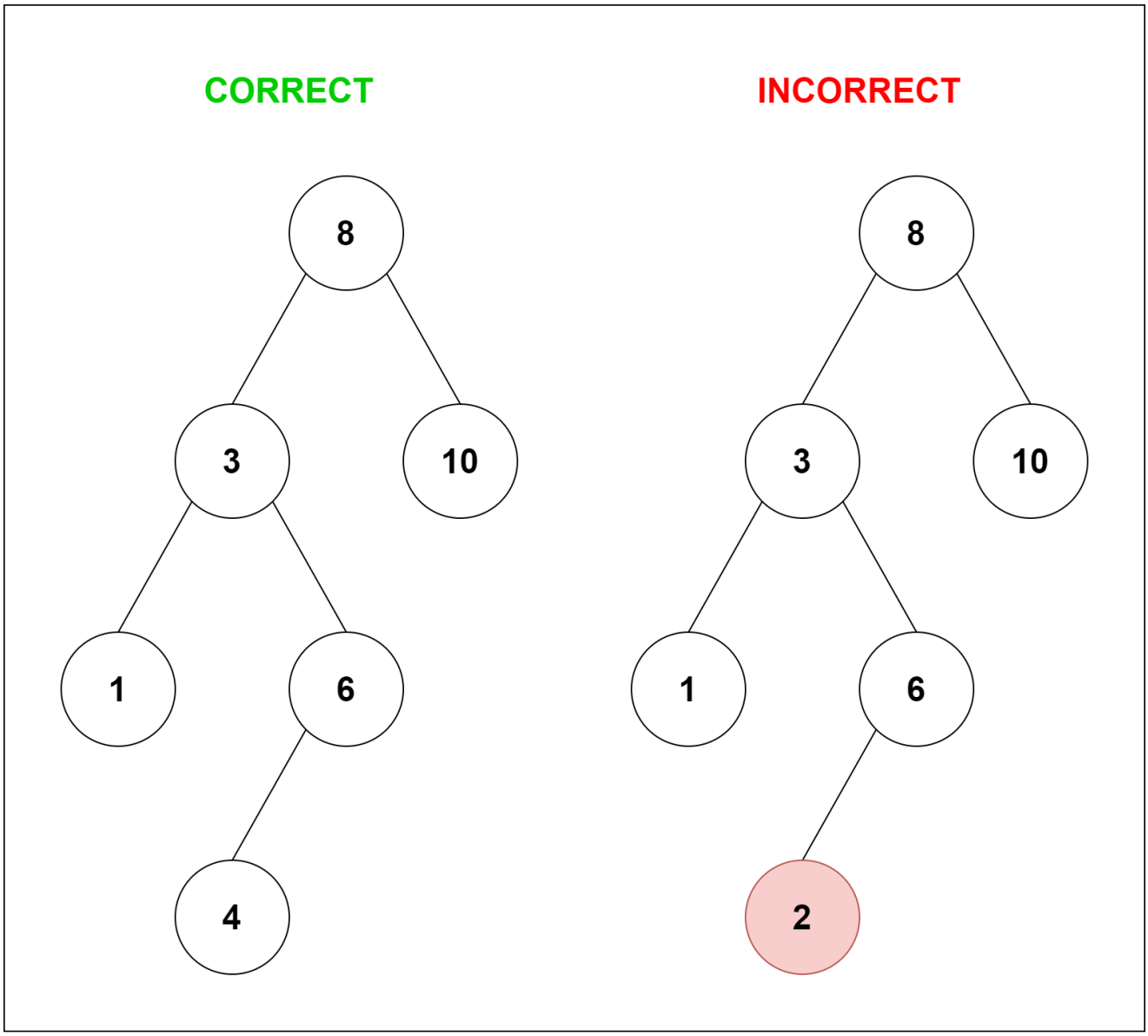
Binary Search Tree (BST)

Alright, let's delve into a fascinating data structure known as a Binary Search Tree or BST for short.

First, the name - binary because each node in this tree structure can have a maximum of two children. And, it's called a search tree because it's fantastic at helping us find things quickly, especially numbers. In fact, it can search for a number in a list of numbers incredibly fast, typically in O(log(n)) time, which is quite efficient.

Now, what makes a BST special compared to a regular binary tree? Well, it has some rules:

1. All the nodes in the left subtree of a node must have values less than that node.
2. All the nodes in the right subtree of a node must have values greater than that node.
3. Both the left and right subtrees of a node must also be BSTs, meaning they follow these same rules.



These rules create a structure where if you start at the root node and move left or right based on the value you're looking for, you can quickly zero in on your target.

But, be careful! If these rules are violated, like if you have a smaller value on the right side, it's not a proper BST.

Now, in terms of operations, you mainly do two things with a BST:

1. **Search:** You look for a specific value in the tree. It's like searching for a book in a library where all the books are neatly arranged.

2. **Insert:** You add a new value to the tree while maintaining these BST rules. Think of it like adding a new book to a library and ensuring it's placed in the right order on the shelf.

So, that's the Binary Search Tree - a clever way to sort and search for data efficiently.

Search Operation

Alright, let's dive into the search operation in a Binary Search Tree. Remember, the cool thing about a BST is that it helps us search for a specific value super fast.

Here's how it works:

1. First, we check if the root of the tree is NULL. If it is, that means the tree is empty, and there's nothing to search. We return NULL in this case.
2. Next, we check if the number we're looking for is equal to the value stored in the current root node. If it is, we found our number, and we return it.
3. Now, if the number is less than the value stored in the current root node, we know it can't be on the right side of the tree because, in a BST, all the values on the left are smaller than the root. So, we do a search on the left subtree by calling the search function recursively with the left child of the current root.
4. Conversely, if the number is greater than the value in the current root node, we do a search on the right subtree by calling the search function recursively with the right child

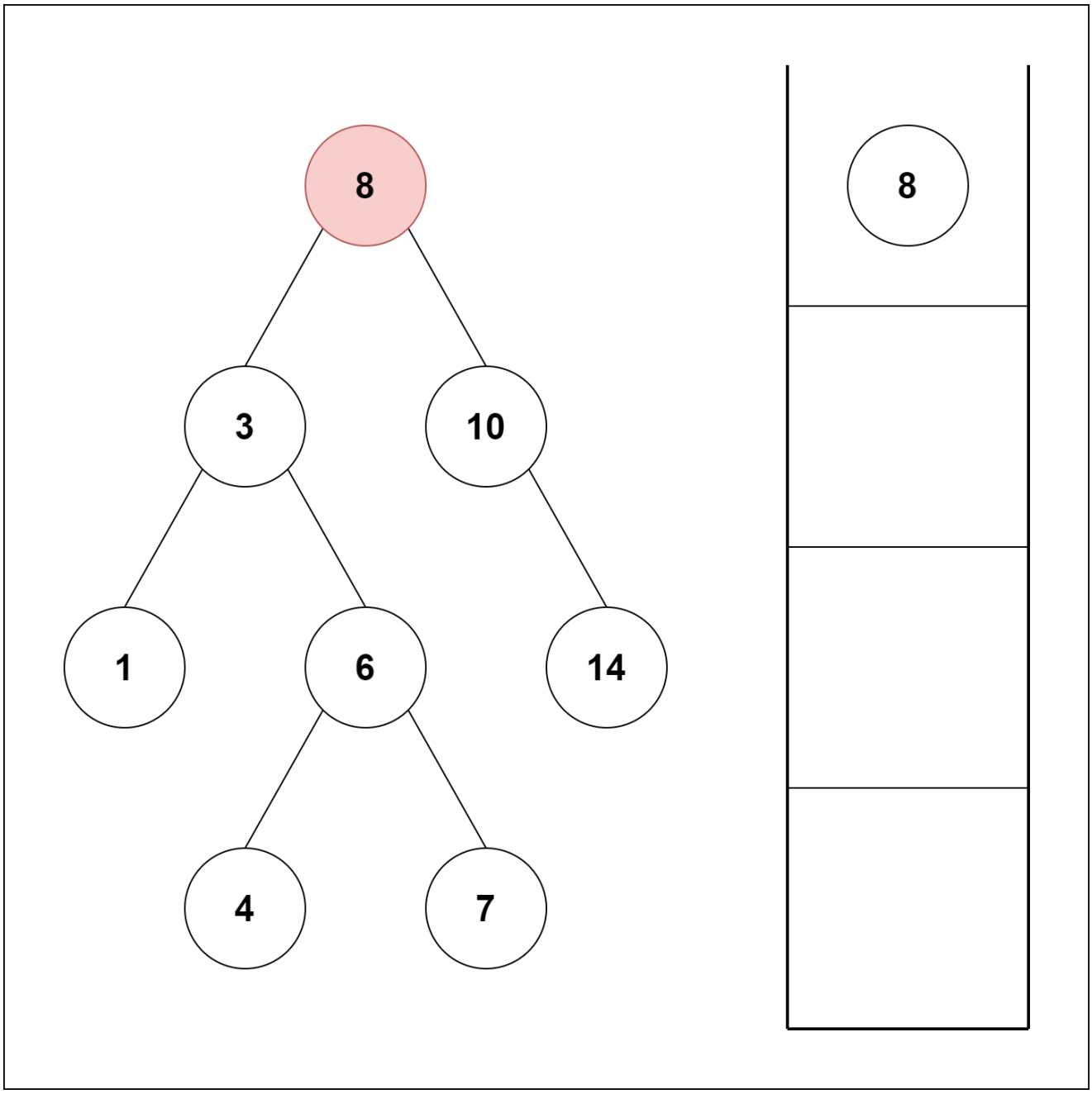
of the current root.

This process keeps going until one of two things happens:

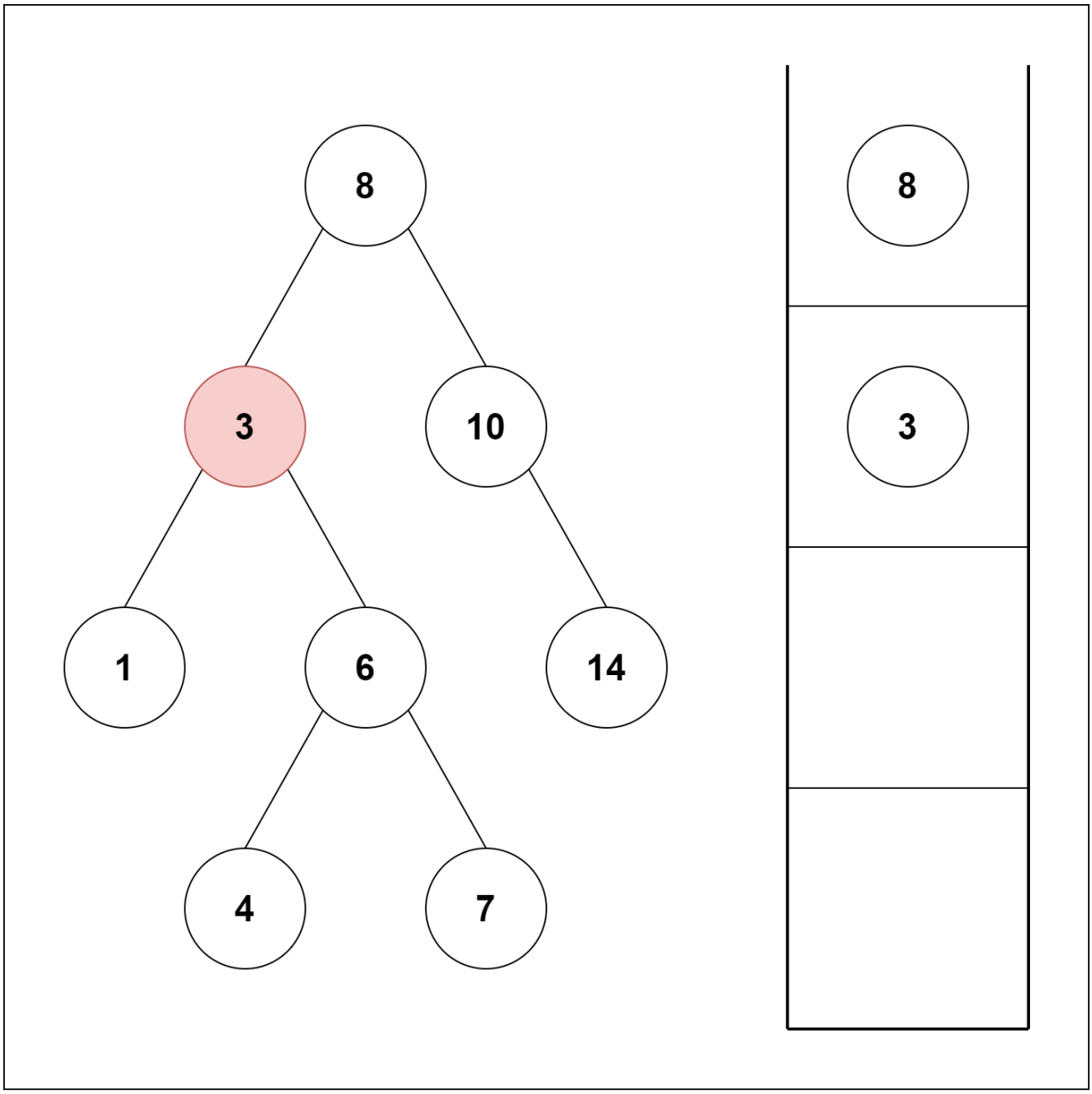
- We find the value we're looking for, in which case we return it.
- We reach a point where there's no more tree to search, like a leaf node with no children (this is when we hit NULL), and we return NULL.

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

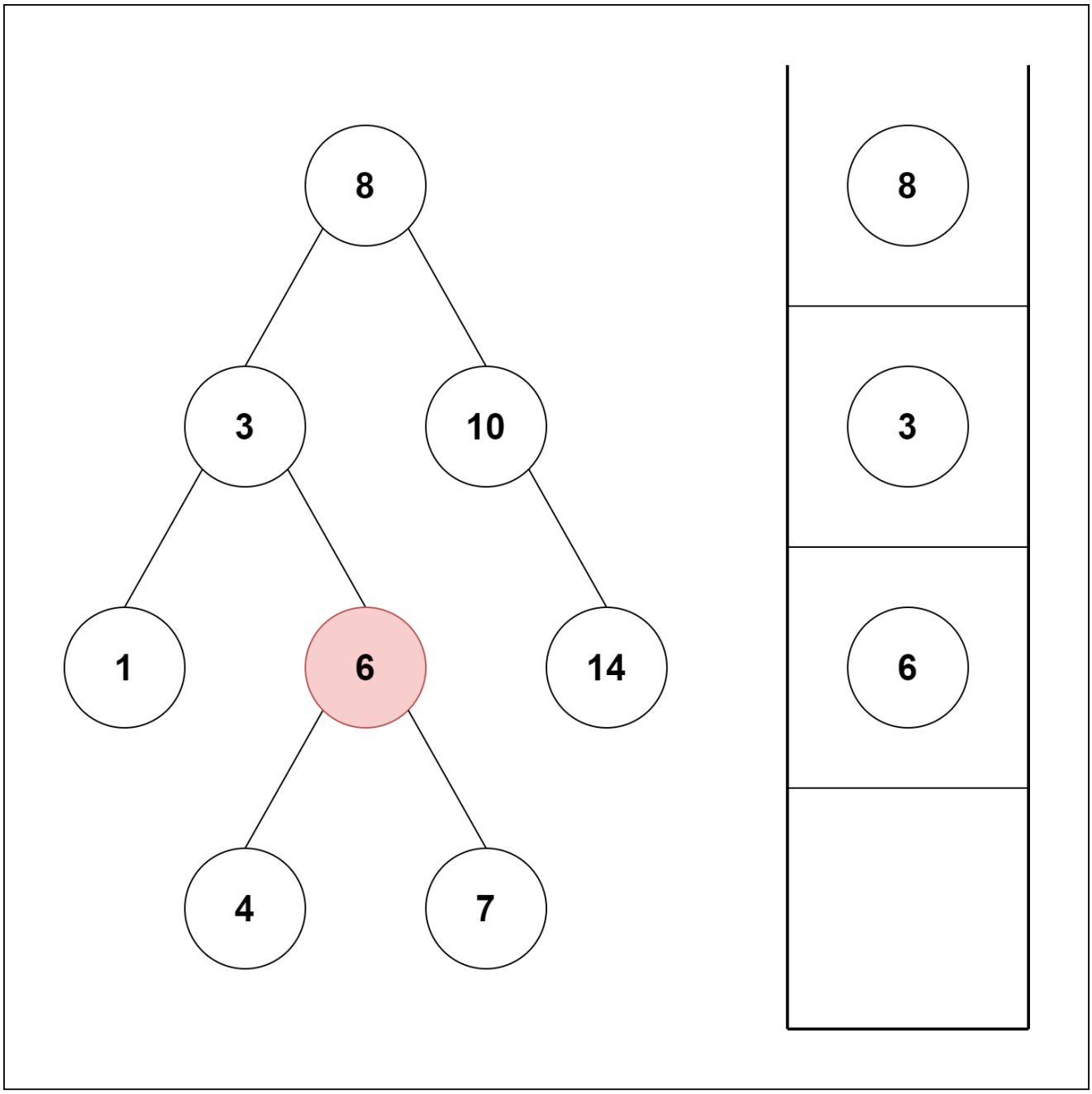
Let us try to visualize this with a diagram.



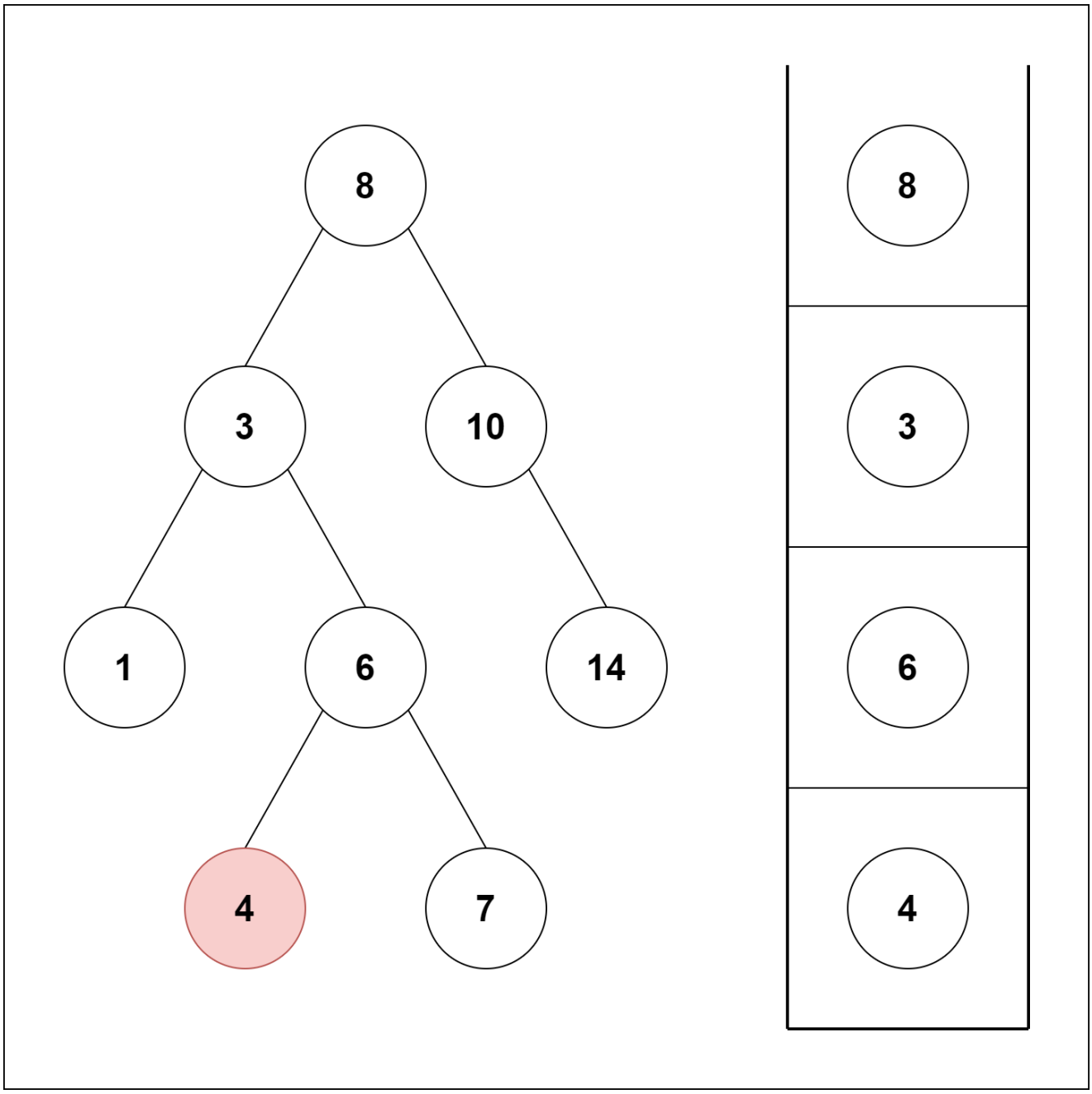
4 is not found so, traverse through the left subtree of 8.



4 is not found so, traverse through the right subtree of 3.



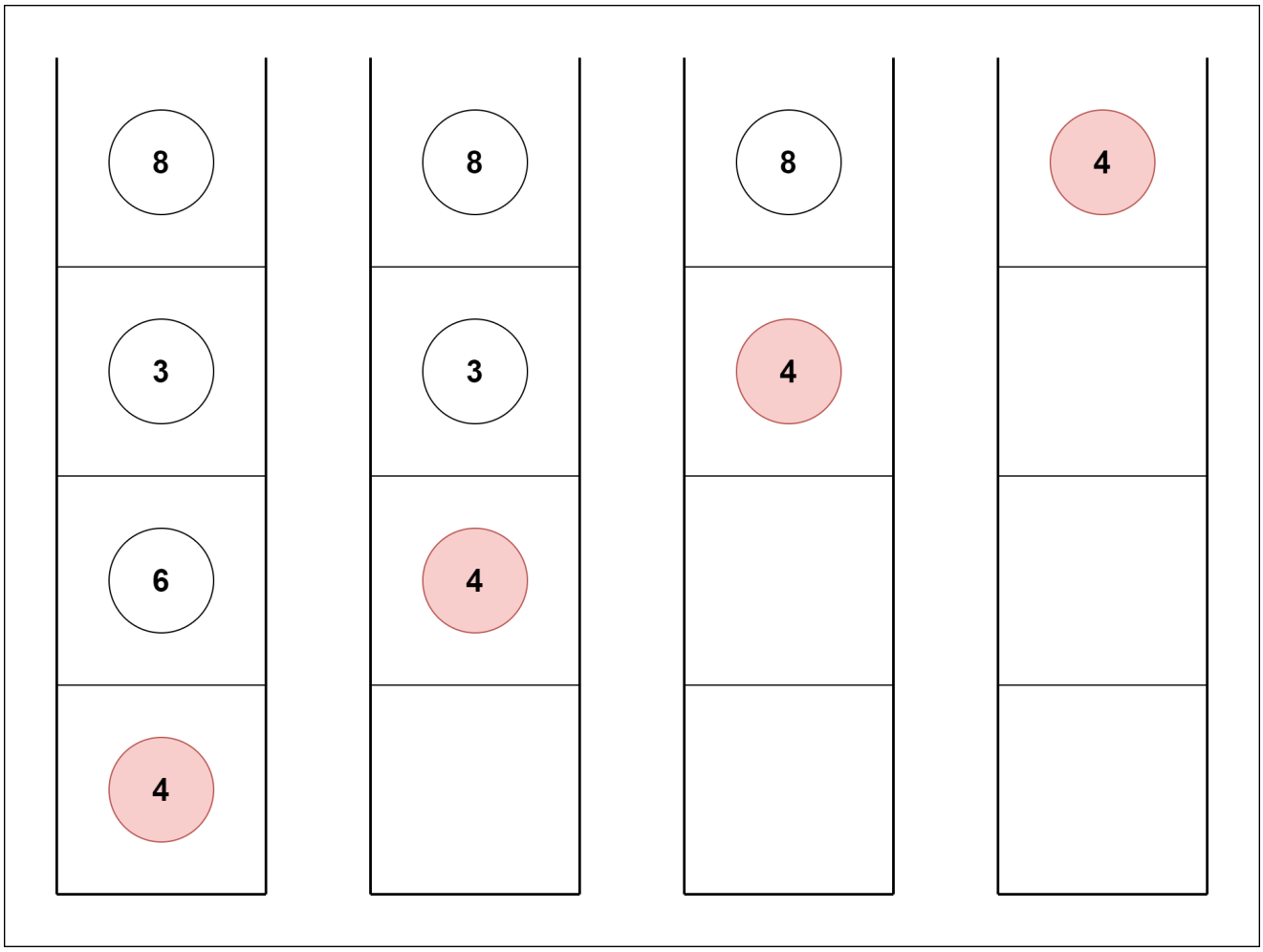
4 is not found so, traverse through the left subtree of 6.



4 is found.

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search(struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.



If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

So, essentially, we keep narrowing down our search by choosing the left or right subtree based on whether the number is smaller or greater than the current root value. This continues until we find the number or exhaust all possibilities in the tree.

This is the magic of how Binary Search Trees make searching for things so efficient!

Insert Operation

Now, let's talk about the insert operation in a Binary Search Tree. This is how we add a new value while keeping the tree in its sorted order.

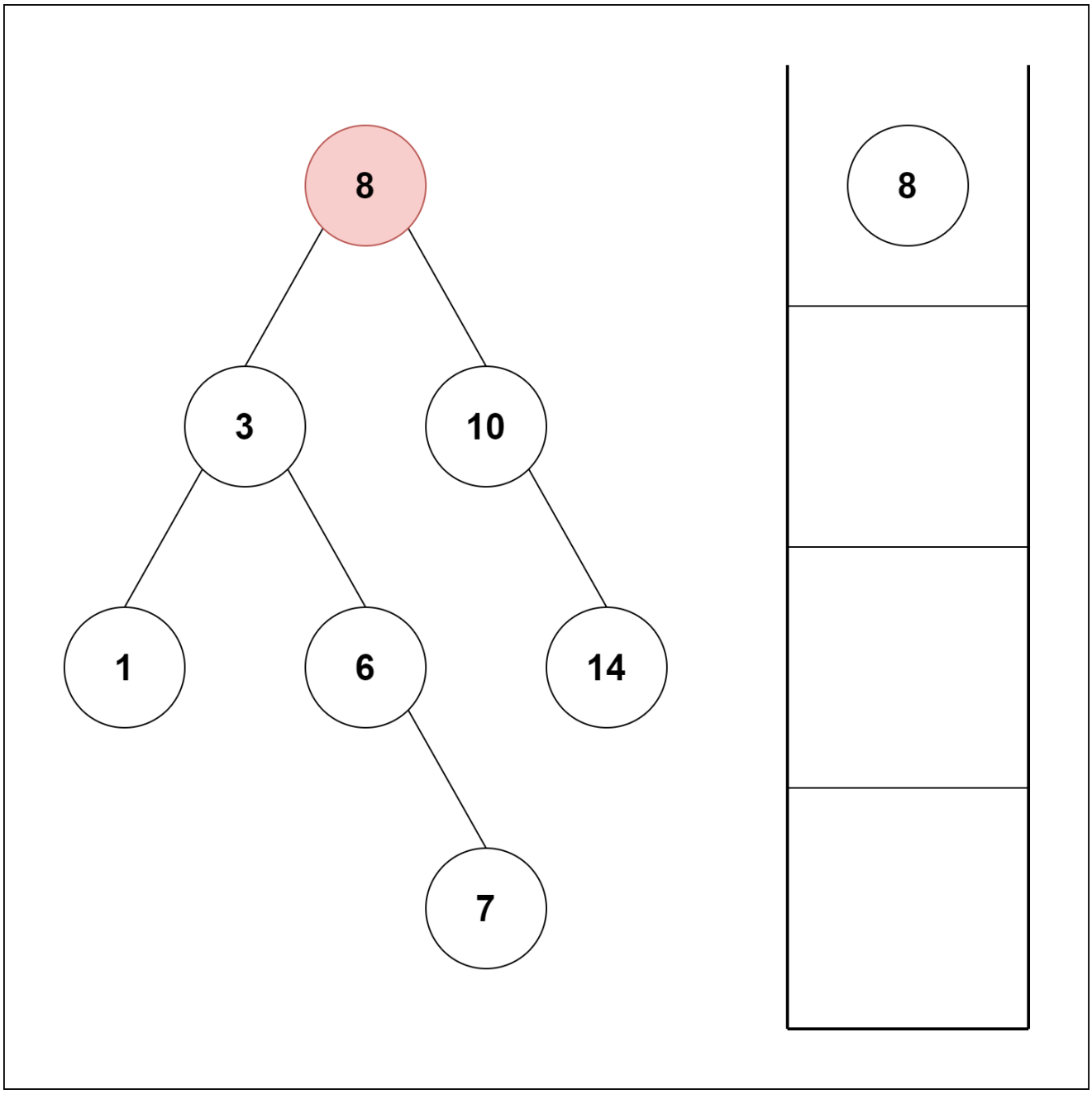
Here's how it works:

1. We start at the root of the tree and compare the value we want to insert with the value at the current node.
2. If the value to insert is smaller than the current node's value, we move to the left subtree because, in a BST, all the values on the left are smaller than the root.

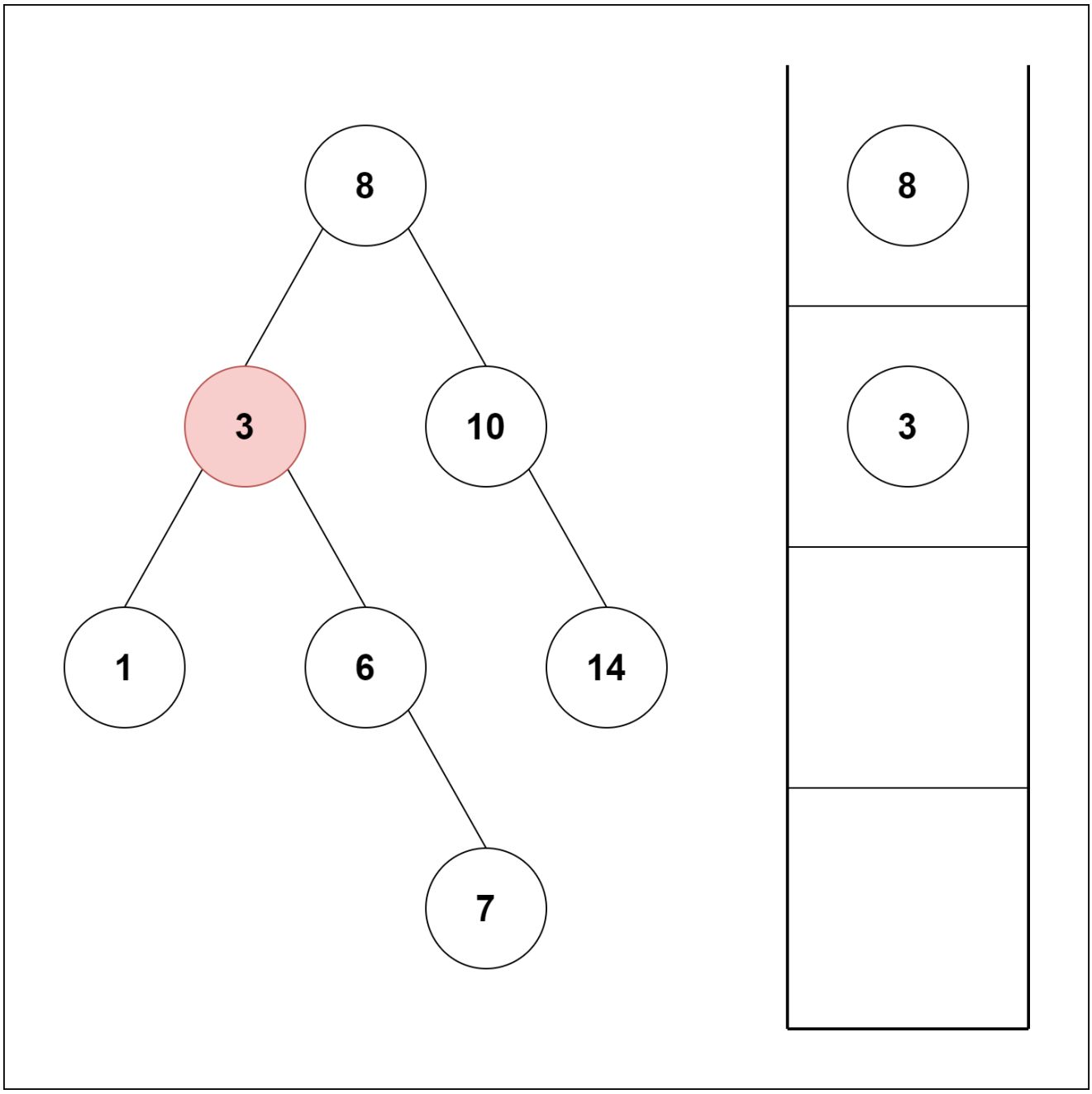
3. If the value to insert is larger than the current node's value, we move to the right subtree because, in a BST, all the values on the right are greater than the root.
4. We keep doing this until we reach a point where either the left or the right subtree is empty, which means we've found the spot to insert our new value.
5. We create a new node with the data we want to insert and attach it as the left or right child of the current node, depending on whether it's smaller or larger.
6. Finally, we return the current node. This might sound a bit strange, but it's crucial. As we've been moving down the tree and inserting nodes, we need to make sure that when we go back up the tree after the insertion, we don't mess up the rest of the tree. So, by returning the current node, we're essentially saying, "Hey, everything above me is fine; no changes needed."

```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

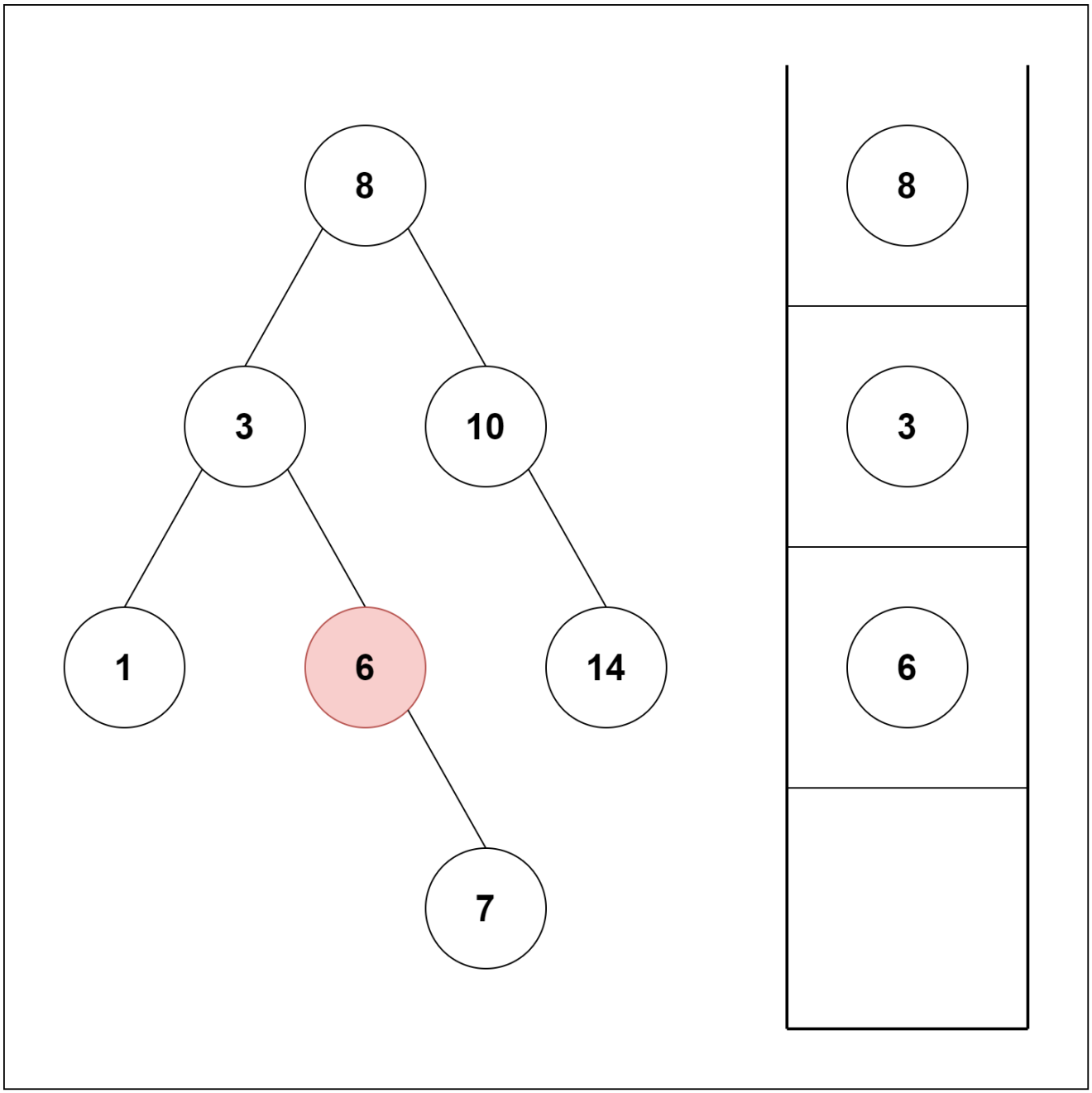
Let's try to visualize how we add a number to an existing BST.



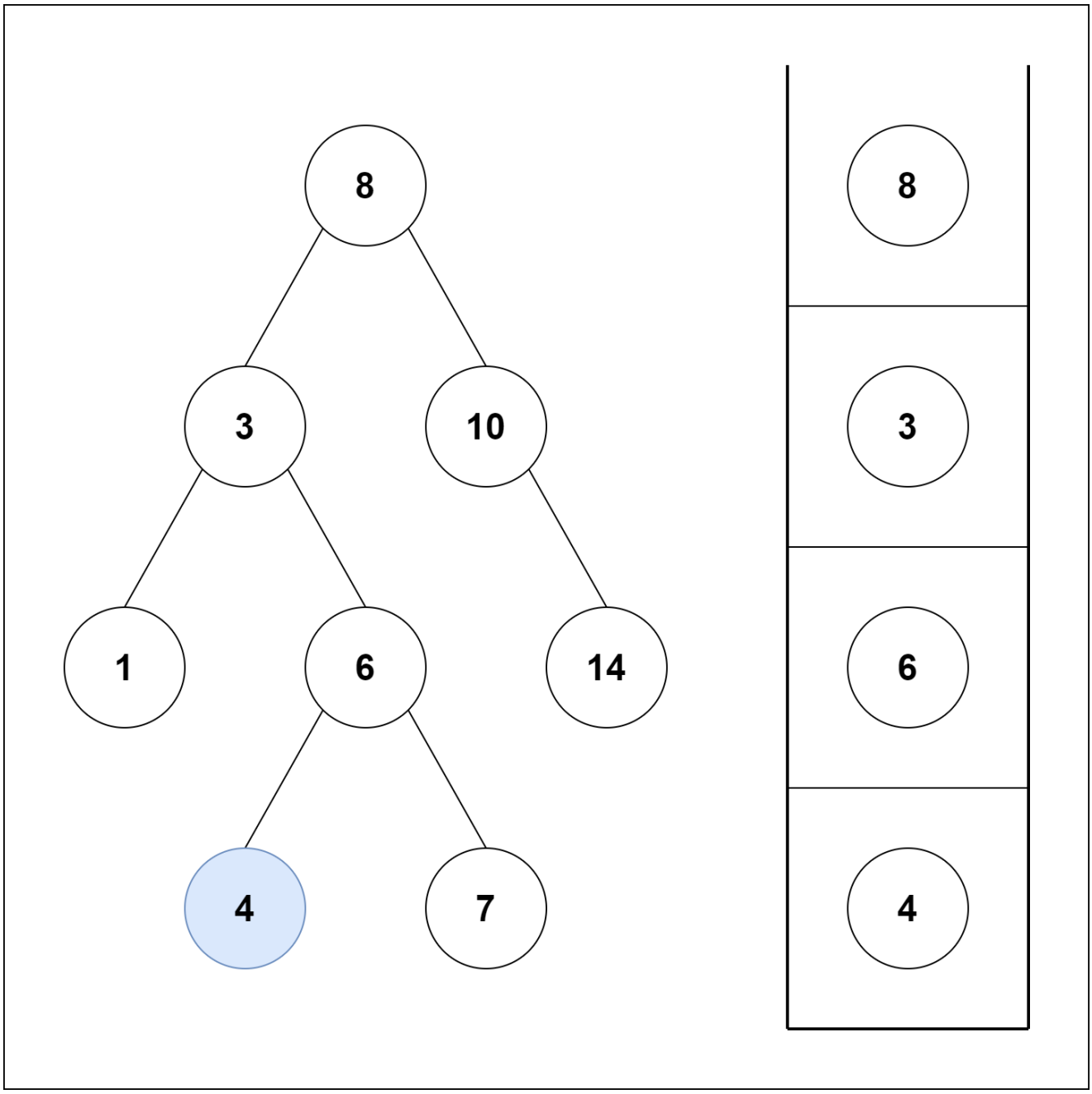
4<8 so, transverse through the left child of 8.



4>3 so, transverse through the right child of 8.



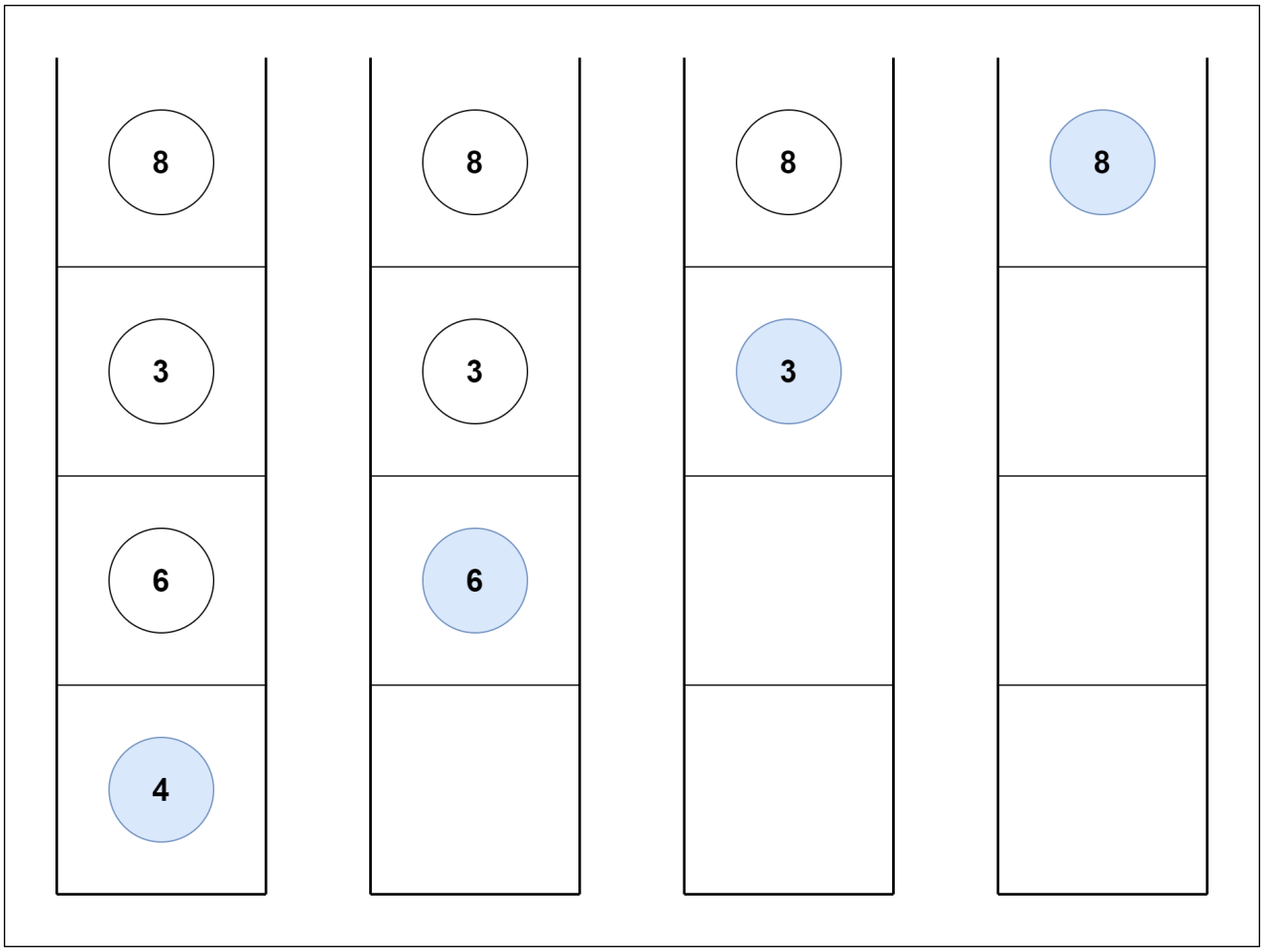
4<6 so, transverse through the left child of 6.



Insert 4 as a left child of 6.

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the `return node;` at the end comes in handy. In the case of `NULL`, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.



This process ensures that the BST property is maintained - the left subtree has values smaller than the root, and the right subtree has values larger than the root.

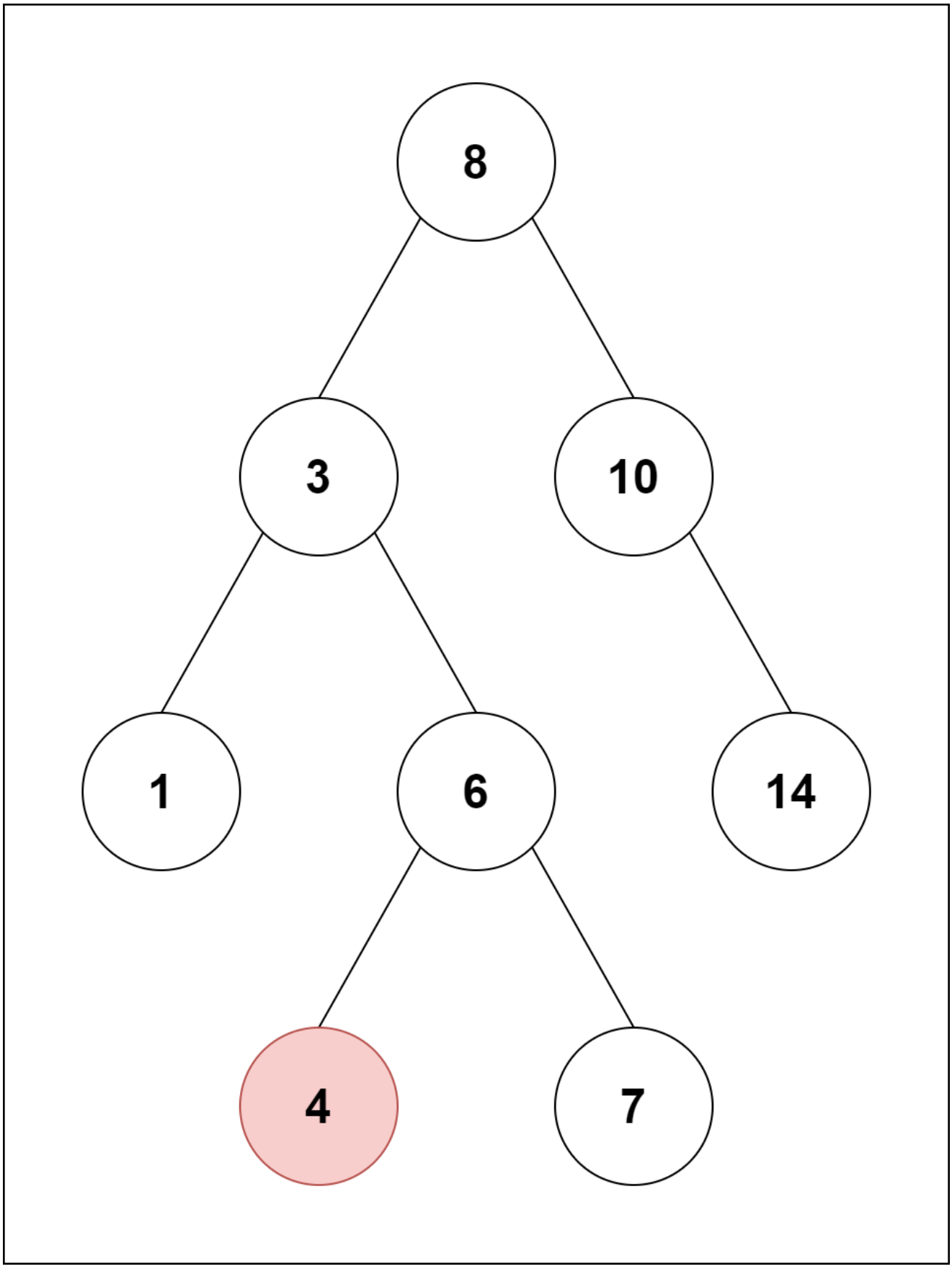
So, think of it like finding the perfect spot to insert a new member into a sorted list while keeping the list sorted. This is how Binary Search Trees stay organized as we add new values.

Deletion Operation

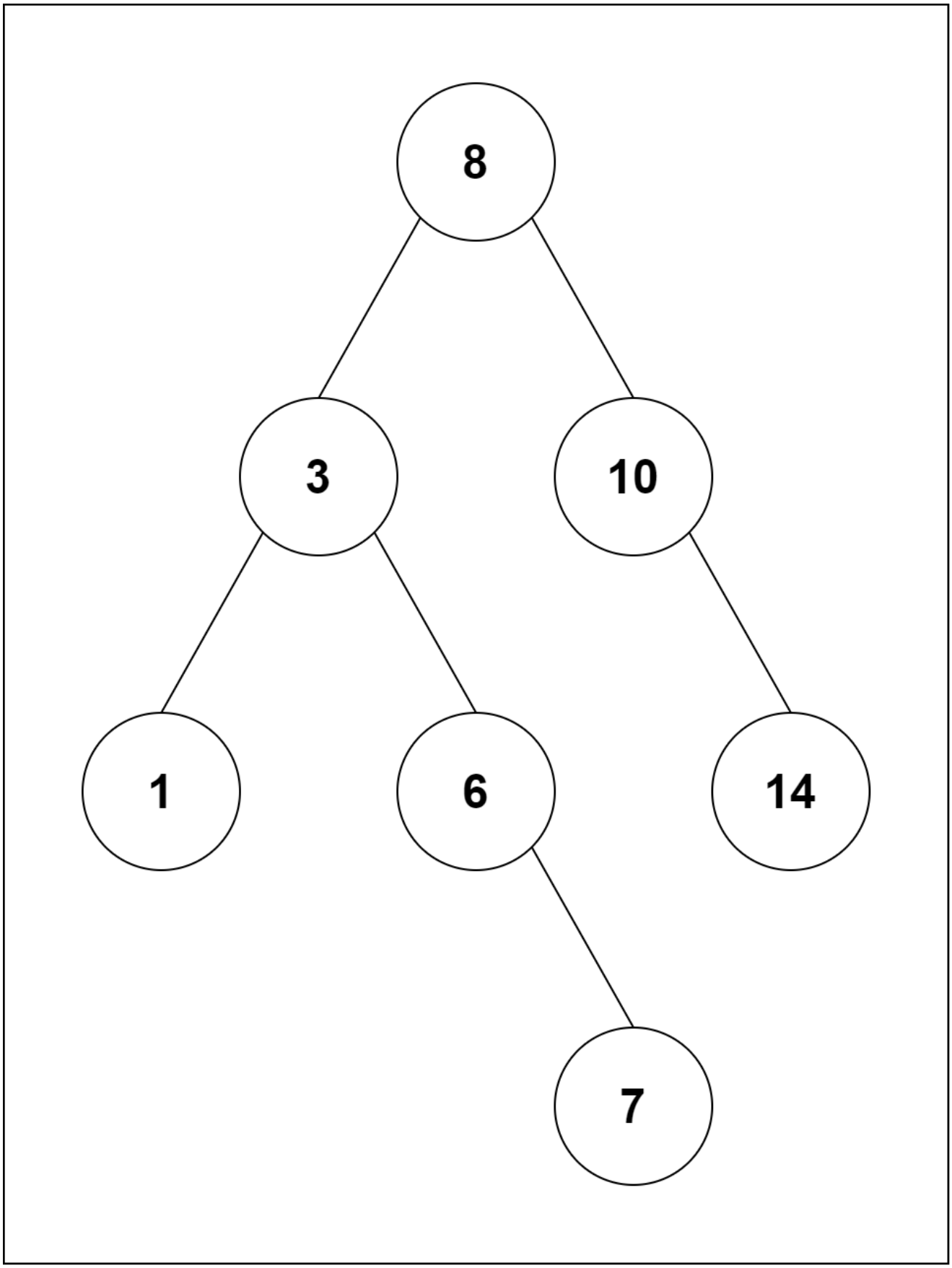
Let's discuss how we can delete a node from a Binary Search Tree (BST). There are three different scenarios we might encounter when trying to delete a node:

Case I:
In the first scenario, the node we want to delete is a leaf node. In simple terms, it doesn't have any children of its own. In this case, we can just go ahead and remove that node from the tree. It's like plucking a leaf off a tree.

4 is to be deleted.



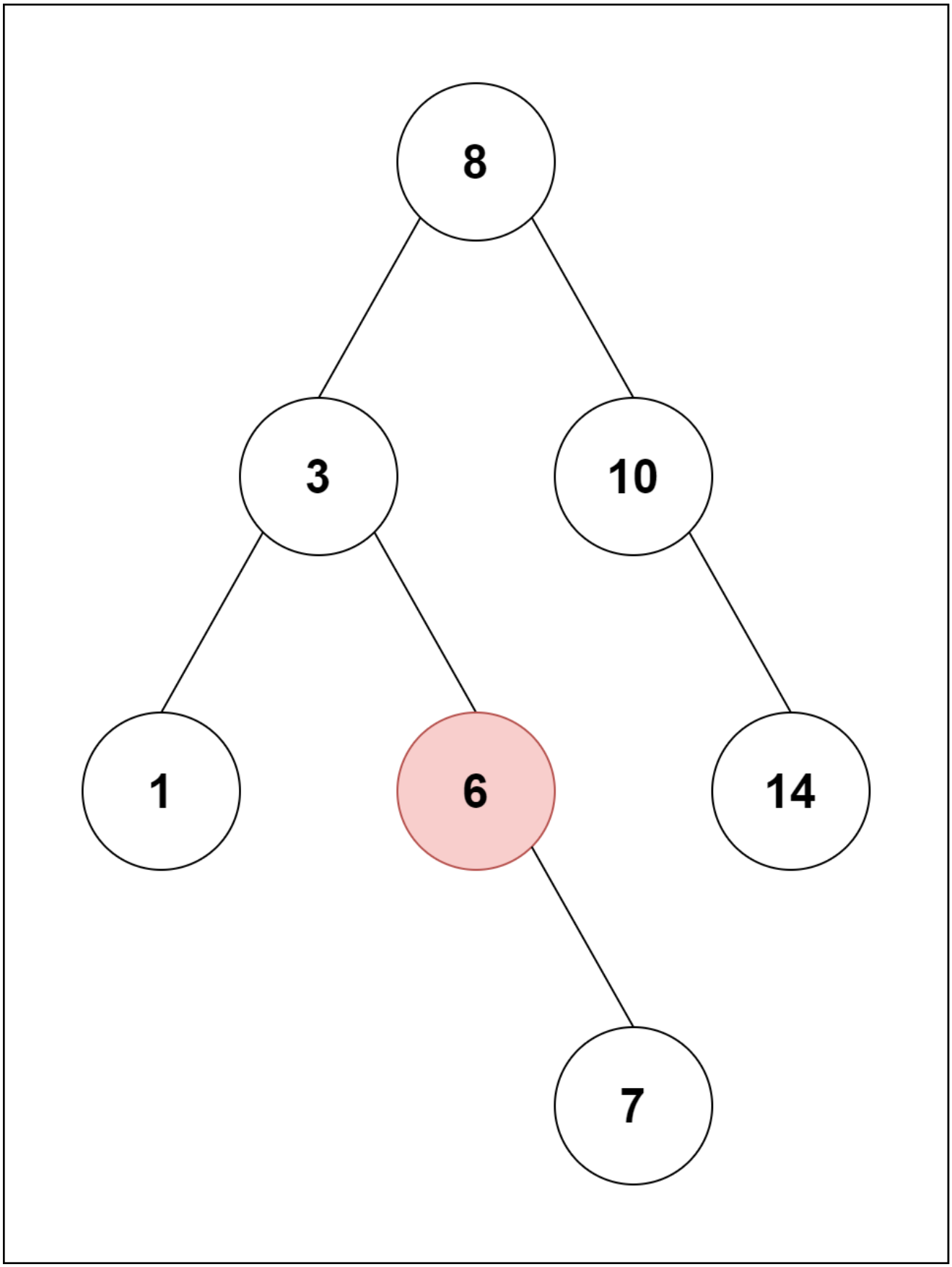
Delete the node.



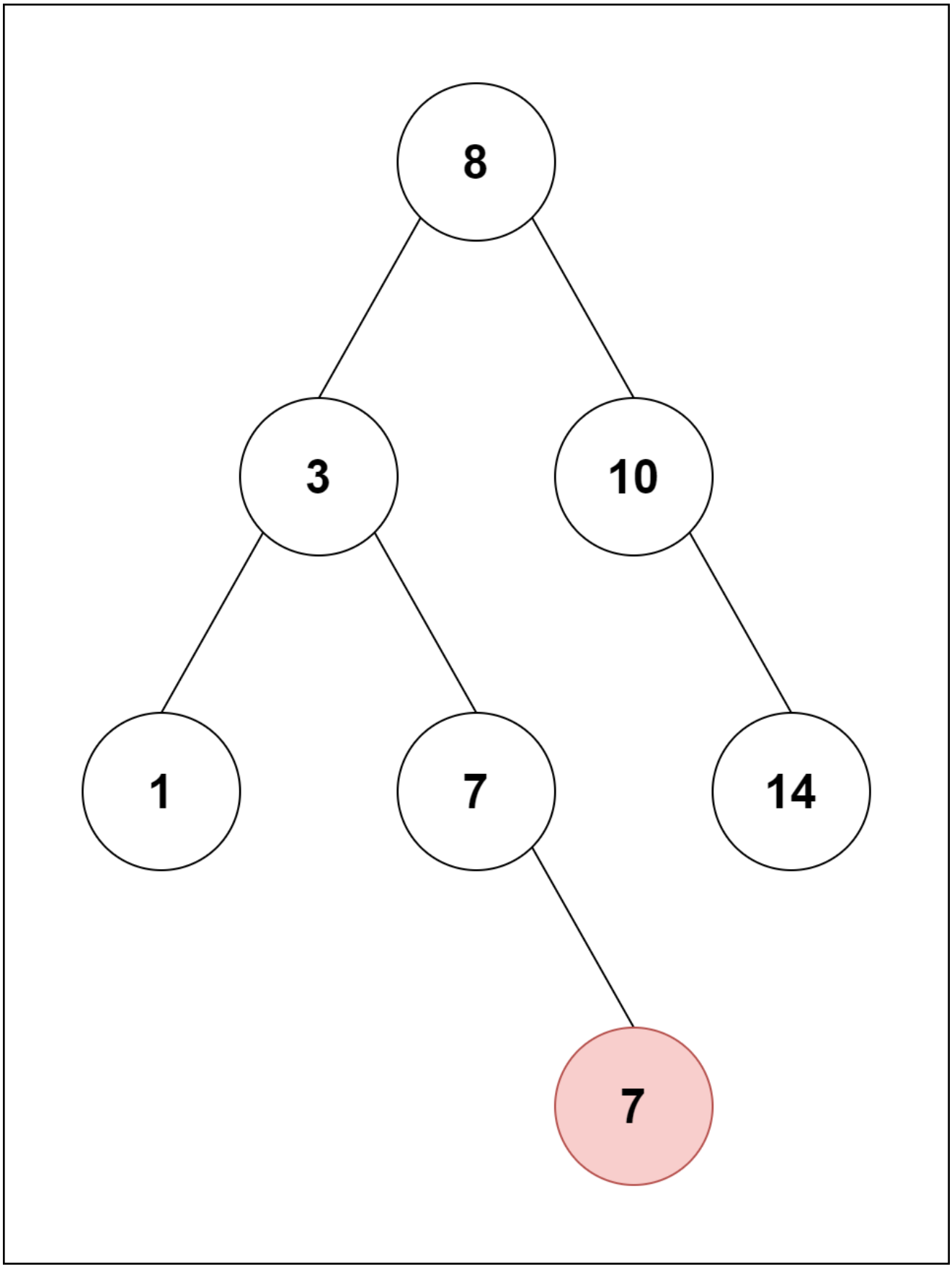
Case II:

In the second scenario, the node we want to delete has just one child node. Now, what we do here is replace the node we want to delete with its child node. It's similar to adopting the child. After that, we remove the child node from its original position in the tree.

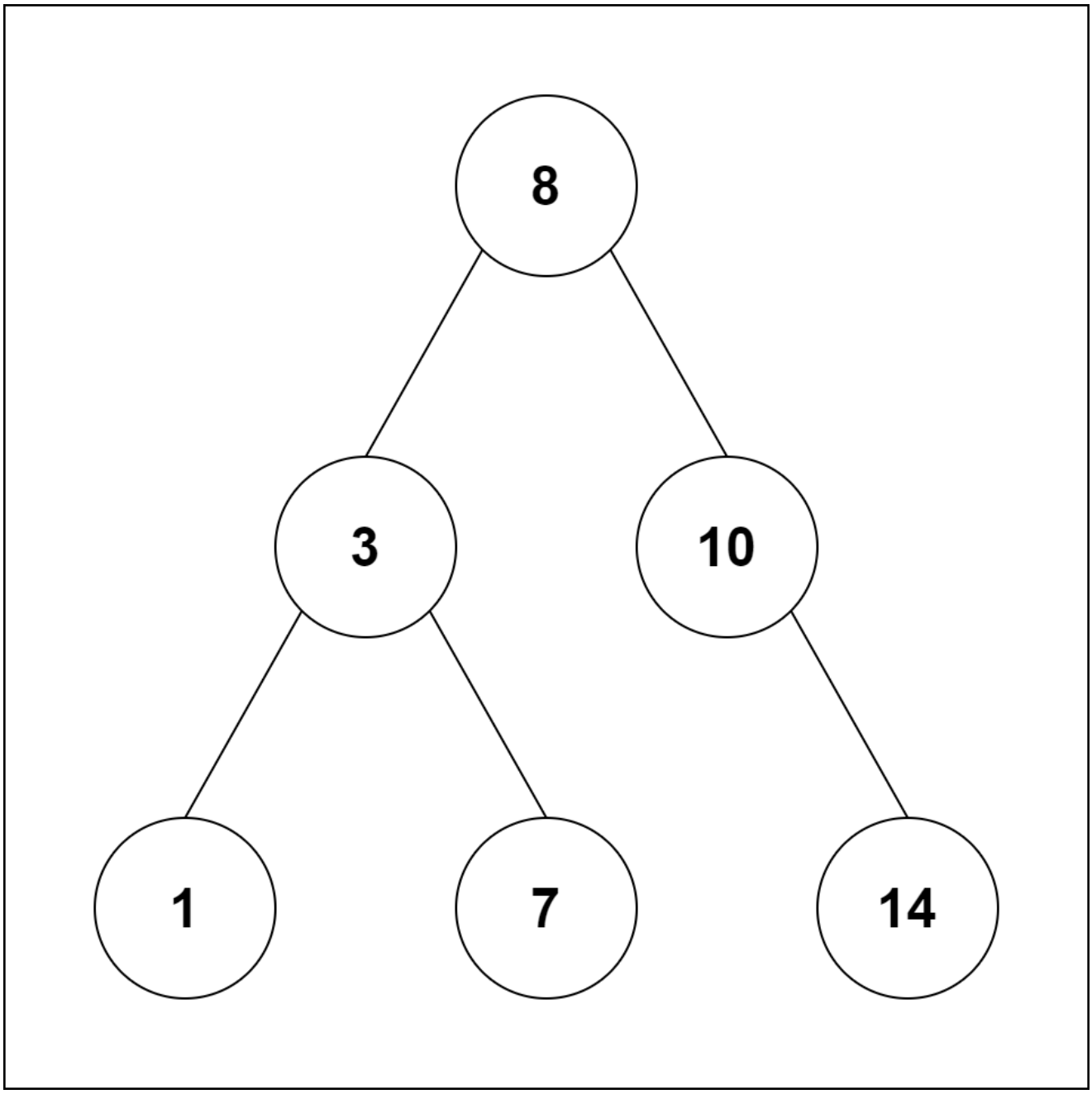
6 is to be deleted.



copy the value of its child to the node and delete the child.



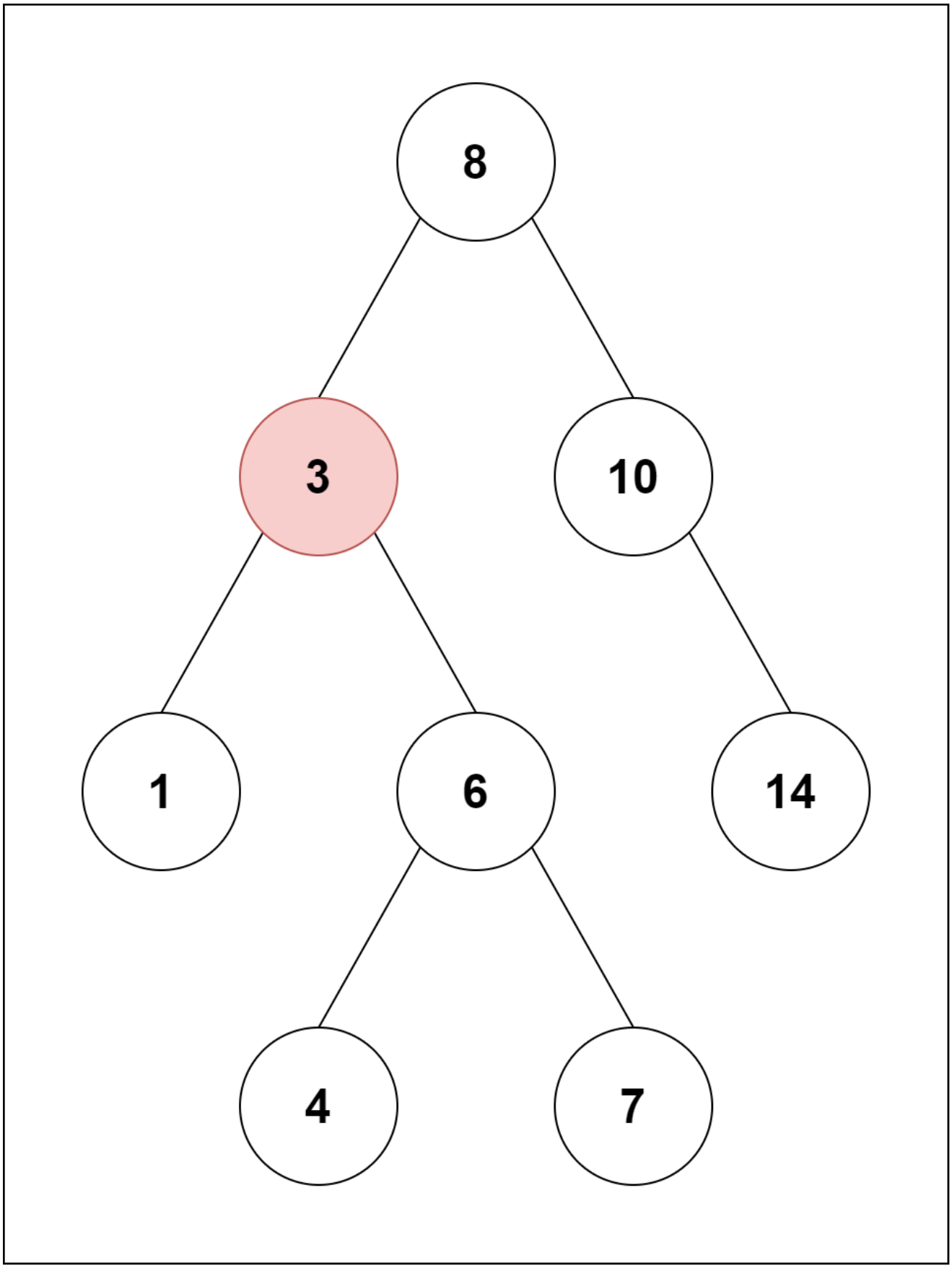
Final tree.



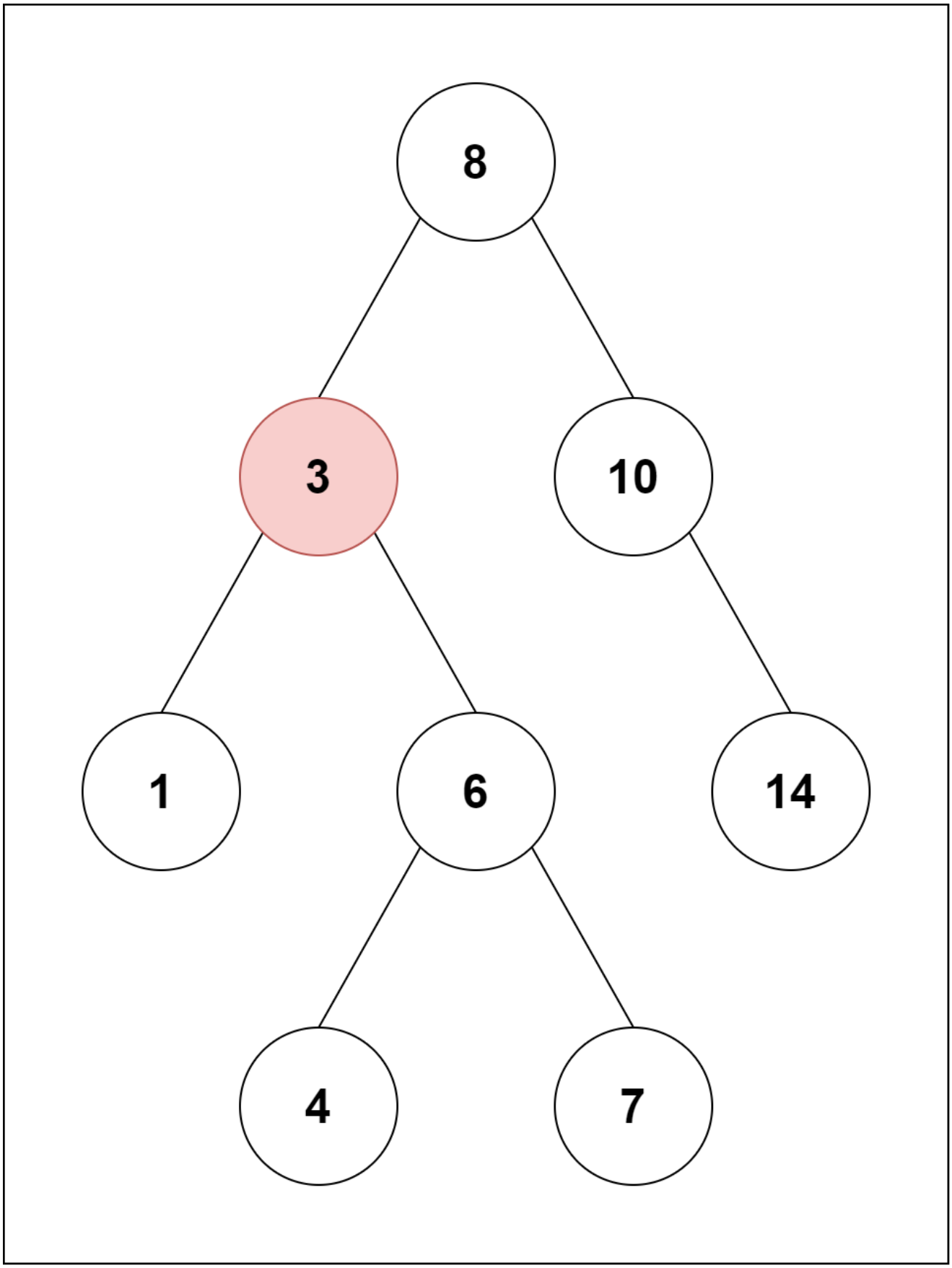
Case III:
In the third scenario, things get a bit more interesting. The node we want to delete has two children. Here's what we do:

1. First, we find the **inorder successor** of the node we want to delete. The inorder successor is essentially the smallest node in the right subtree of the node to be deleted.
2. Next, we replace the node we want to delete with this inorder successor. It's like swapping seats with someone.
3. Finally, we remove the inorder successor from its original position in the tree.

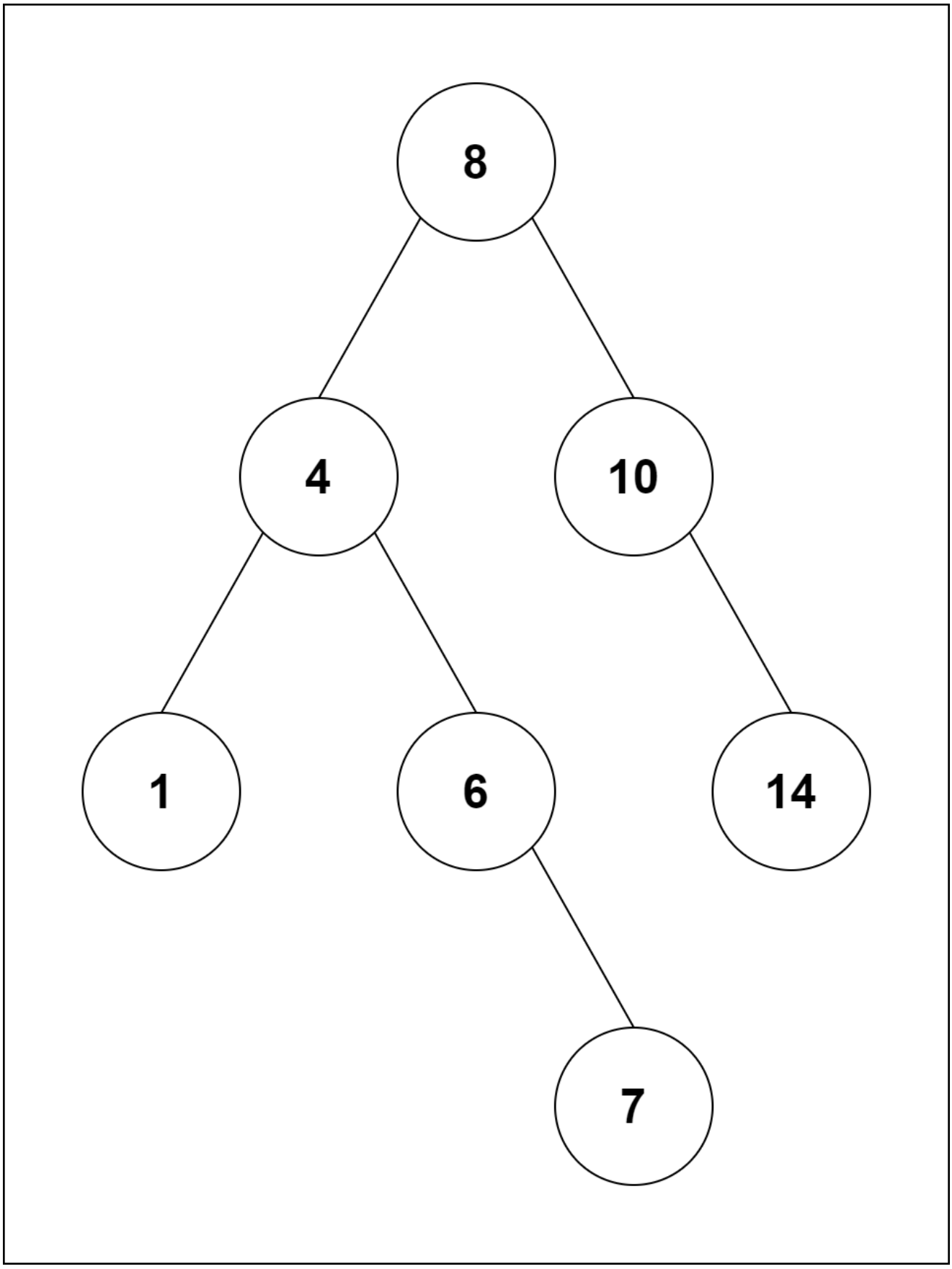
3 is to be deleted.



Copy the value of the inorder successor (4) to the node.



Delete the inorder successor.



These are the three cases you need to consider when deleting a node from a Binary Search Tree. Depending on whether the node is a leaf, has one child, or has two children, you follow the corresponding steps to maintain the structure and order of the tree.

C++ Example

```cpp
// Binary Search Tree operations in C++

#include <iostream>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);

    // Traverse root
    cout << root->key << " -> ";

    // Traverse right
    inorder(root->right);
  }
}

// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

// Find the inorder successor
```

```c
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver code
int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  root = insert(root, 3);
```

```
    root = insert(root, 1);
    root = insert(root, 6);
    root = insert(root, 7);
    root = insert(root, 10);
    root = insert(root, 14);
    root = insert(root, 4);

    cout << "Inorder traversal: ";
    inorder(root);

    cout << "\nAfter deleting 10\n";
    root = deleteNode(root, 10);
    cout << "Inorder traversal: ";
    inorder(root);
}
```

Binary Search Tree Complexities

Let's talk about the complexities associated with Binary Search Trees. Remember, these complexities tell us how efficient or how fast different operations are when we work with a Binary Search Tree.

Time Complexity

- When we perform a **search** operation, it's usually pretty quick. In the **best case**, it only takes O(log n) time, which is a very efficient search. However, in the **average case**, it also takes O(log n) time. But sometimes, in the **worst case**, it can take O(n) time, which is slower. The "n" here is the total number of nodes in our tree.
- When we do an **insertion** operation, it's quite similar to search. In the **best** and **average cases**, it takes O(log n) time. But, again, in the **worst case**, it can be as slow as O(n).
- Now, for the **deletion** operation, it's pretty much the same as insertion and search. In the **best** and **average cases**, it takes O(log n) time. Yet, in the **worst case**, it can take O(n) time.

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Space Complexity

- For all these operations, whether it's searching, inserting, or deleting, the space we need to store things in our computer memory is O(n). This means the space complexity for all these operations is O(n).

So, when we're working with Binary Search Trees, we usually focus more on the time complexity because we want our operations to be fast. But keep in mind that in some situations, like the worst-case scenarios, things can slow down a bit.

Binary Search Tree Applications

Let's explore some real-world uses of Binary Search Trees:

1. **Multilevel Indexing in Databases**: Imagine you have a massive database with tons of data, and you need to quickly find specific information. Binary Search Trees help in creating efficient indexes, which are like organized guides to your data. These indexes speed up searching and sorting operations, making databases much more efficient.

2. **Dynamic Sorting**: Sometimes, you have a constantly changing list of items that need to be sorted quickly. Binary Search Trees come to the rescue here. They allow you to maintain a sorted list, and when new items are added or old ones are removed, they can be reorganized on the fly. This dynamic sorting is super useful in various computer applications.

3. **Managing Virtual Memory in Unix Kernel**: Operating systems like Unix use Binary Search Trees to manage memory. Think of your computer's memory as a finite resource, and programs running on your computer need space in this memory to function. Binary Search Trees help manage this memory efficiently by keeping track of which parts are in use and which are available. This ensures that your computer runs smoothly, even when running multiple applications.

So, Binary Search Trees aren't just abstract data structures; they have real-world applications that make our digital lives more efficient and organized.