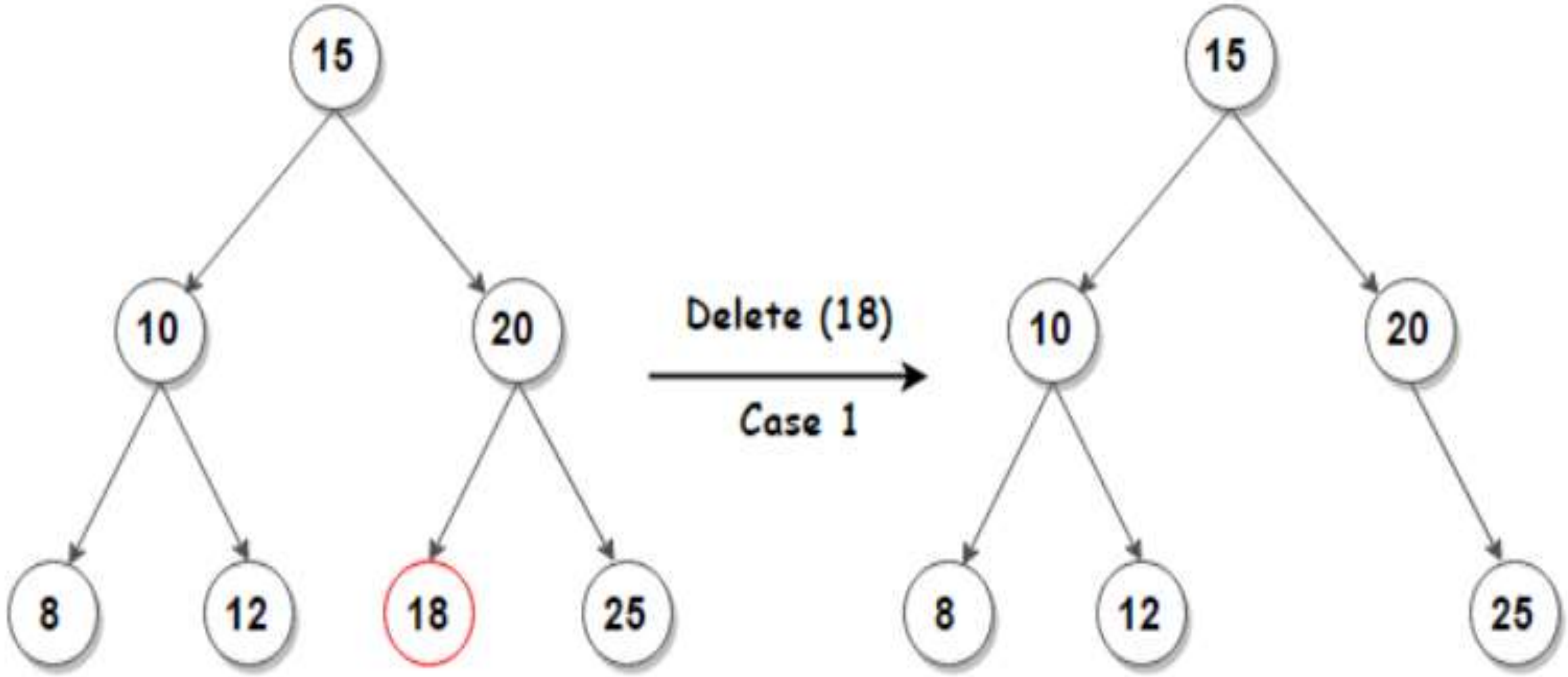
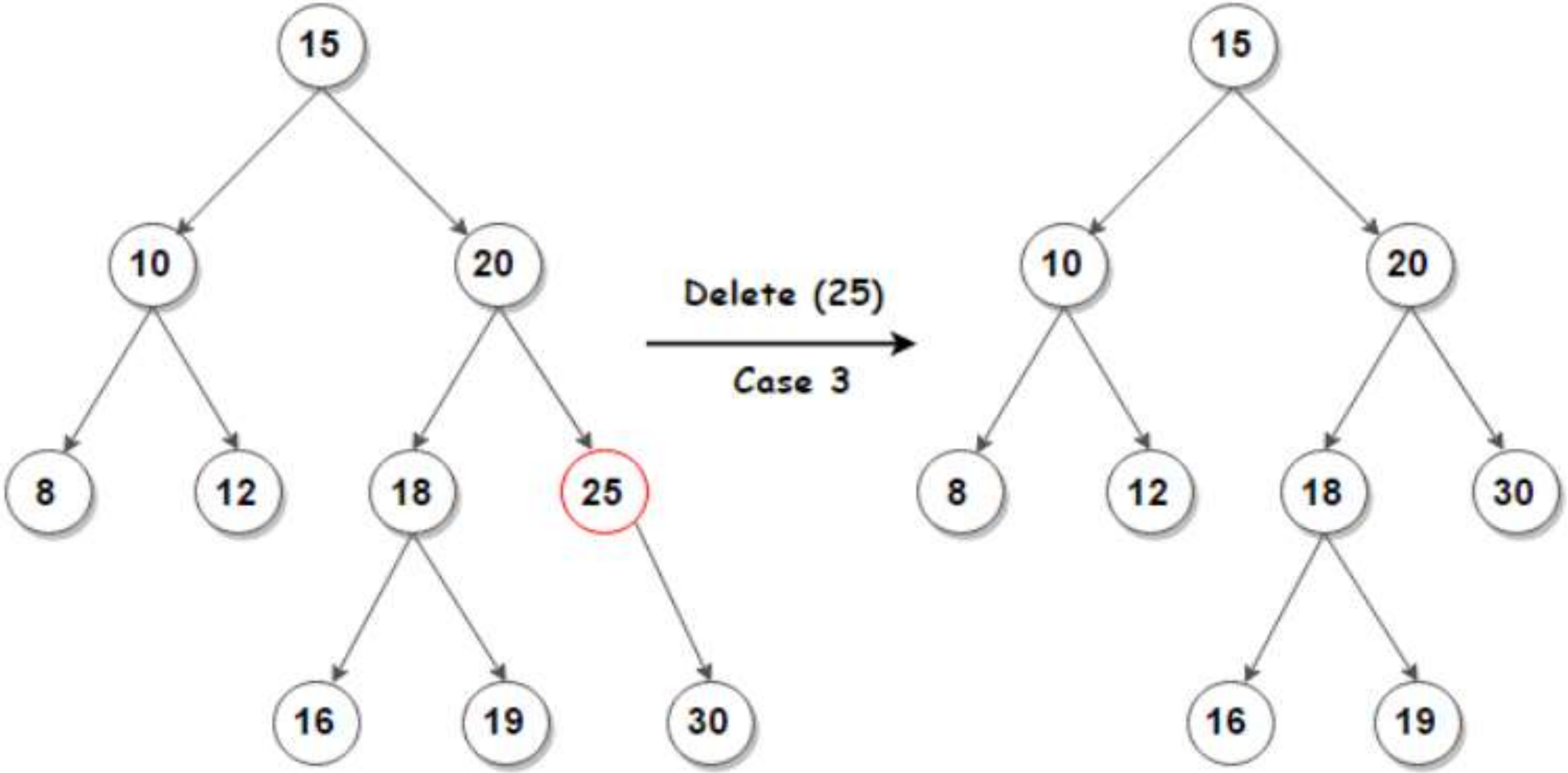
Deleting Elements from BST

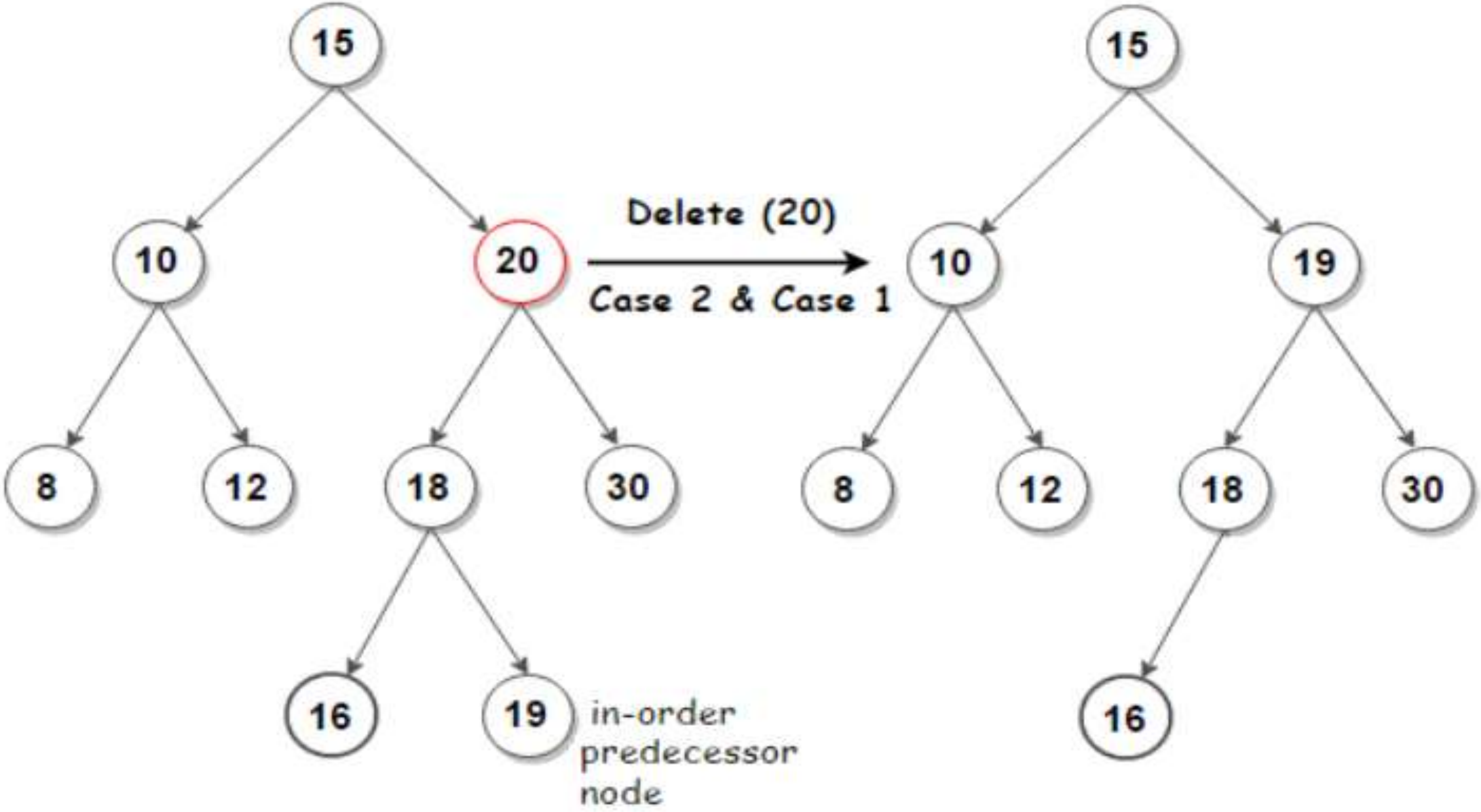
- Three Scenarios:
  - Scenario 1: No Children (Leave Nodes)



- Scenario 2: Deleting One Child node Such as example given below



- Scenario 3: Deleting Two Children node Such as example given below



```
// Data structure to store a Binary Search Tree node
struct Node {
	int data;
	Node *left, *right;
};

// Function to create a new binary tree node having given key
Node* newNode(int key)
{
	Node* node = new Node;
	node->data = key;
	node->left = node->right = nullptr;

	return node;
}
```

```
// Function to perform inorder traversal of the BST
void inorder(Node *root)
{
	if (root == nullptr)
		return;

	inorder(root->left);
	cout << root->data << " ";
	inorder(root->right);
}

// Helper function to find minimum value node in subtree rooted at curr
Node* minimumKey(Node* curr)
{
	while(curr->left != nullptr) {
		curr = curr->left;
	}
	return curr;
}
```

```
// Recursive function to insert a key into BST
Node* insert(Node* root, int key)
{
        // if the root is null, create a new node and return it
        if (root == nullptr)
                return newNode(key);

        // if given key is less than the root node, recur for left subtree
        if (key < root->data)
                root->left = insert(root->left, key);

        // if given key is more than the root node, recur for right subtree
        else
                root->right = insert(root->right, key);

        return root;
}
```

```
// Iterative function to search in subtree rooted at curr & set its parent
// Note that curr & parent are passed by reference
void searchKey(Node* &curr, int key, Node* &parent)
{
        // traverse the tree and search for the key
        while (curr != nullptr && curr->data != key)
        {
                // update parent node as current node
                parent = curr;

                // if given key is less than the current node, go to left subtree
                // else go to right subtree
                if (key < curr->data)
                        curr = curr->left;
                else
                        curr = curr->right;
        }
}
```

```
// Function to delete node from a BST
void deleteNode(Node*& root, int key)
{
	// pointer to store parent node of current node
	Node* parent = nullptr;

	// start with root node
	Node* curr = root;

	// search key in BST and set its parent pointer
	searchKey(curr, key, parent);

	// return if key is not found in the tree
	if (curr == nullptr)
		return;
```

```
// Case 1: node to be deleted has no children i.e. it is a leaf node
        if (curr->left == nullptr && curr->right == nullptr)
        {
                // if node to be deleted is not a root node, then set its
                // parent left/right child to null
                if (curr != root)
                {
                        if (parent->left == curr)
                                parent->left = nullptr;
                        else
                                parent->right = nullptr;
                }
                // if tree has only root node, delete it and set root to null
                else
                        root = nullptr;

                // deallocate the memory
                free(curr);    // or delete curr;
        }
```

```
// Case 2: node to be deleted has only one child
        else
        {
                // find child node
                Node* child = (curr->left)? curr->left: curr->right;

                // if node to be deleted is not a root node, then set its parent
                // to its child
                if (curr != root)
                {
                        if (curr == parent->left)
                                parent->left = child;
                        else
                                parent->right = child;
                }

                // if node to be deleted is root node, then set the root to child
                else
                        root = child;

                // deallocate the memory
                free(curr);
        }
}
```

```
// Case 3: node to be deleted has two children
        else if (curr->left && curr->right)
        {
                // find its in-order successor node
                Node* successor  = minimumKey(curr->right);

                // Replace current value with successor value
                curr->data = successor->data;;

                // recursively delete the successor. Note that the successor
                // will have at-most one child (right child)
                deleteNode(root, successor->data);

                        }
```

```
// main function
int main()
{
	Node* root = nullptr;
	int keys[] = { 15, 10, 20, 8, 12, 16 };

	for (int key : keys)
		root = insert(root, key);

	deleteNode(root, 16);
	inorder(root);

	return 0;
}
```

Height of tree code

```
int height(struct node* node)
{
        if (node == NULL)
                return 0;
        else {
                // Compute the height of each subtree
                int lheight = height(node->left);
                int rheight = height(node->right);
                // Use the larger one
                if (lheight > rheight)
                        return (lheight + 1);
                else
                        return (rheight + 1);
        }
}
```

Level Traversing of tree Elements

```
void printLevelOrder(struct node* root) // Function to print level order traversal a tree
{               int h = height(root);
                int i;
                for (i = 1; i <= h; i++)
                                printCurrentLevel(root, i);
}
void printCurrentLevel(struct node* root, int level) // Print nodes at a current level
{               if (root == NULL)
                                return;
                if (level == 1)
                                printf("%d ", root->data);
                else if (level > 1) {
                                printCurrentLevel(root->left, level - 1);
                                printCurrentLevel(root->right, level - 1);
                }
}
```