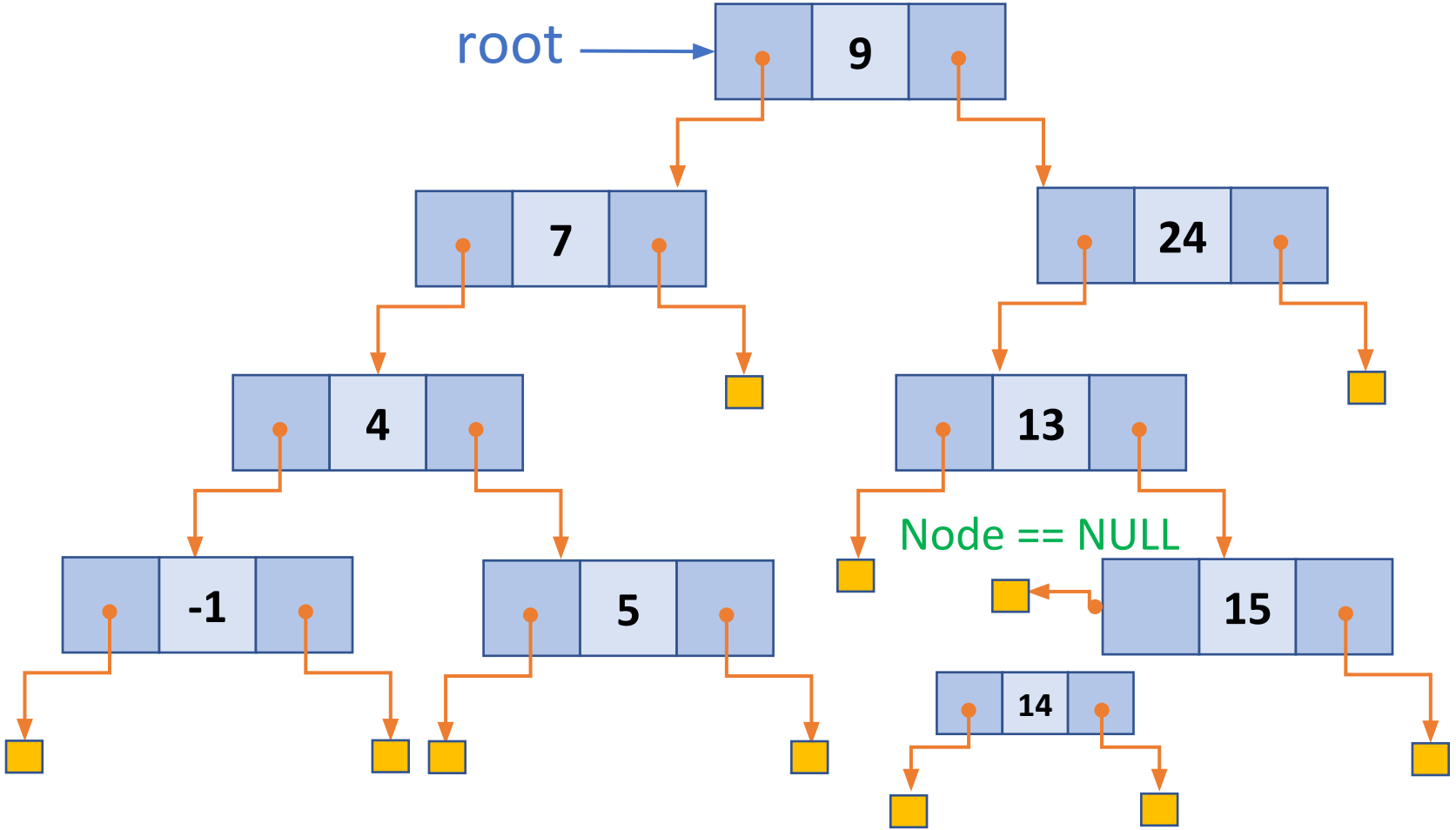
# BST: addNode(root, 14)



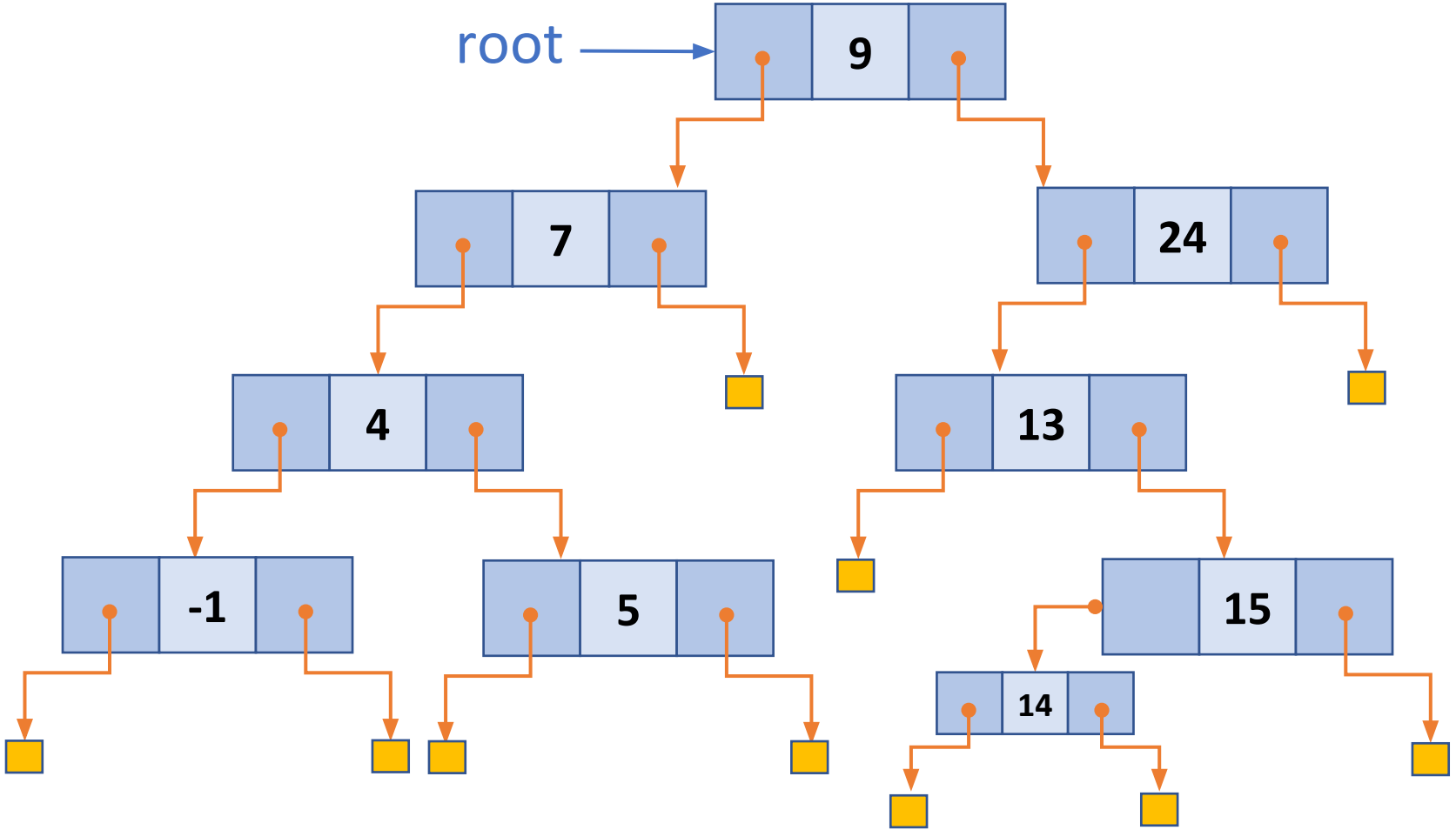
# BST: addNode(root, 14)



# BST: addNode(root, 14)



# BST: addNode(root, 14)



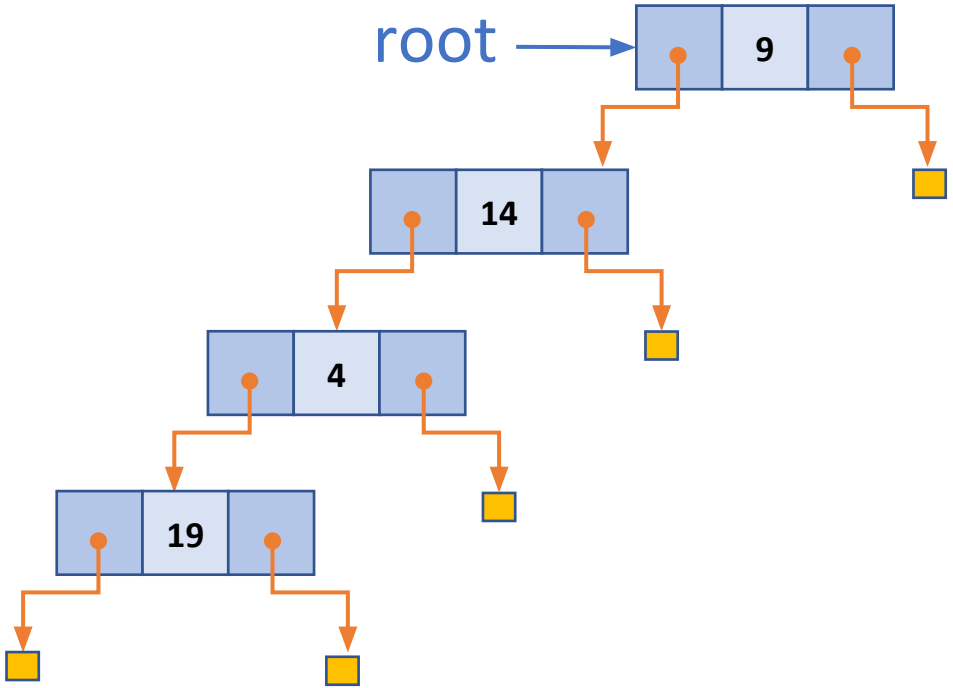
# So why BSTs?

# So why BSTs?

- In a regular tree, the worst case complexity for a search can be high.
  - One special case is : What if each node had a left child, and an empty right child?

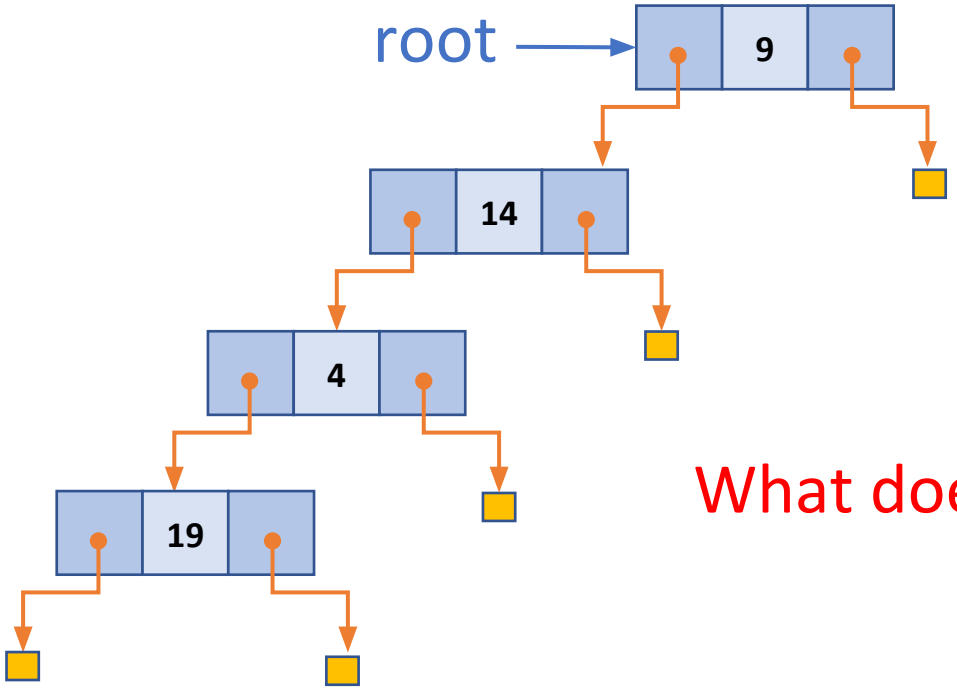
# So why BSTs?

- In a regular tree, the worst case complexity for a search can be high.
- What if each node in a BST has a left child, and an empty right child?



# So why BSTs?

- In a regular tree, the worst case complexity for a search can be high.
- What if each node has a left child, and an empty right child?

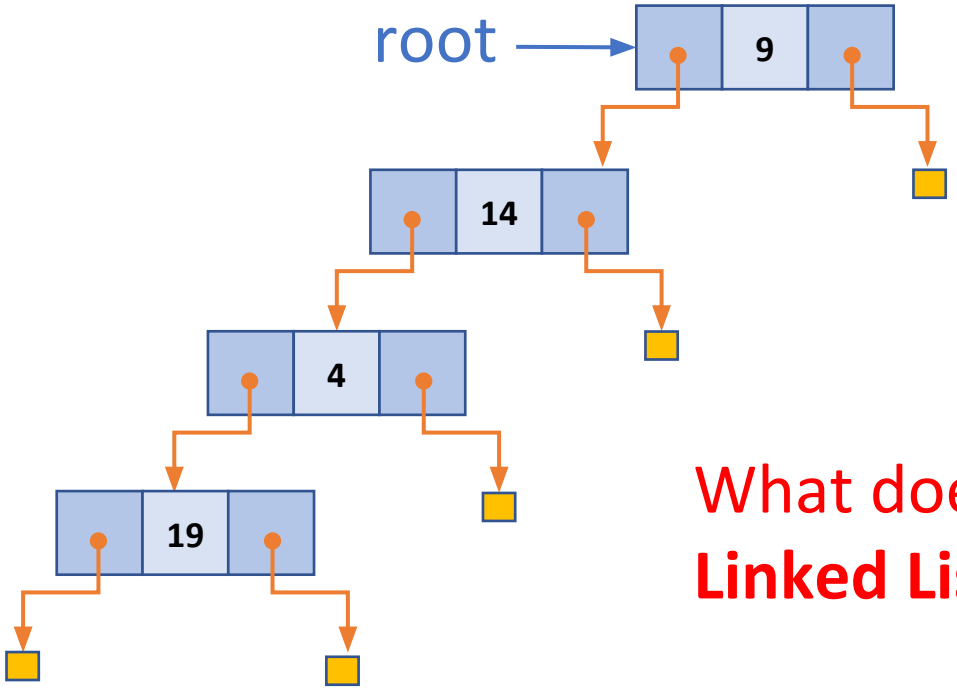



What does this look like?

# So why BSTs?

O(n)!

- In a regular tree, the worst case complexity for a search can be high.
- What if each node had a left child, and an empty right child?




What does this look like? :

**Linked List!**

# BST Complexity

- Imbalanced BSTs also face this issue.
- A BST is constructed based on the order of the data processed.
- Suppose the data observation sequence is a purely descending order:

17, 9, 4, 3, 1, 0

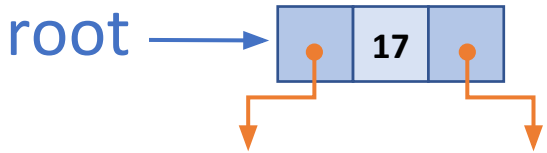
# BST Construction

Data observed:
17, 9, 4, 3, 1, 0, -1

# BST Construction

Data observed:

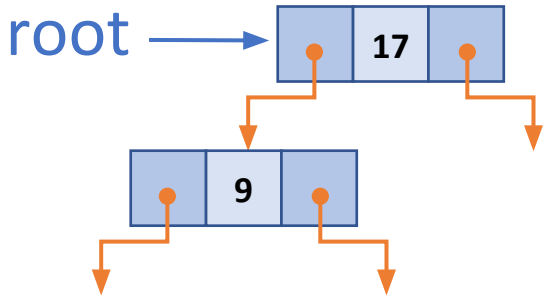
17, 9, 4, 3, 1, 0, -1



# BST Construction

Data observed:

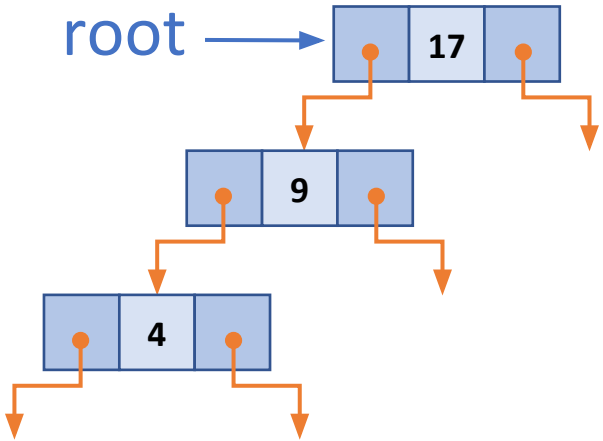
17, 9, 4, 3, 1, 0, -1



# BST Construction

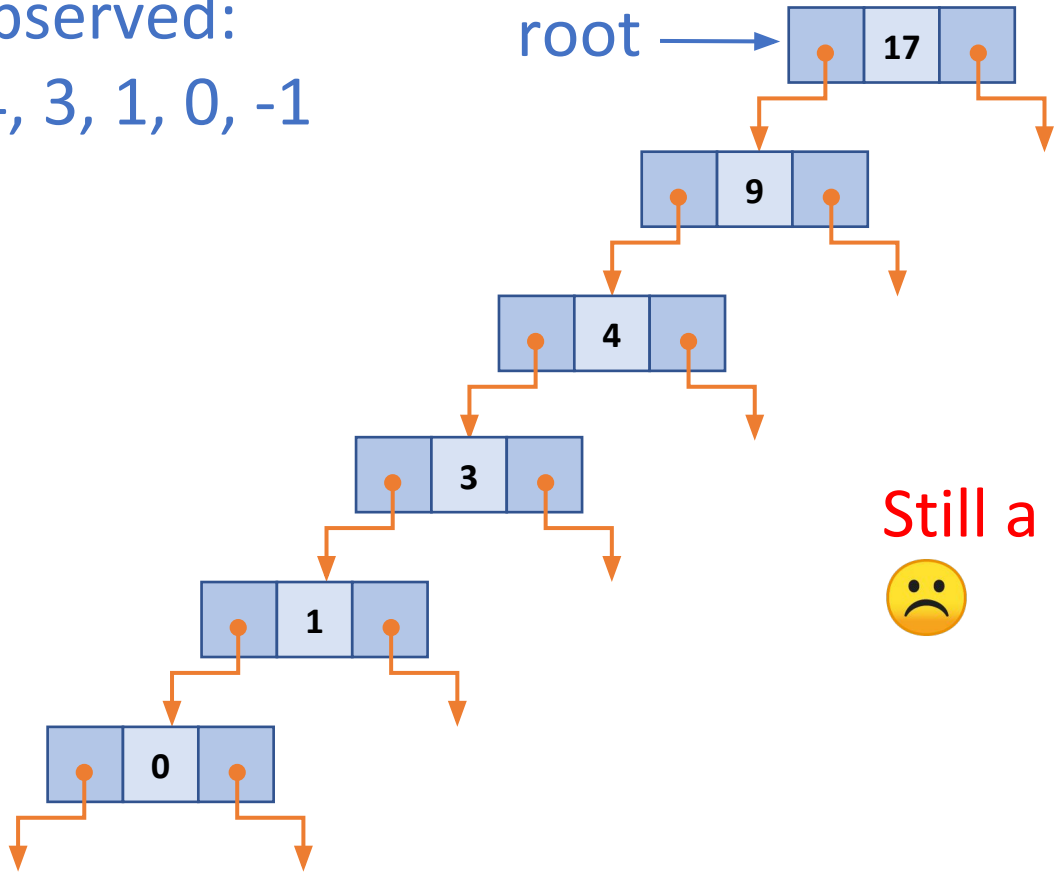
Data observed:

17, 9, 4, 3, 1, 0, -1



# BST Construction

Data observed:
17, 9, 4, 3, 1, 0, -1




Still a linked list!

☹️

# BST Construction

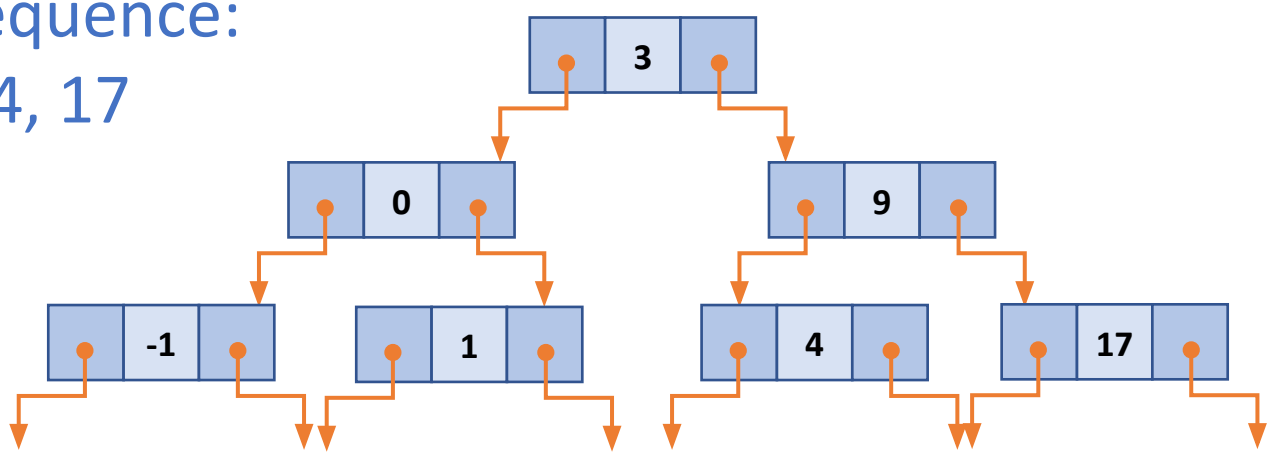
A different sequence:

3, 0, -1, 1, 9, 4, 17

# A more balanced BST!

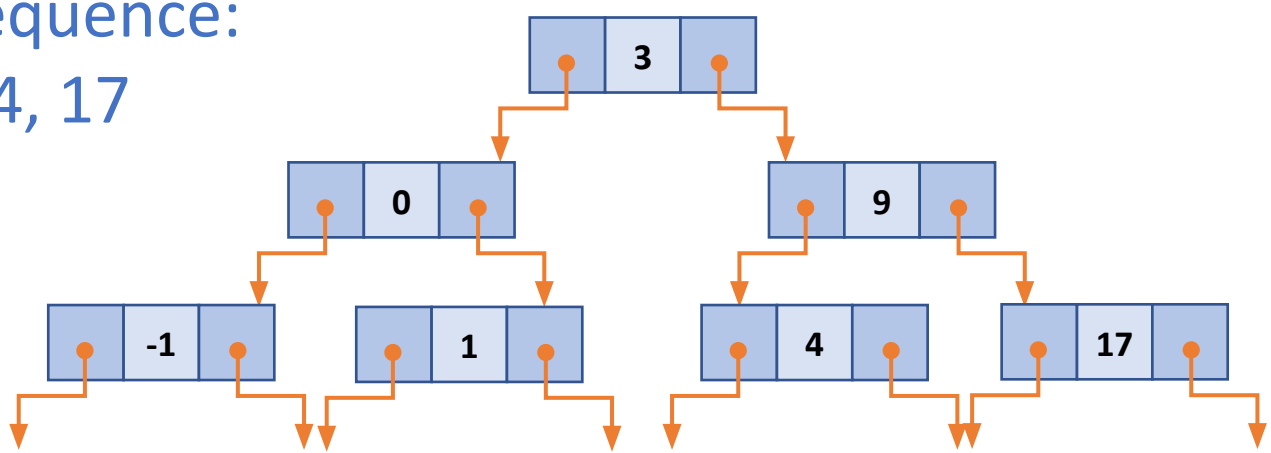
A different sequence:

3, 0, -1, 1, 9, 4, 17



# A more balanced BST!

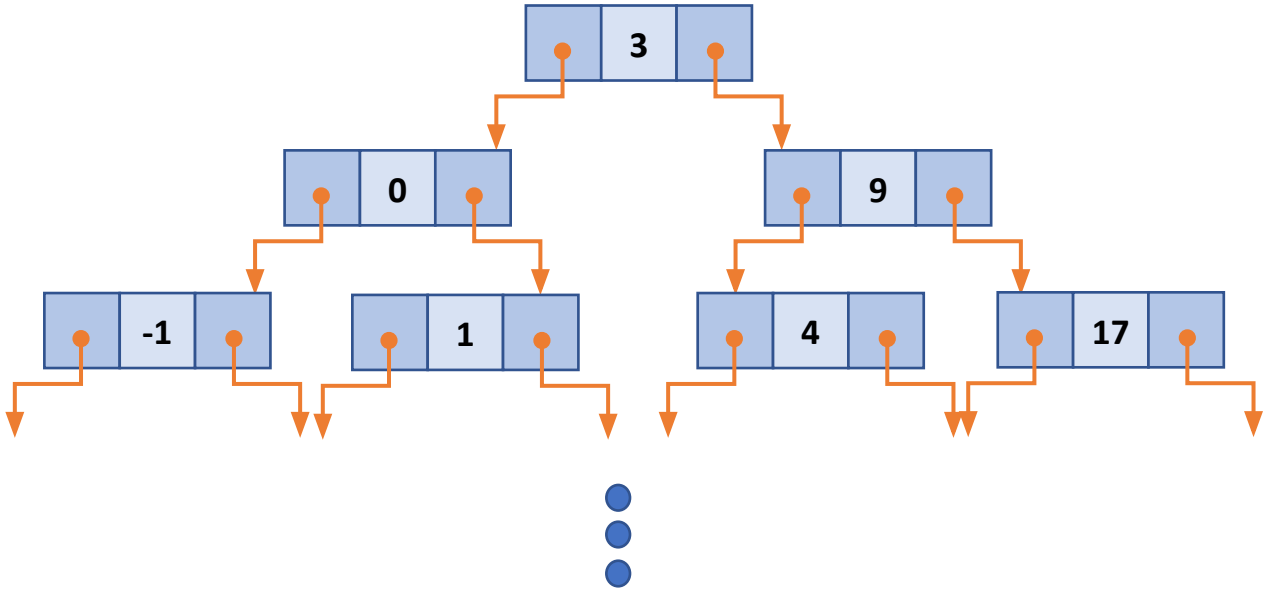
A different sequence:
3, 0, -1, 1, 9, 4, 17




Balanced BST: Except for leaves,
Each node has a left and a right child.

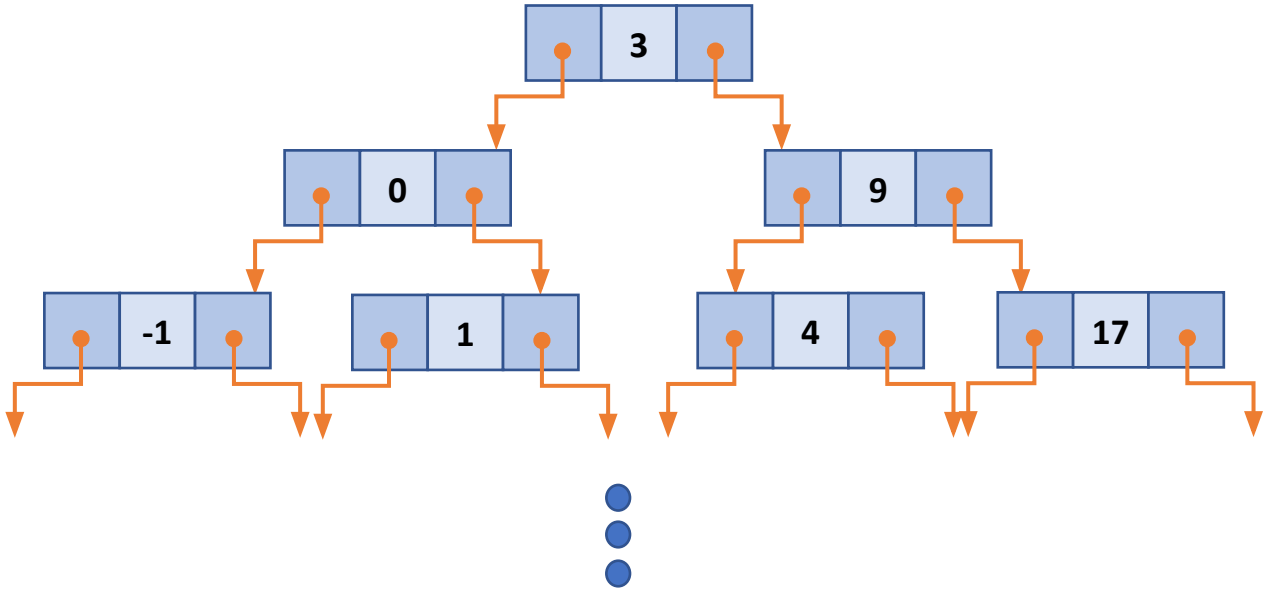
# Complexity of a search in a BST

- Consider a balanced tree of depth **h** and number of nodes **n**.
- What is the complexity of searching in a BST?



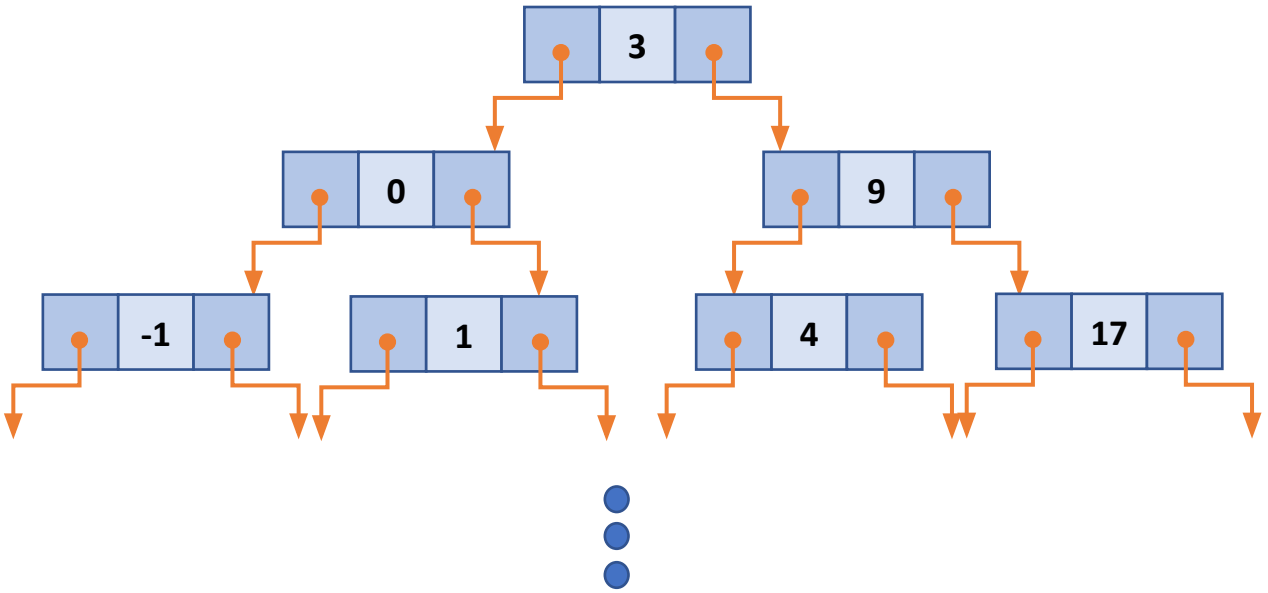
# Complexity of a search in a BST

- Consider a balanced tree of depth **h** and number of nodes **n**.
- What is the complexity of searching in a BST? Answer: **O(h)**



# BST Search Complexity in Terms of n.

- Let's count the nodes at each level!



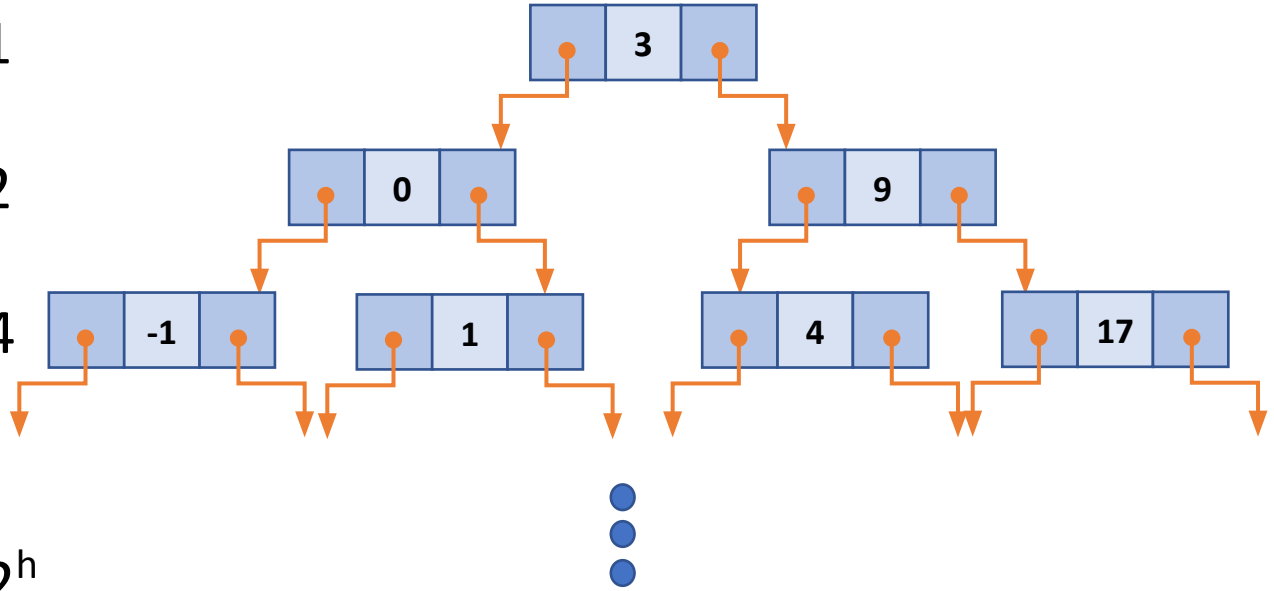
# BST Search Complexity in Terms of n.

- Let's count the nodes at each level!

Nodes = 1

Nodes = 2

Nodes = 4




Nodes = $2^h$

Level = 0

Level = 1

Level = 2

Level = h

# Relationship between depth **h** and nodes **n**.

$$n = 1 + 2 + 4 + \ldots + 2^h$$
$$n = 2^{h+1} - 1$$
$$h = \log_2(n+1) - 1$$

- Complexity of a search in balanced BST is $O(\log_2(n))$
- Linked List search: O(n)

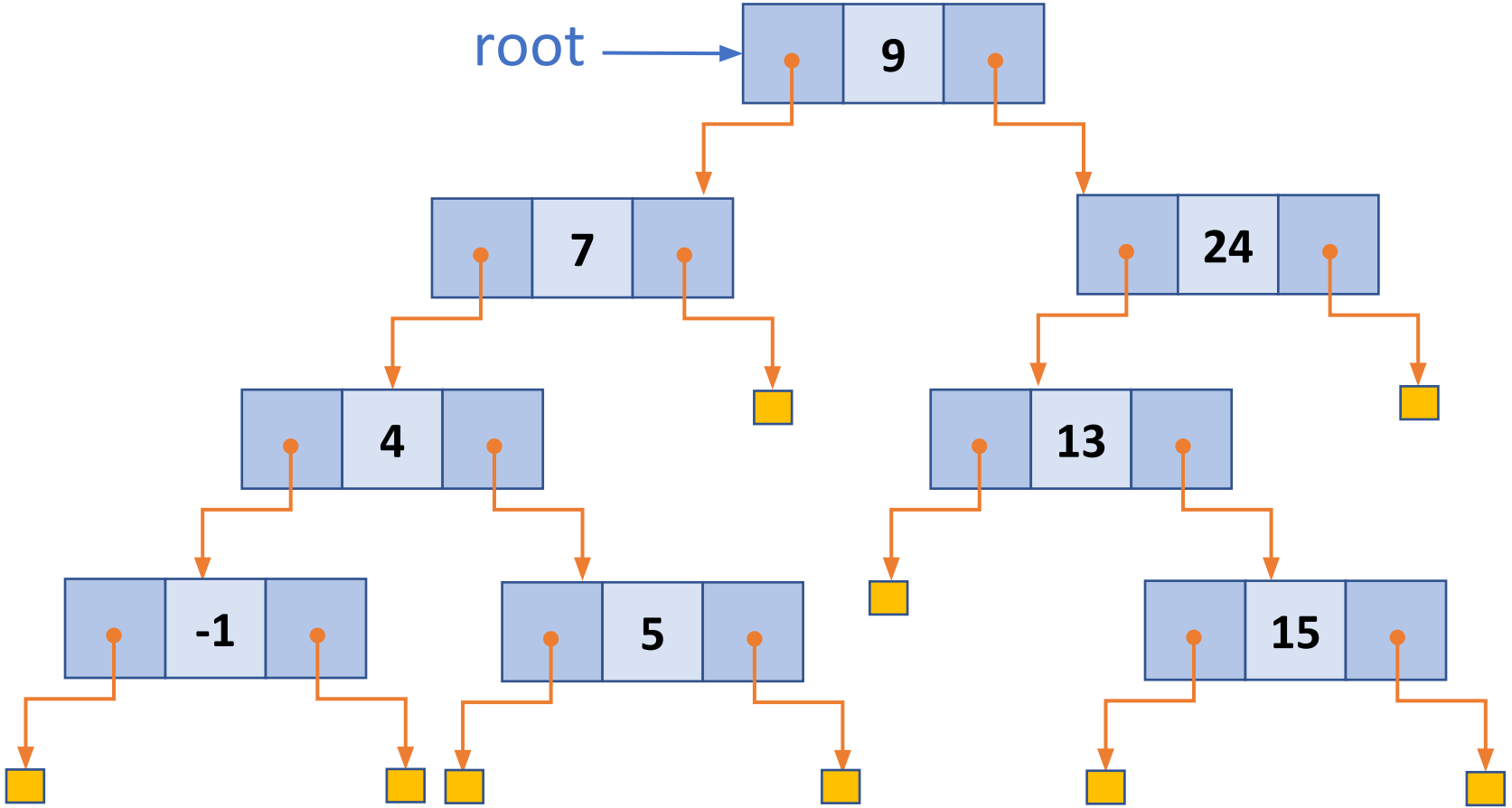
# Deleting a node

- Deleting a node requires rearranging the tree.
- BST structure should be preserved at each node.
- Multiple cases:
  - Deleting a leaf node.
  - Deleting a node with only left child
  - Deleting a node with only right child
  - Deleting a node with both left and right child

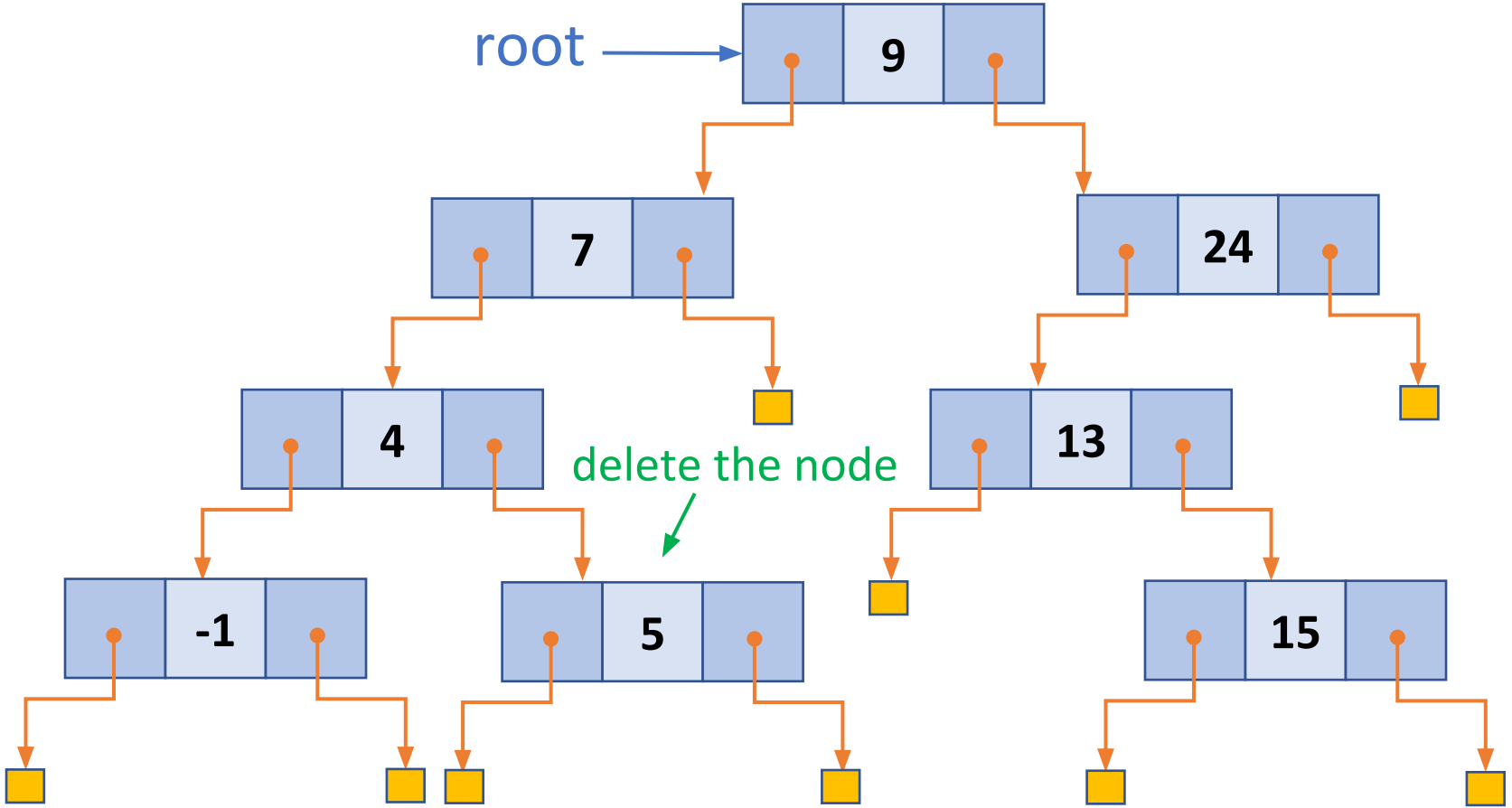
# Case 1: Deleting a leaf node

- Delete the node with the key.
- Set its parent’s child pointer to NULL

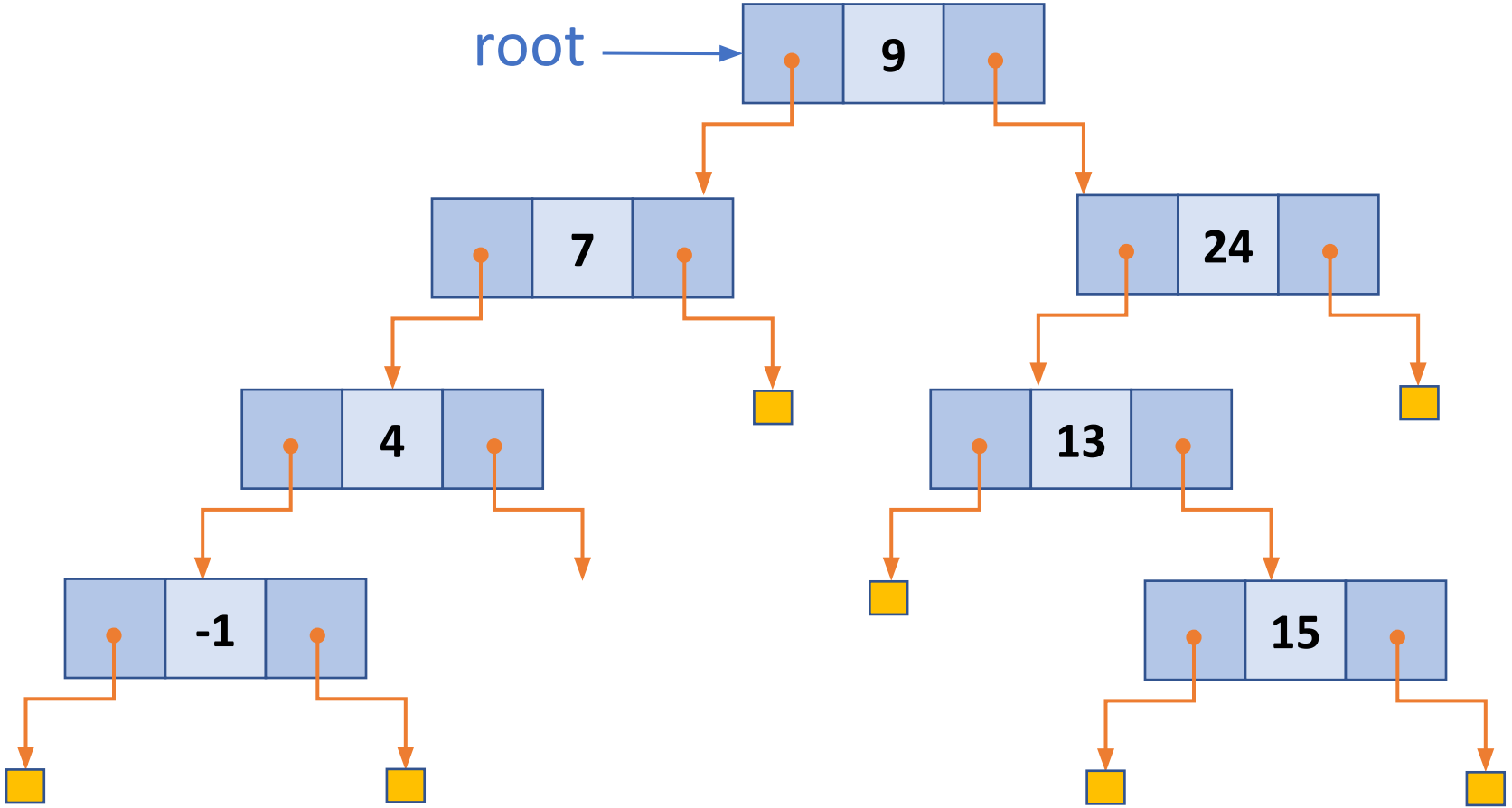
# Case 1: deleteNode(root, 5);



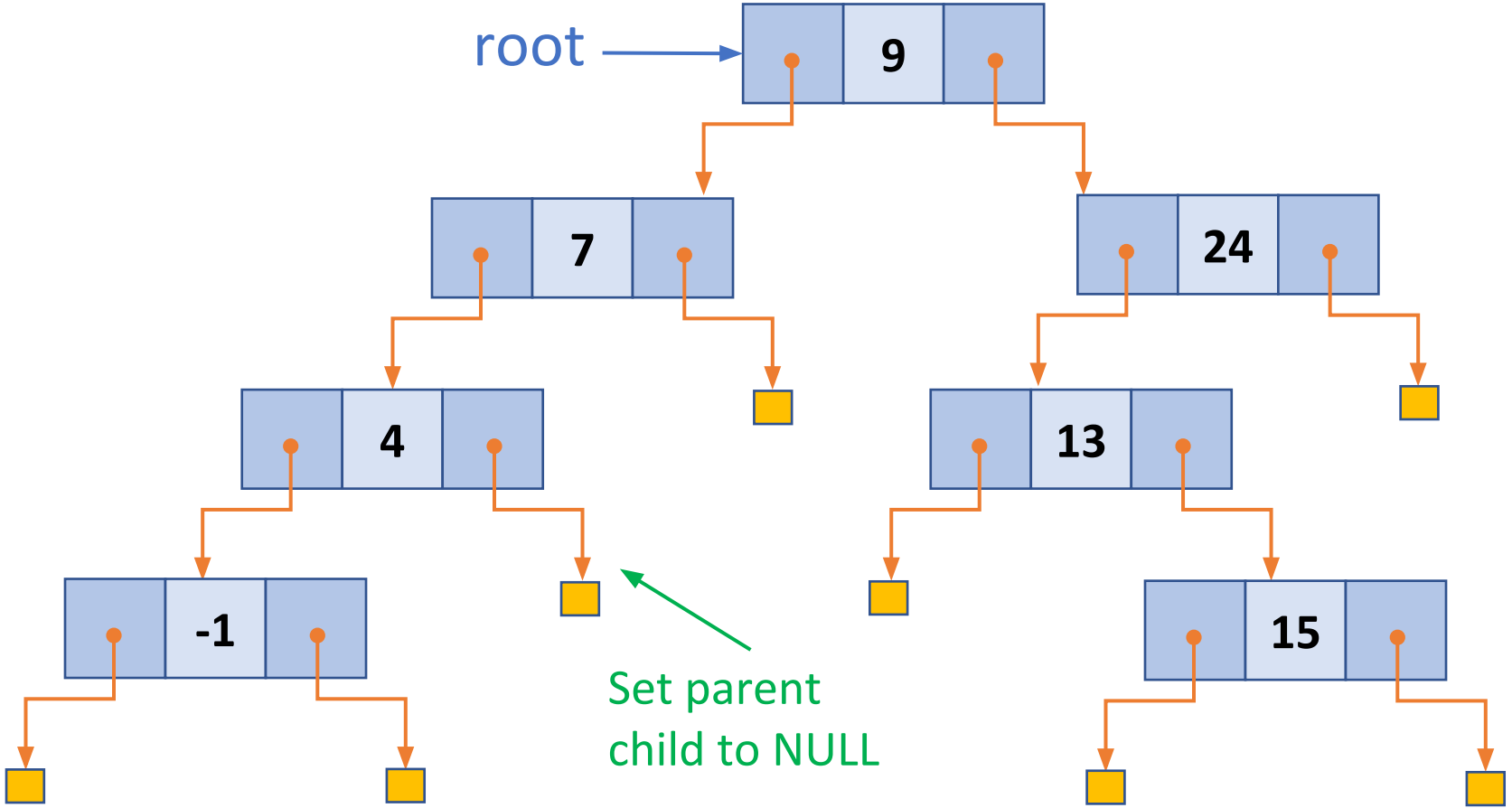
# Case 1: deleteNode(root, 5);



# Case 1: deleteNode(root, 5);



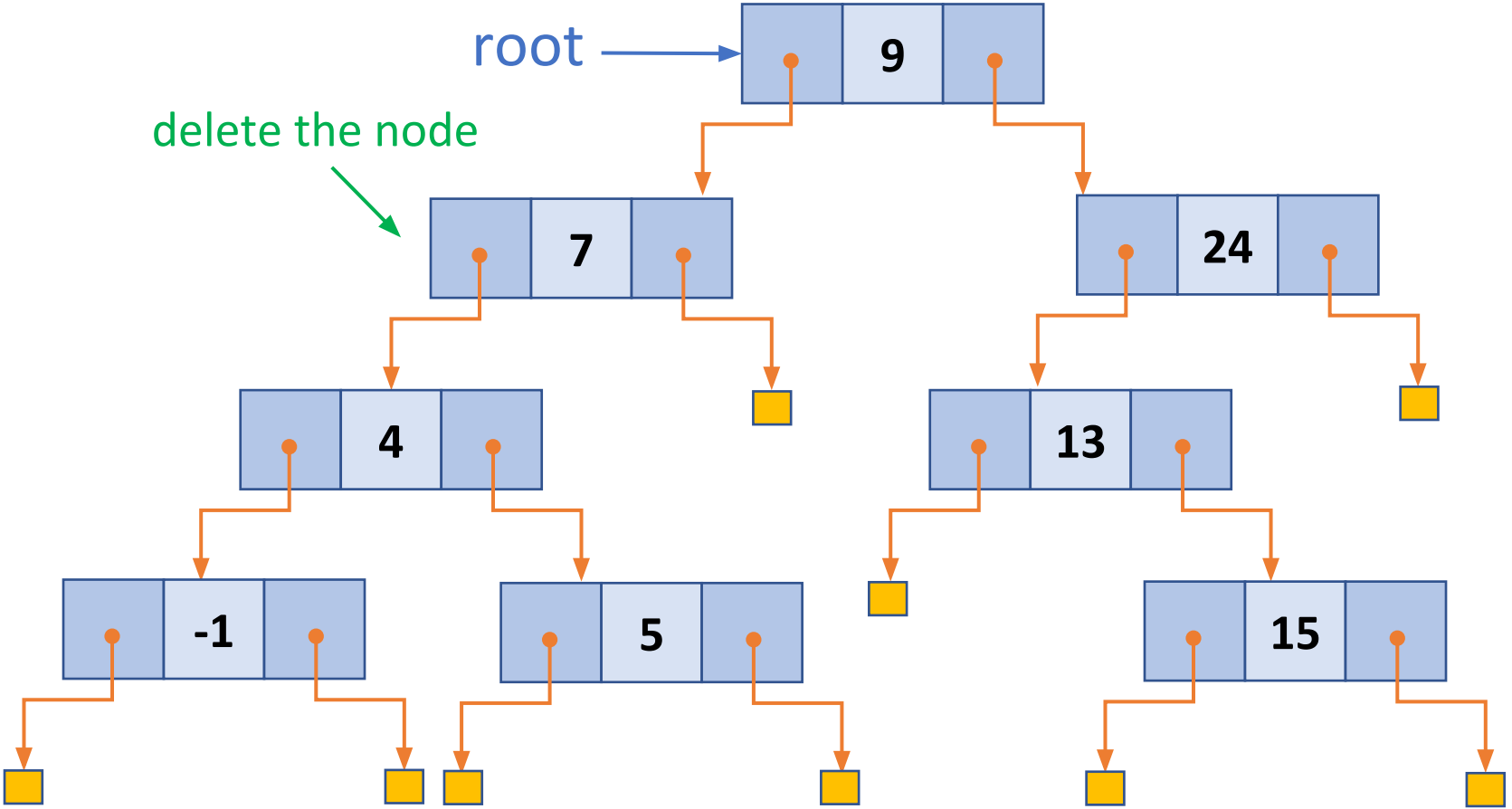
# Case 1: deleteNode(root, 5);



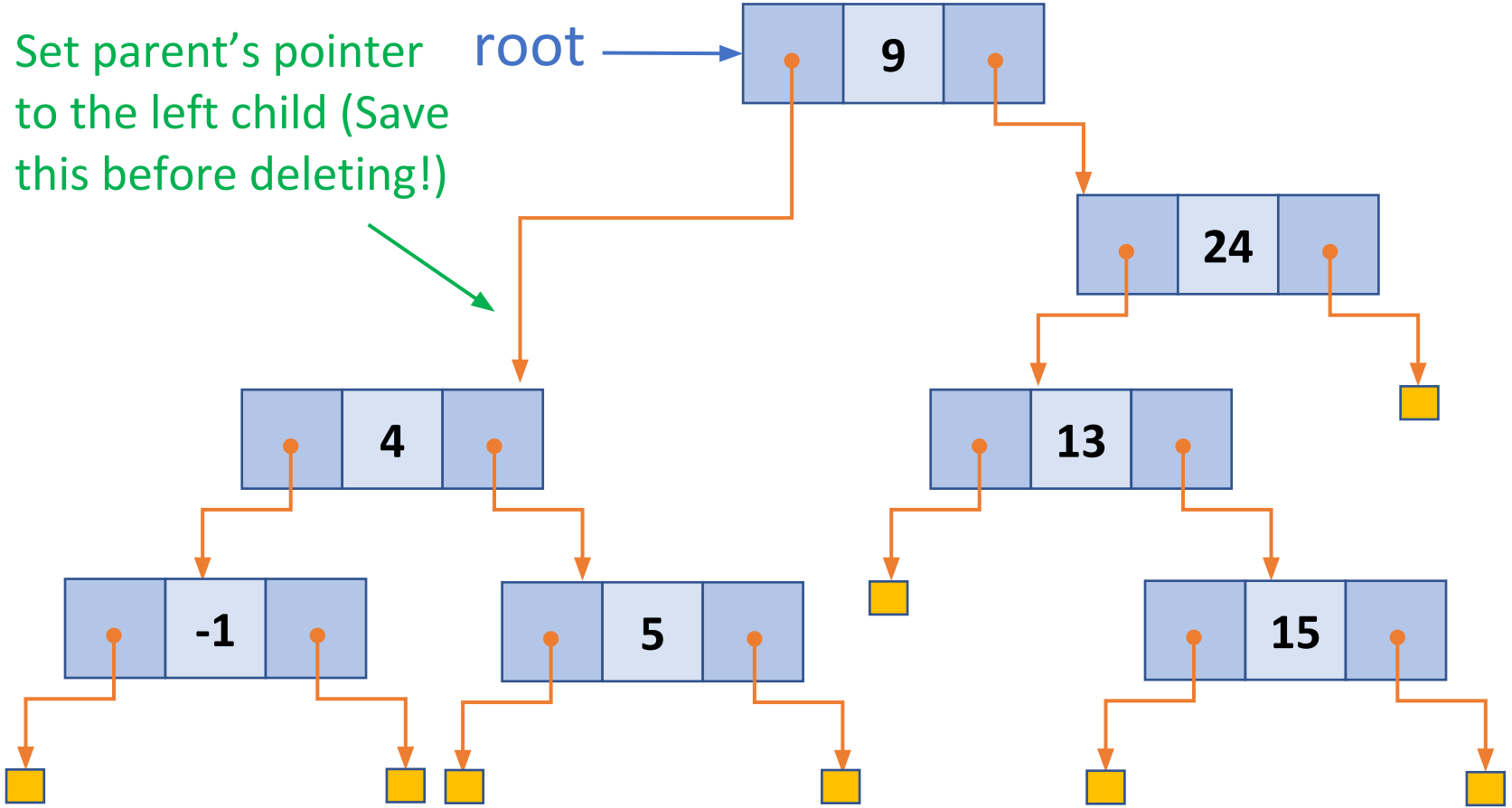
# Case 2: Deleting node with only left child

- Delete the node with the key.
- Set its parent’s child pointer to the node’s left child.

# Case 2: deleteNode(root, 7);



# Case 2: deleteNode(root, 7);



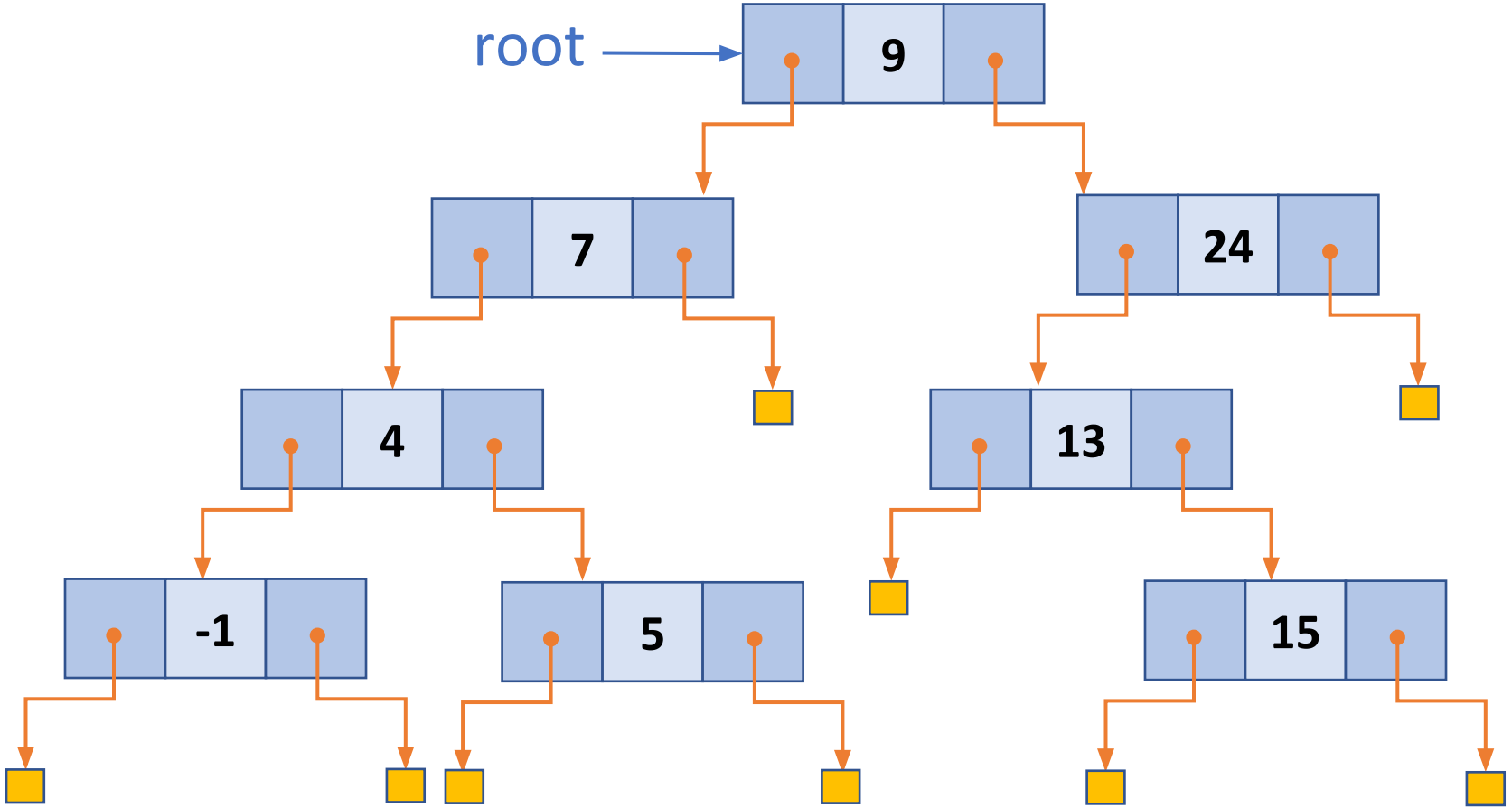
# Case 3: Deleting node with only right child

- Delete the node with the key.
- Set its parent’s child pointer to the node’s right child.

# Case 4: Deleting node with both children

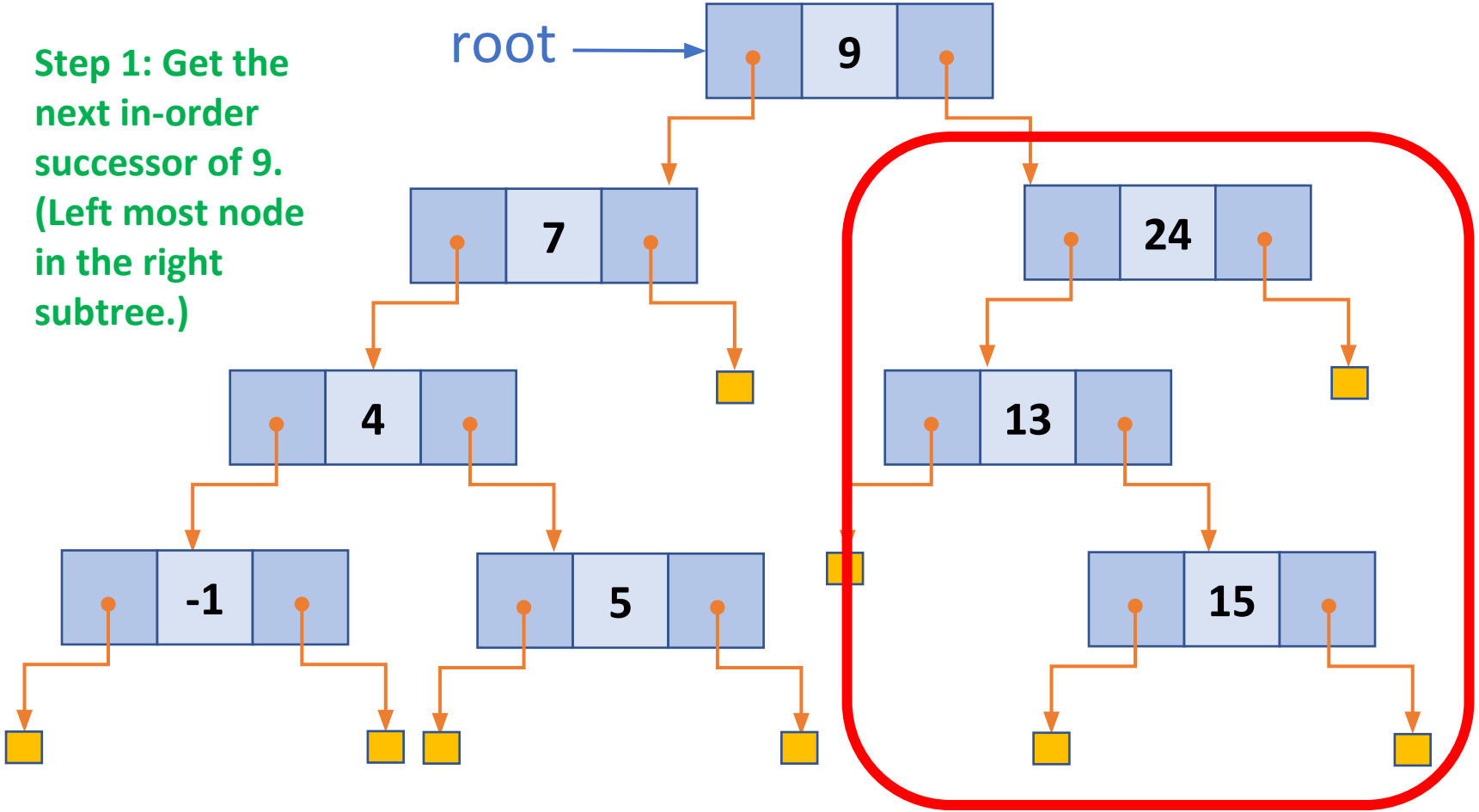
- Find the node which is the next in-order successor.
  - This is the leftmost child in the right subtree.
- Copy the value of the successor into the node.
- Delete the in-order successor.
  - This operation can be any of the first 3 cases.

# Case 4: `deleteNode(root, 9);`

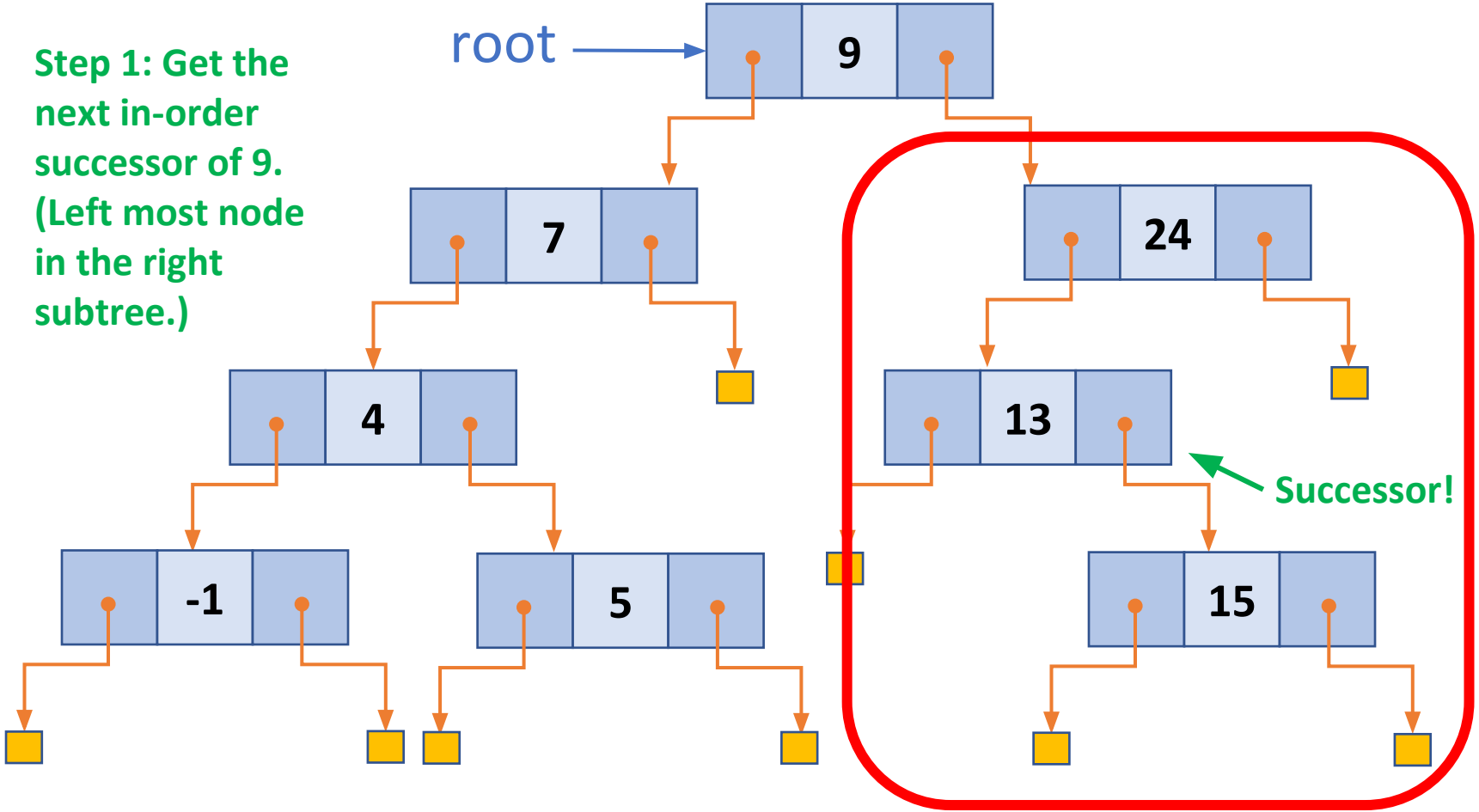


# Case 4: `deleteNode(root, 9);`

**Step 1: Get the next in-order successor of 9. (Left most node in the right subtree.)**

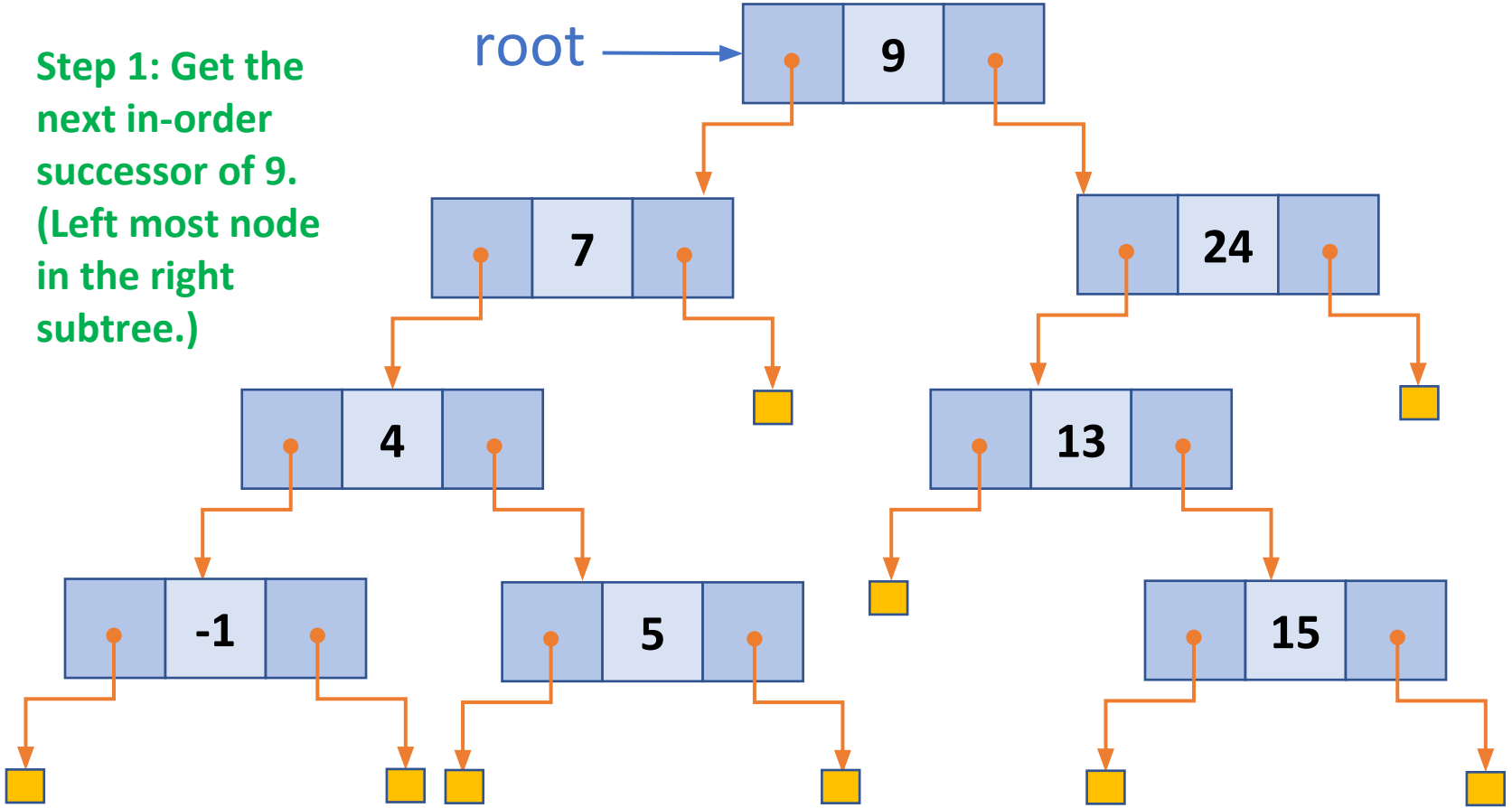


# Case 4: `deleteNode(root, 9);`

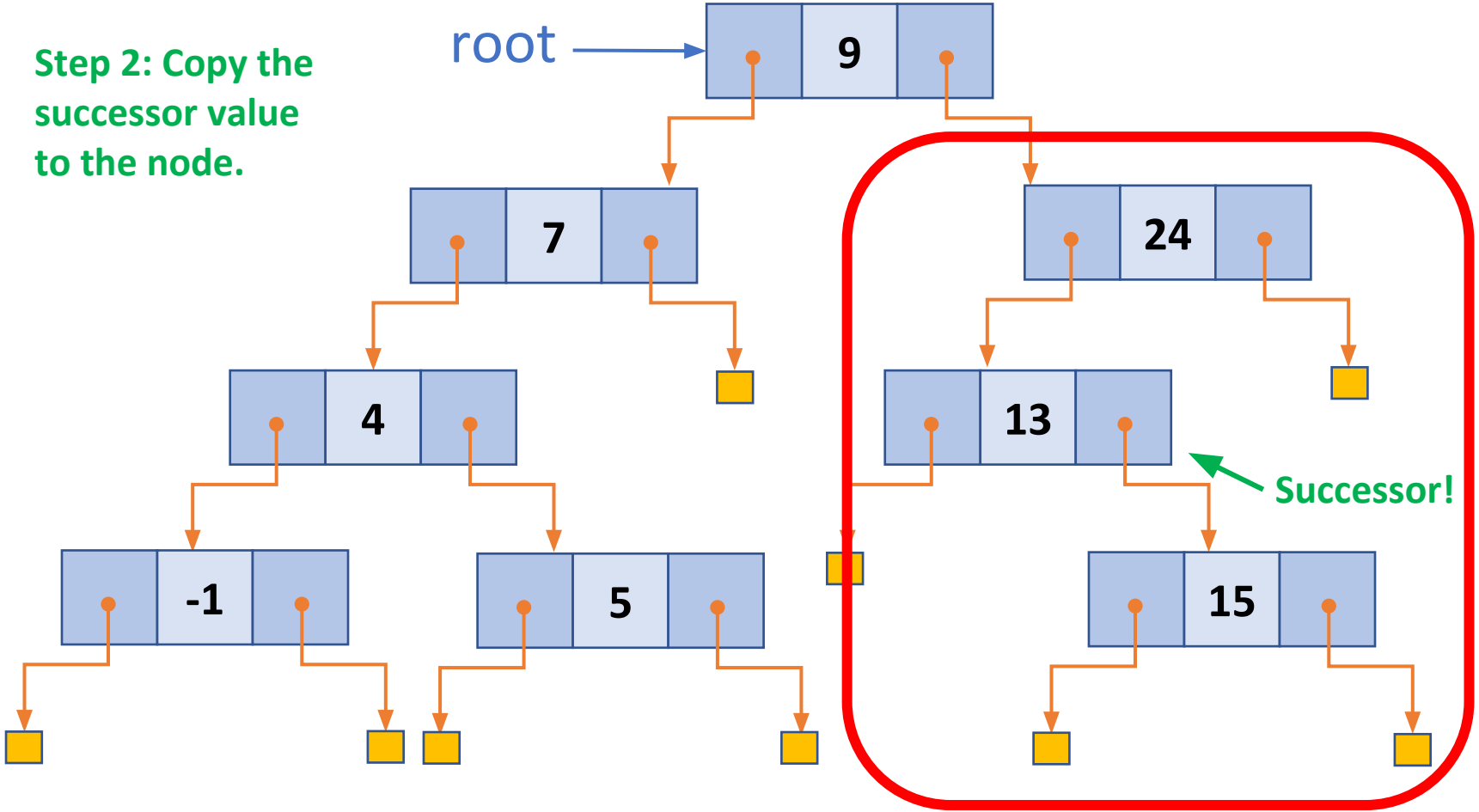


# Case 4: `deleteNode(root, 9);`

**Step 1: Get the next in-order successor of 9. (Left most node in the right subtree.)**

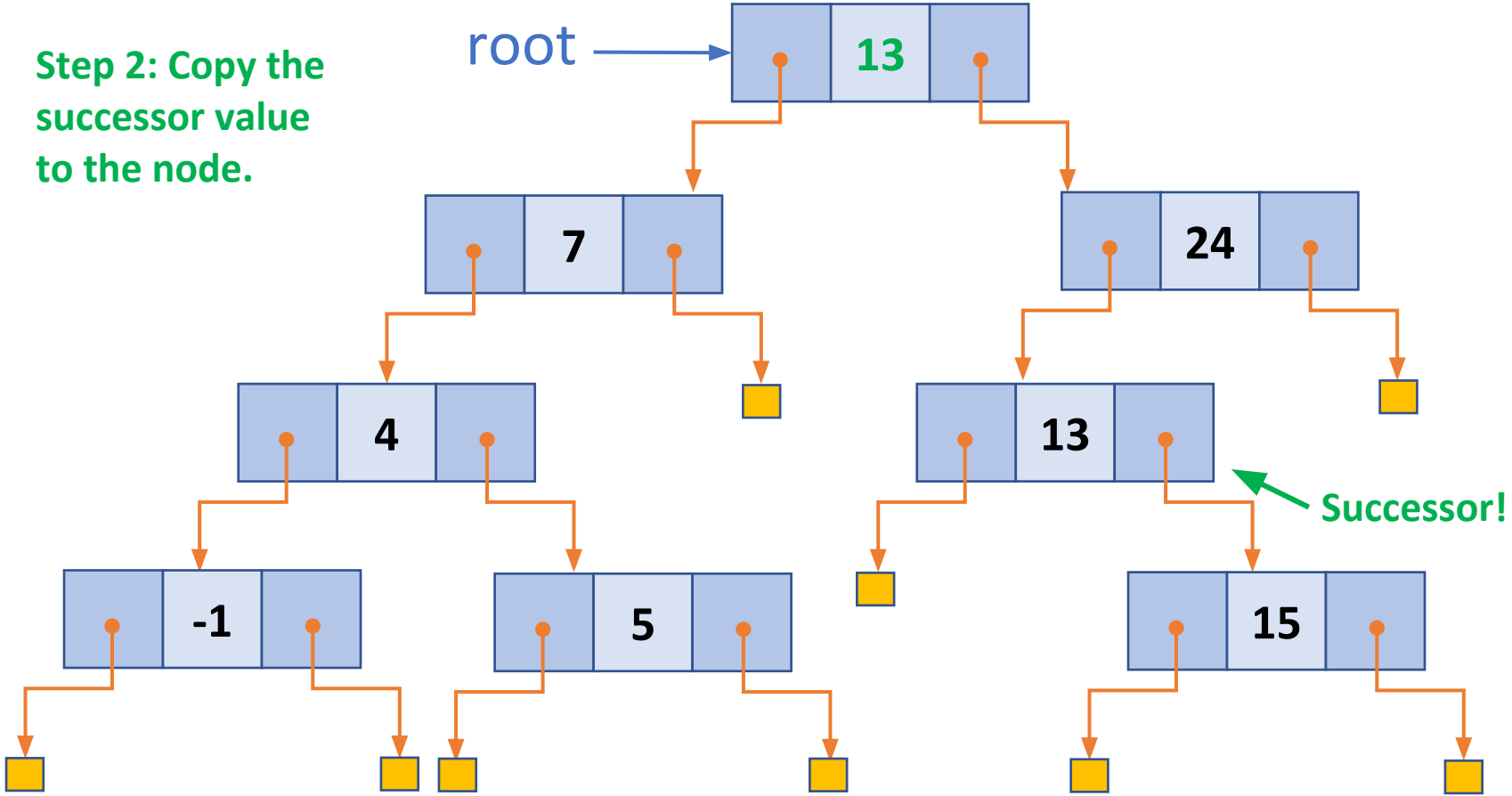


# Case 4: `deleteNode(root, 9);`



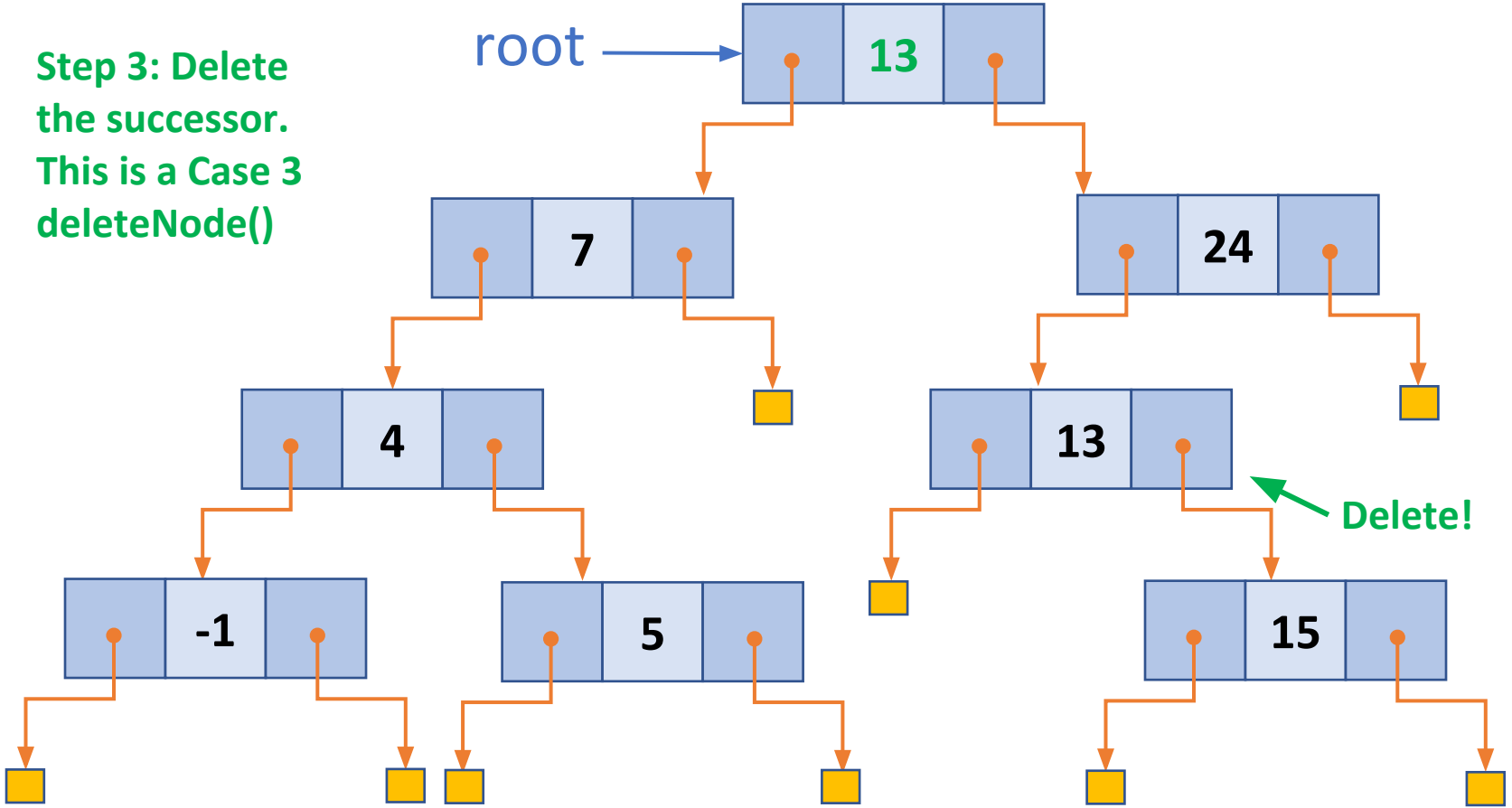
# Case 4: `deleteNode(root, 9);`

**Step 2: Copy the successor value to the node.**



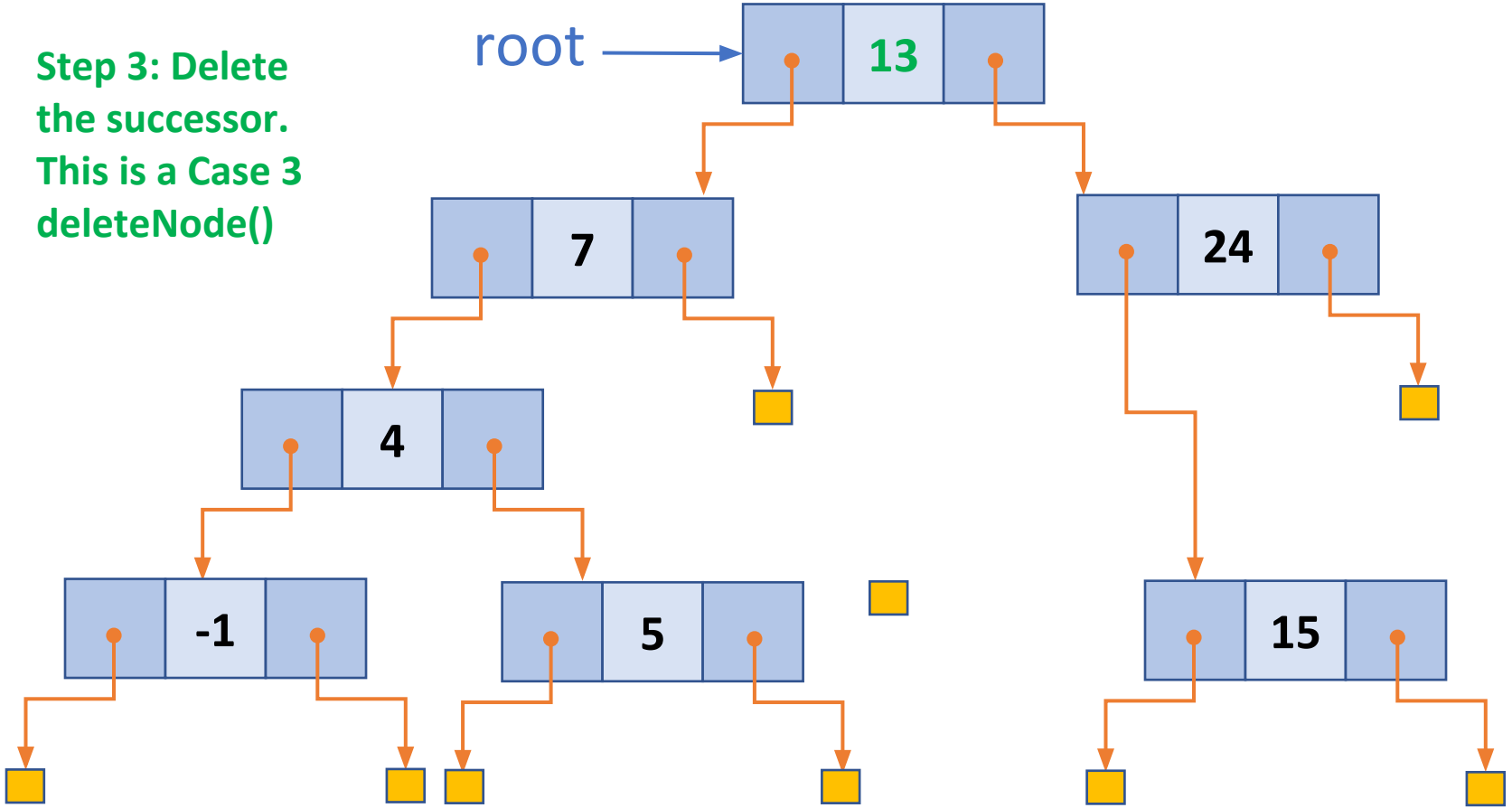
# Case 4: `deleteNode(root, 9);`

**Step 3: Delete the successor. This is a Case 3 deleteNode()**



# Case 4: `deleteNode(root, 9);`

**Step 3: Delete the successor. This is a Case 3 deleteNode()**



# Exercise (Silver Problem)

- This problem is mandatory.
- Definition of the Tree class and Node struct is present in tree.hpp
  - I highly encourage you to read it.
- You will be implementing the BST::deleteNode(Node *currNode, int value) in BST.cpp
- Compile both driver.cpp and tree.cpp together for successful compilation.
  - **g++ -std=c++11 driver.cpp BST.cpp -o rec8**
  - **./rec8**

# Recursive deleteNode (Silver Problem)

- Recursion: Tree Traversal until you reach the right node.

```
// SILVER TODO Complete the implementation of this function
Node* BST::deleteNode(Node *currNode, int value)
{

  if(currNode == NULL)
  {
    return NULL;
  }
  else if(value < currNode->key)
  {
    currNode->left = deleteNode(currNode->left, value);
  }
  else if(value > currNode->key)
  {
    currNode->right = deleteNode(currNode->right, value);
  }
  // We found the node with the value
  else
  {
```

# Recursive deleteNode (Silver Problem)

- Case 1: Delete the node and return __ ?
- The return value will set the parent's child pointer.

```
else
{
  //TODO Case : No child
  if(currNode->left == NULL && currNode->right == NULL)
  {

  }
```

# Recursive deleteNode (Silver Problem)

- Case 2,3: Delete the node and return __ ?
- The return value will set the parent's child pointer.

```
//TODO Case : Only right child
else if(currNode->left == NULL)
{

}
//TODO Case : Only left child
else if(currNode->right == NULL)
{

}
```

# Recursive deleteNode (Silver Problem)

- Case 4:

1. Find in-order successor
2. Copy its value to the node.
3. Recursively call deleteNode on the successor value.
4. Return the node pointer.

```
//TODO Case: Both left and right child
else
{
  ///Replace with Minimum from right subtree

}
```

# Gold Problem

- Check if a binary tree is a valid BST.
- Traverse the tree using InOrder Traversal and store the elements in an array.
- What should be special about this array if the tree is a BST?