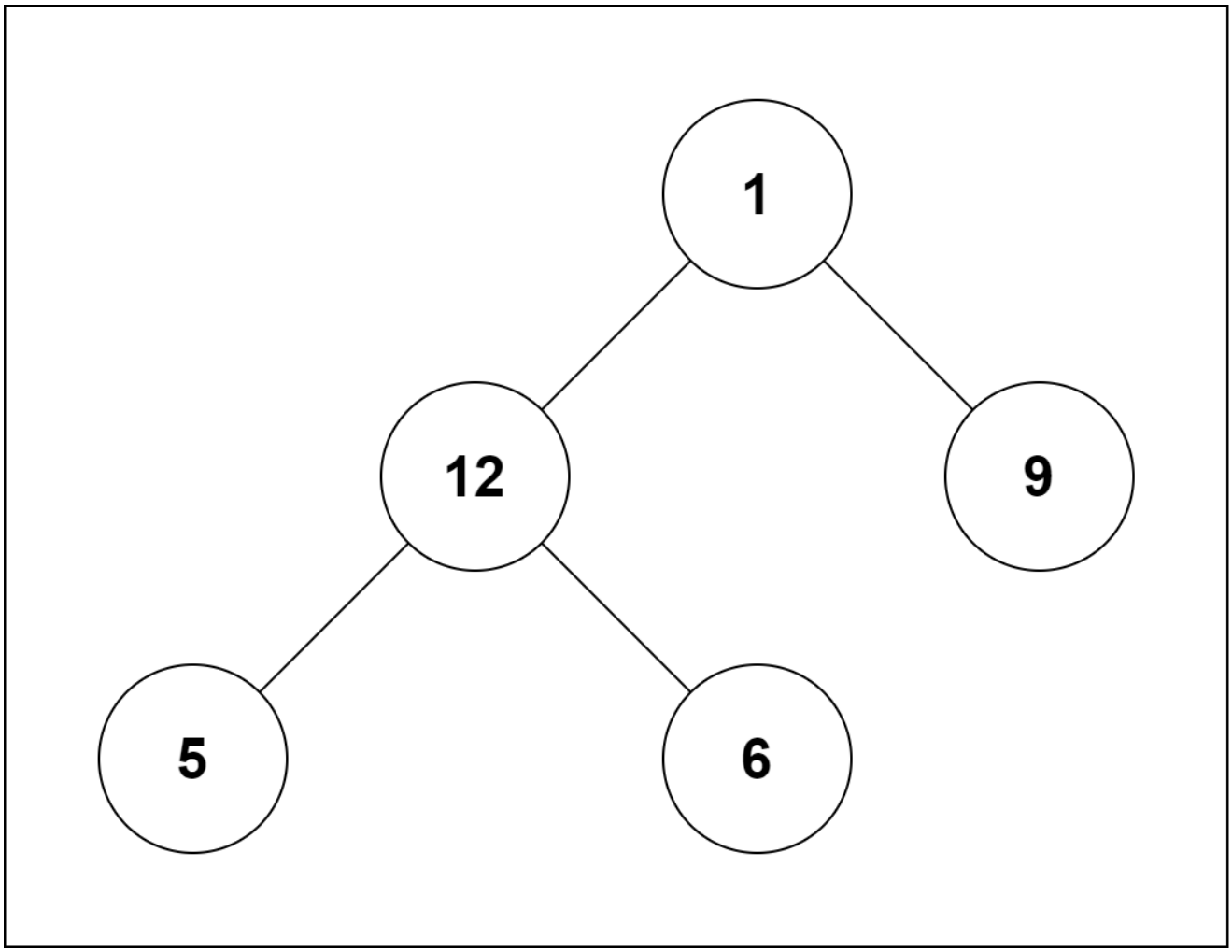
# Tree Traversal

When we talk about traversing a tree, we mean visiting every single node in that tree. It's like taking a tour of the tree to see what's there. Why would we want to do this? Well, there are many reasons. Maybe we want to add up all the values in the tree, find the biggest one, or perform some other operation on the data inside the tree.

Now, here's the interesting part: when it comes to trees, we have options. You see, in linear data structures like arrays, stacks, queues, or linked lists, there's usually only one way to read the data - from start to finish. But with a tree, things are different. It's like exploring a maze with multiple paths. You can traverse, or move through, a tree in various ways, depending on what you want to do with the data. So, trees give us this flexibility in how we explore and work with our data.




Now, let's talk about how we can read the elements of the tree in the image above. Imagine you're reading a book from the top left corner and moving left to right.

If we do that with our tree, it would look like this:

```
1 -> 12 -> 5 -> 6 -> 9
```

Or, imagine starting from the bottom left corner and moving left to right:

```
5 -> 6 -> 12 -> 9 -> 1
```

But here's the thing - even though these methods are quite straightforward, they don't really respect the structure of the tree. They only consider the depth of the nodes, not their position in the hierarchy.

So, instead, we use traversal methods that take into account the fundamental structure of a tree, which looks something like this:

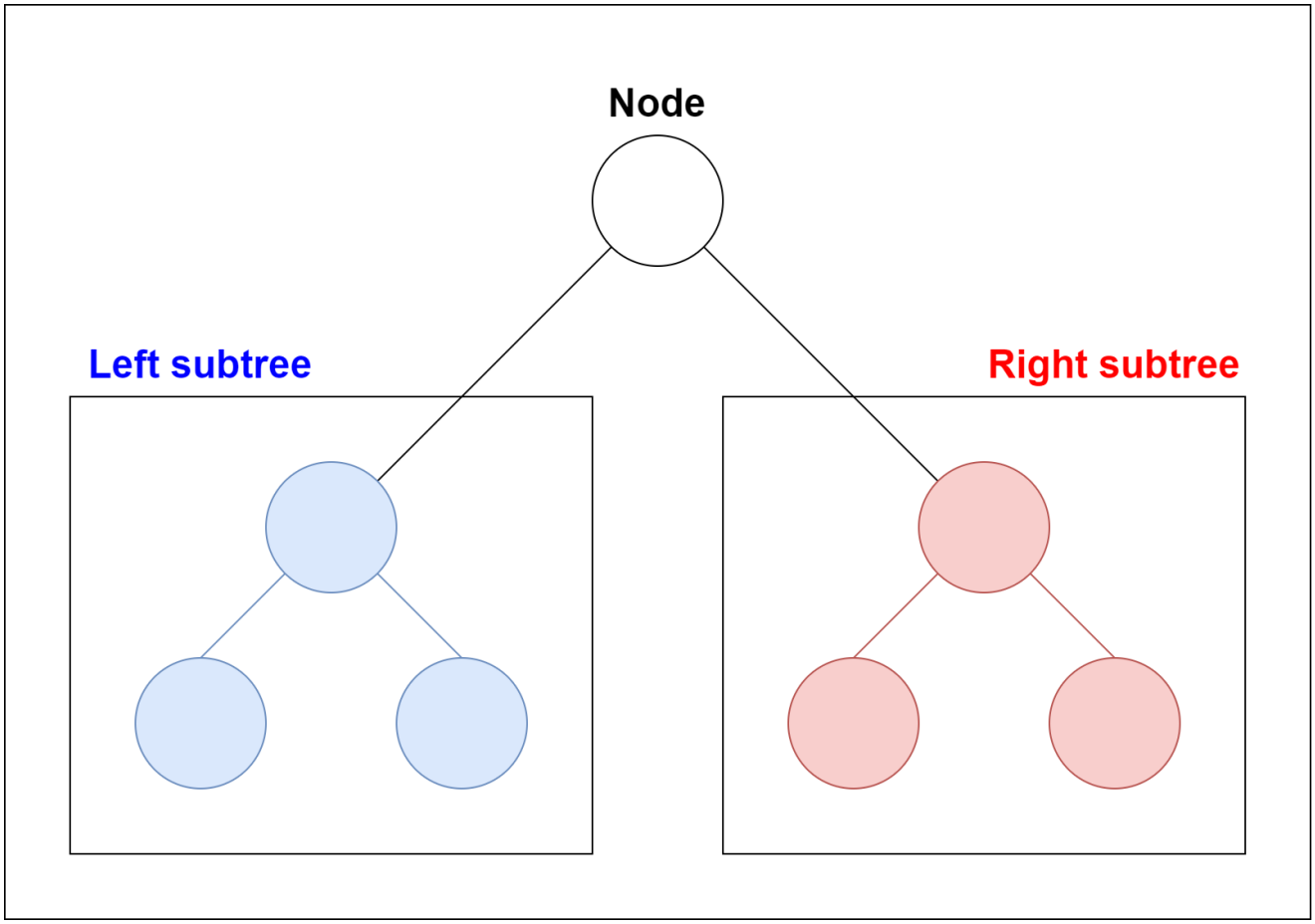
```
struct node {
        int data;
    struct node* left;
```

```
    struct node* right;
}
```

Now, think of the `struct node` pointed to by `left` and `right` as not just individual nodes but as whole sub-trees with their own branches.

Following this structure, every tree is basically a combination of three things:

1. A node carrying some data.
2. A left sub-tree.
3. A right sub-tree.




Our main goal here is to visit every single node in the tree, so we need to visit all the nodes in the left sub-tree, then the root node, and finally, all the nodes in the right sub-tree.

Now, depending on the order in which we do this, there can be three types of traversal.

---

## Inorder traversal

Alright, let's dive into our first tree traversal method, which is called "Inorder traversal."

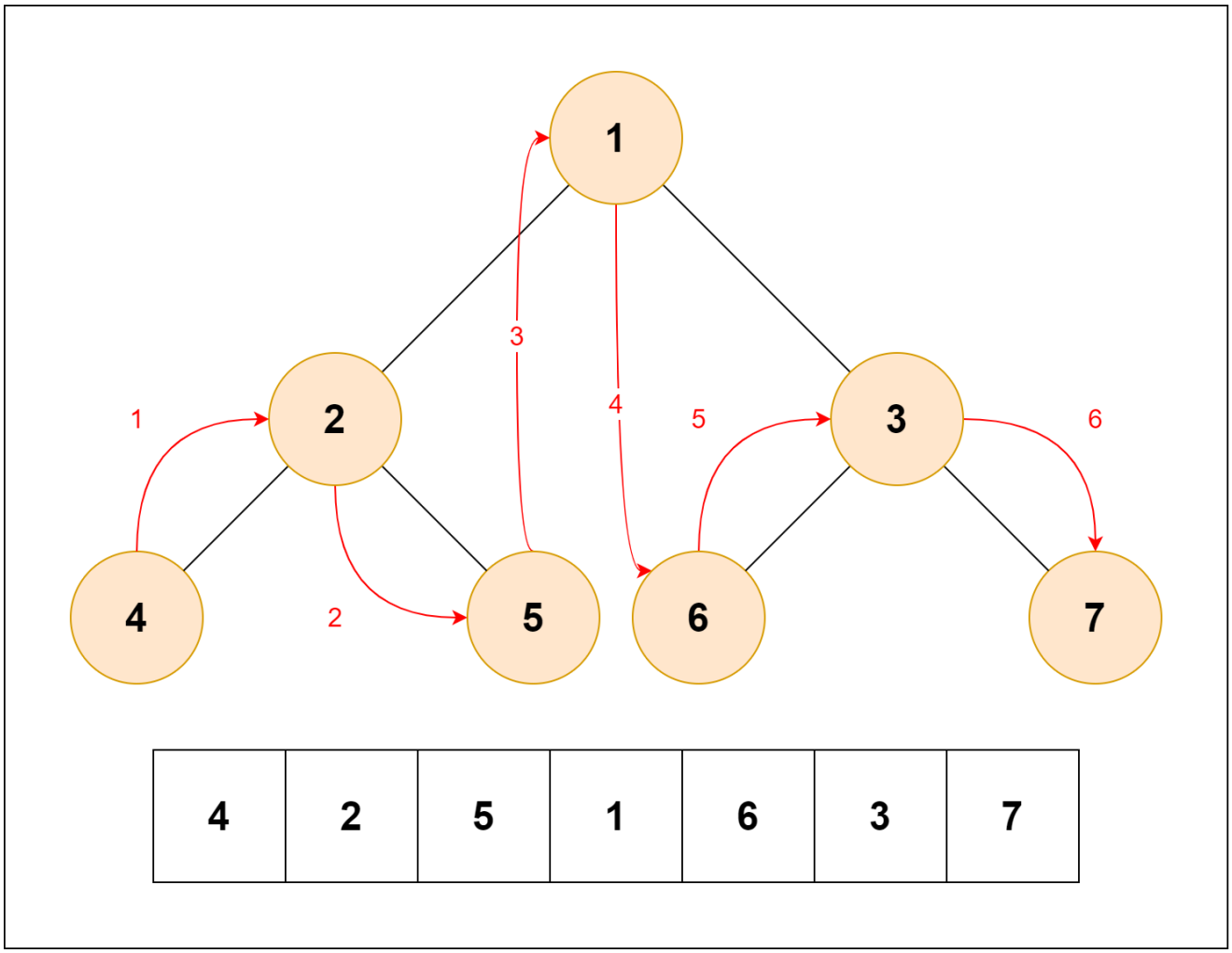
Now, imagine you're in a dense forest, and you want to explore it systematically. Here's how you'd do it:

1. First, you'd explore all the trees to your left.
2. Then, you'd explore the tree right in front of you, the one you're currently in.
3. Finally, you'd move on to explore all the trees to your right.

In the world of trees, especially binary trees, this method is pretty similar. It's called "Inorder traversal," and it works like this:

1. First, you visit all the nodes in the left subtree.
2. Then, you move on to the root node.
3. And finally, you visit all the nodes in the right sub tree.

```
inorder(root->left)
display(root->data)
inorder(root->right)
```




## Preorder traversal

Now, let's talk about another method of exploring trees, called "Preorder traversal."

Imagine you're on a camping trip and you come across a group of trees. Here's how you decide to explore them:

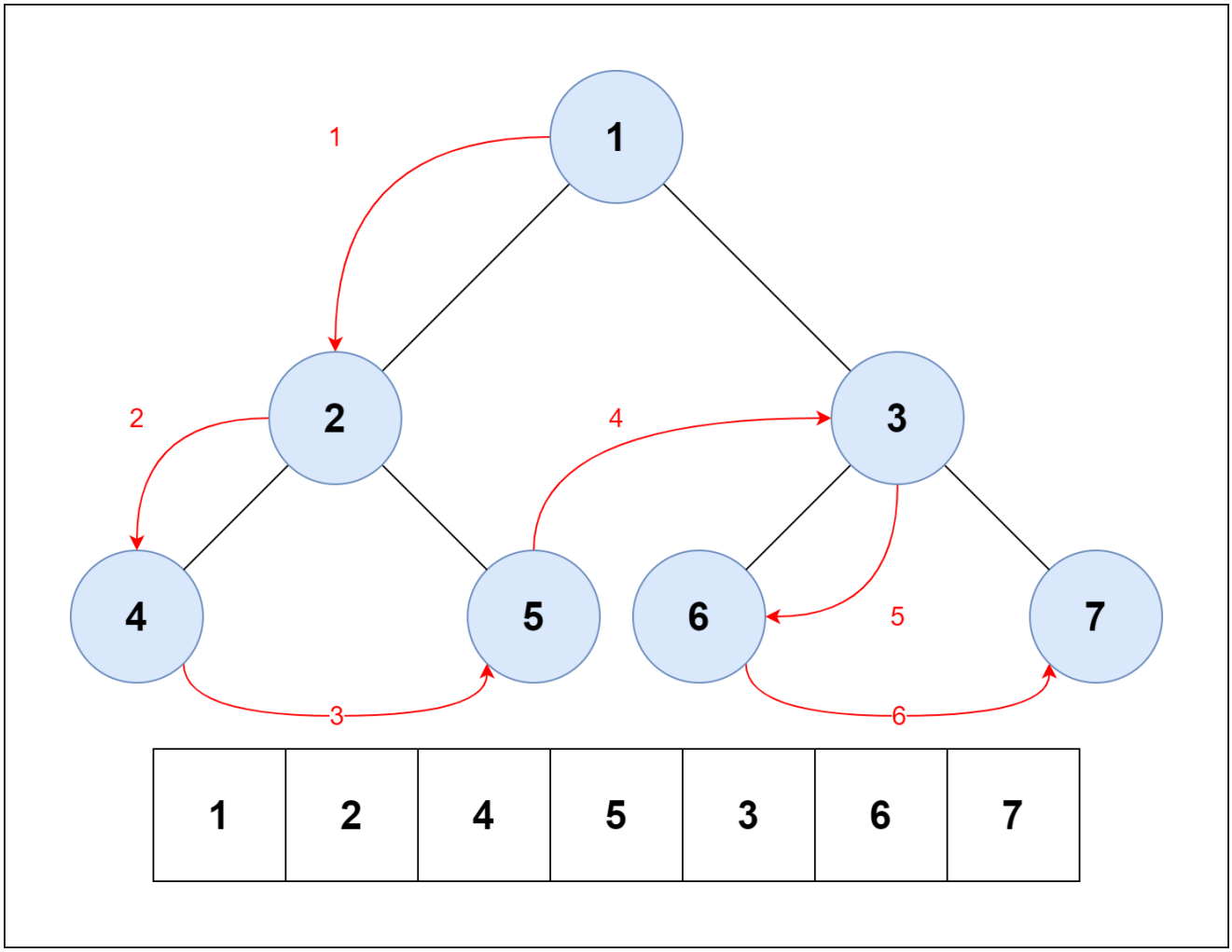
1. First, you visit the tree right in front of you, the one you're currently at.
2. Then, you move on to explore all the trees to your left.
3. Finally, you explore all the trees to your right.

In the world of trees, particularly binary trees, this method is known as "Preorder traversal." It goes like this:

1. You start by visiting the root node.
2. Then, you move on to visit all the nodes in the left subtree.
3. And finally, you visit all the nodes in the right subtree.

So, in a sense, it's like you're exploring the tree in a systematic way, beginning with the current node and then working your way through the left and right subtrees. This method helps you get a good overview of the tree's structure.

```
display(root->data)
preorder(root->left)
preorder(root->right)
```



# Postorder traversal

Let's explore another tree traversal method called "Postorder traversal."

Imagine you're in a forest, and you want to observe the trees in a particular way:

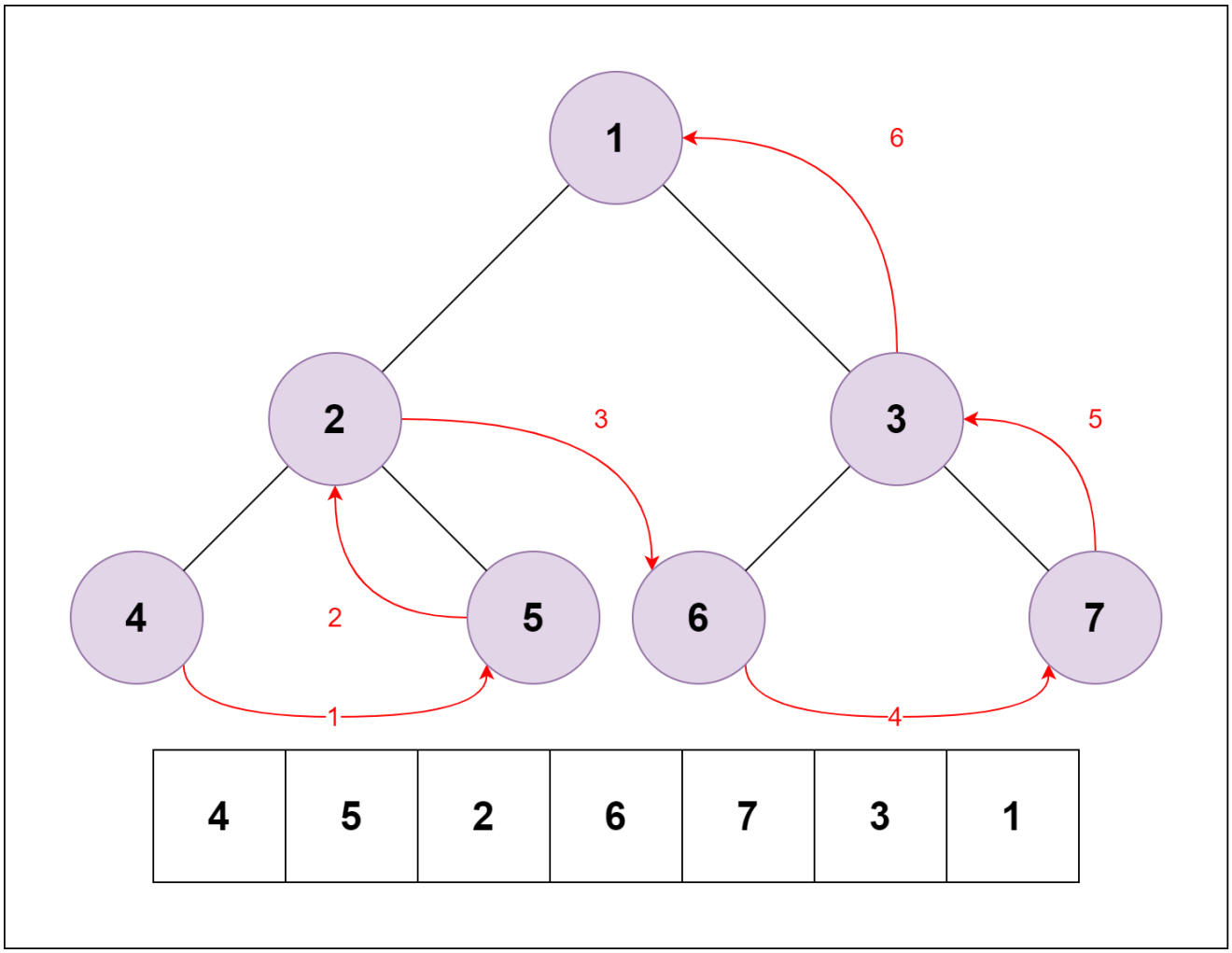
1. First, you explore all the trees to your left.
2. Then, you move on to explore all the trees to your right.
3. Finally, you carefully note the tree you're currently standing at.

In the language of trees, especially binary trees, this method is known as "Postorder traversal." Here's how it works:

1. You start by visiting all the nodes in the left subtree.
2. Then, you visit all the nodes in the right subtree.
3. Finally, you pay attention to the root node.

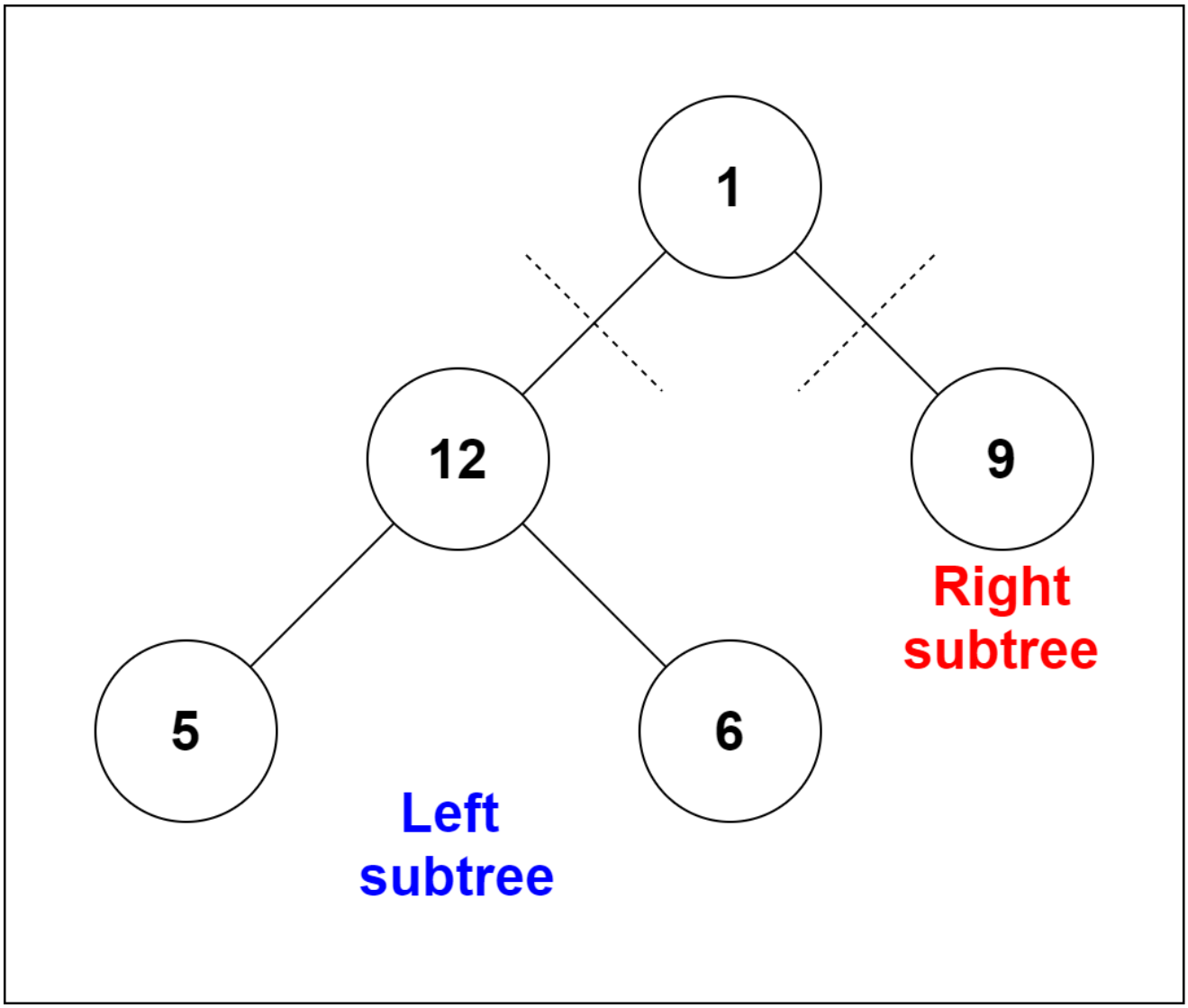
So, in this approach, you first delve deep into the left and right subtrees, and only when you've finished that exploration do you focus on the current node. This method helps you get a comprehensive view of the entire tree, from the bottom up.

```
postorder(root->left)
postorder(root->right)
display(root->data)
```



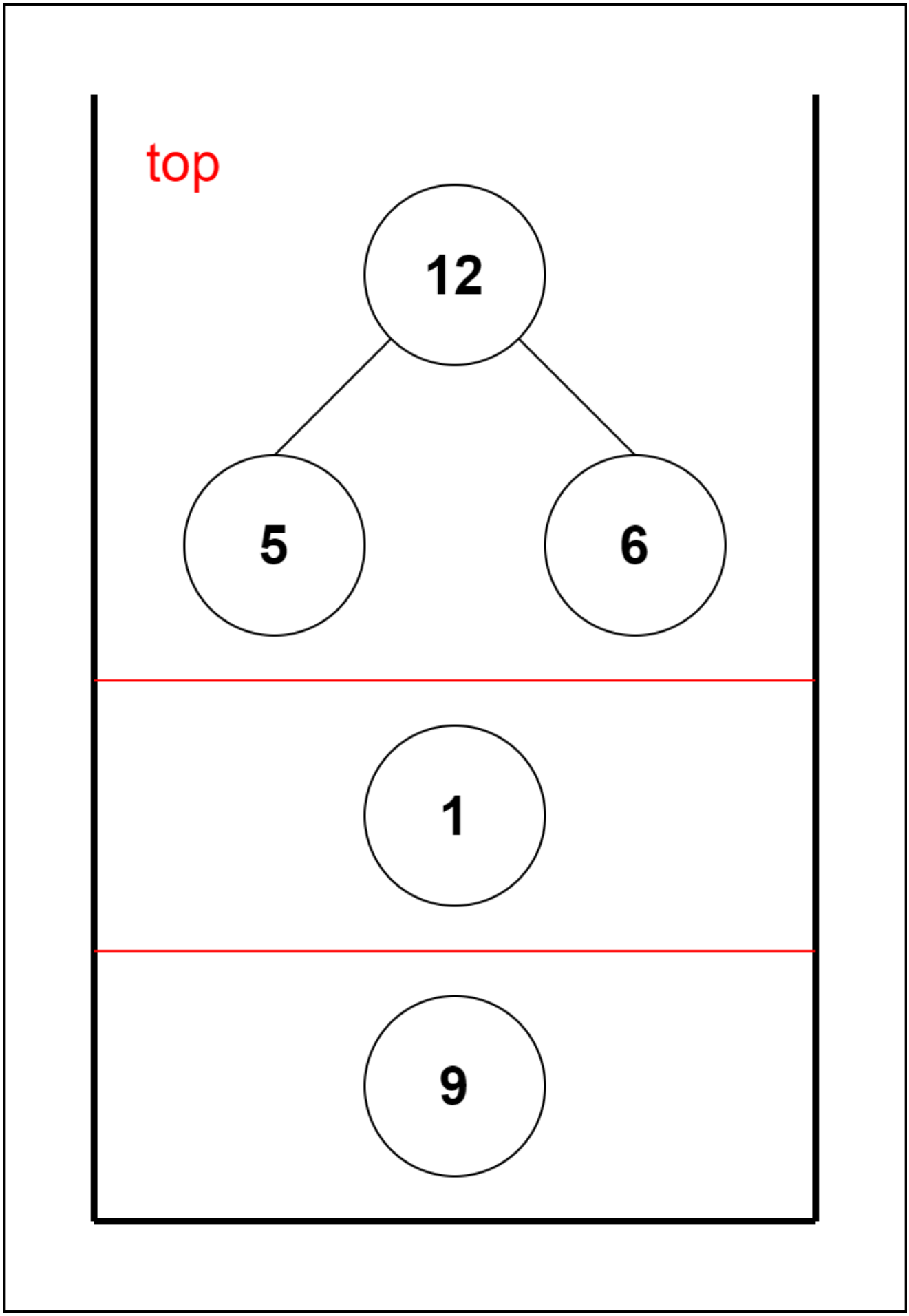

---

Let's visualize in-order traversal. We start from the root node.




We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a stack so that we remember.

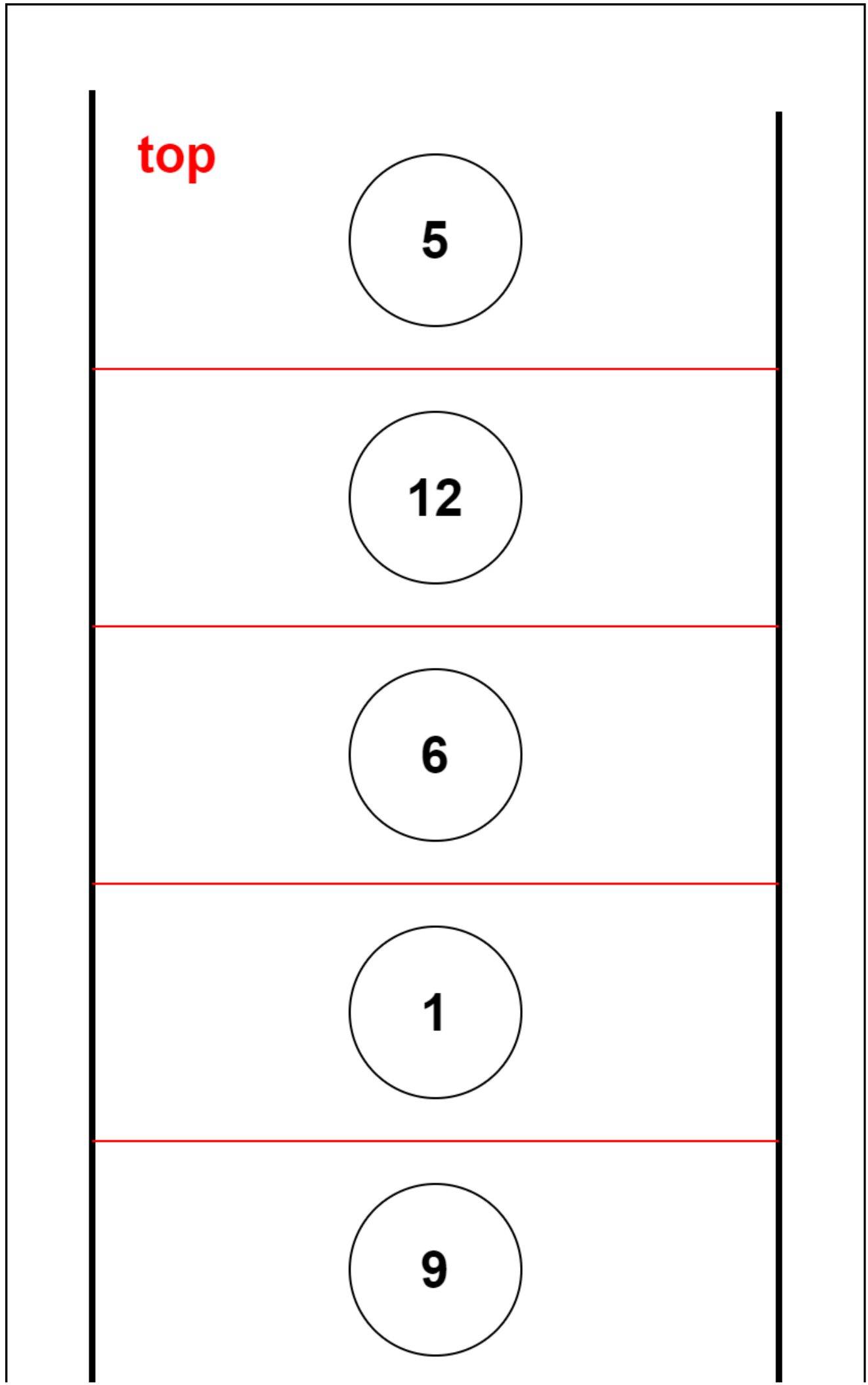


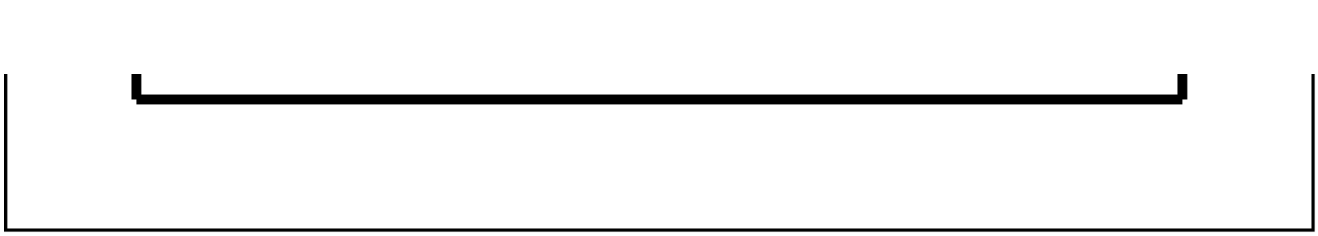

Now we traverse to the subtree pointed on the TOP of the stack.

Again, we follow the same rule of inorder:

```
Left subtree -> root -> right subtree
```

After traversing the left subtree, we are left with;


top
5
12
6
1
9



Since the node "5" doesn't have any subtrees, we print it directly. After that we print its parent "12" and then the right child "6".

Putting everything on a stack was helpful because now that the left-subtree of the root node has been traversed, we can print it and go to the right subtree.

After going through all the elements, we get the inorder traversal as

```
5 -> 12 -> 6 -> 1 -> 9
```

We don't have to create the stack ourselves because recursion maintains the correct order for us.

---

## C++ Example

```
// Tree traversal in C++

#include <iostream>
using namespace std;

struct Node {
  int data;
  struct Node *left, *right;
  Node(int data) {
    this->data = data;
    left = right = NULL;
  }
};

// Preorder traversal
void preorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  cout << node->data << "->";
  preorderTraversal(node->left);
  preorderTraversal(node->right);
}

// Postorder traversal
void postorderTraversal(struct Node* node) {
```

```
  if (node == NULL)
    return;

  postorderTraversal(node->left);
  postorderTraversal(node->right);
  cout << node->data << "->";
}

// Inorder traversal
void inorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  inorderTraversal(node->left);
  cout << node->data << "->";
  inorderTraversal(node->right);
}

int main() {
  struct Node* root = new Node(1);
  root->left = new Node(12);
  root->right = new Node(9);
  root->left->left = new Node(5);
  root->left->right = new Node(6);

  cout << "Inorder traversal ";
  inorderTraversal(root);

  cout << "\nPreorder traversal ";
  preorderTraversal(root);

  cout << "\nPostorder traversal ";
  postorderTraversal(root);
```