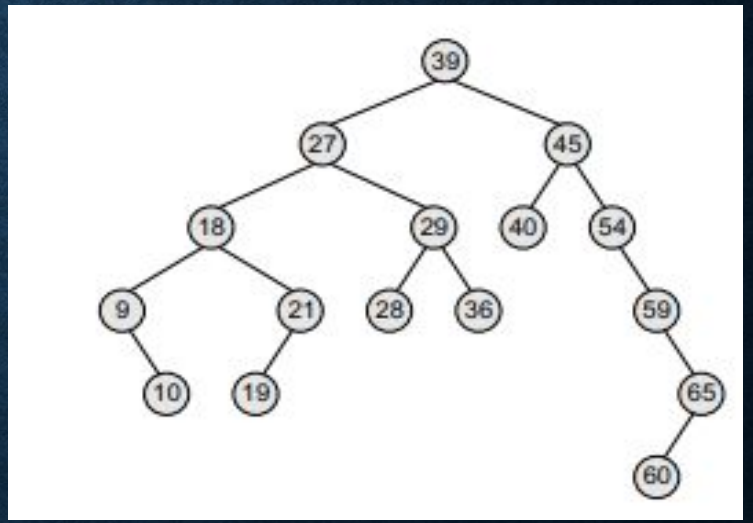
BINARY SEARCH TREE.

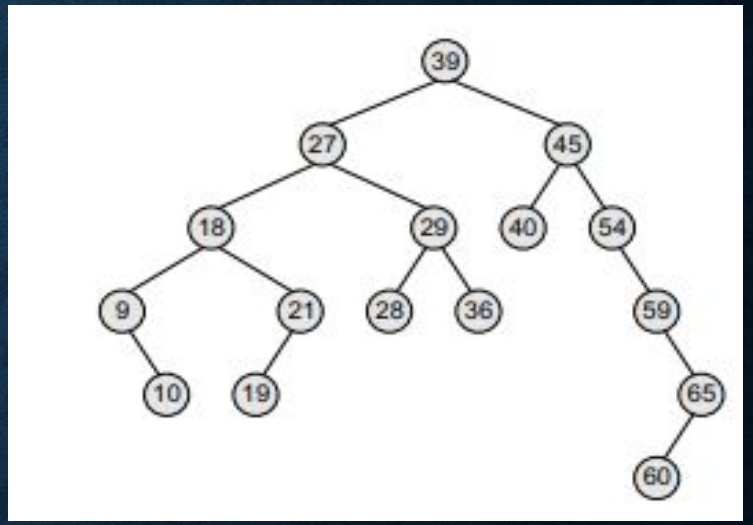
BINARY SEARCH TREE

- In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node



BINARY SEARCH TREE

- The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the root node.
- The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65.
- Recursively, each of the sub-trees also obeys the binary search tree constraint.



ADVANTAGES

1. Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced.

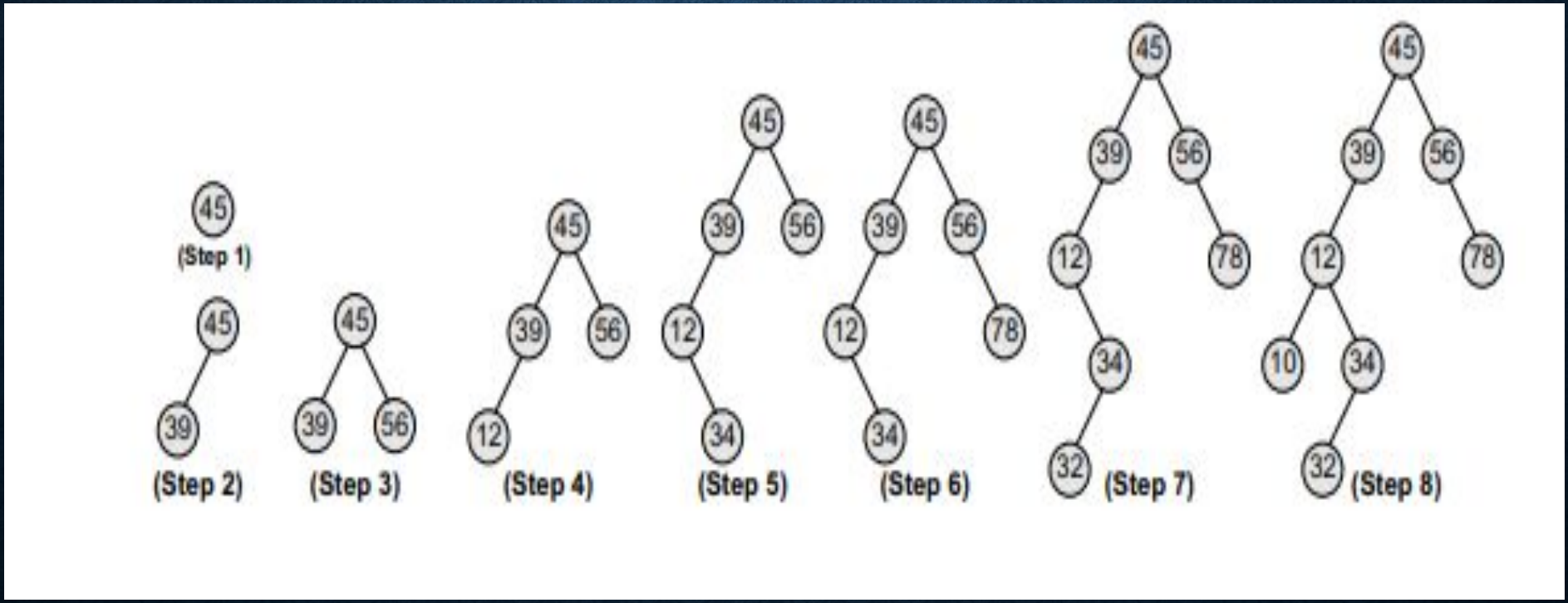
2. Binary search trees also speed up the insertion and deletion operations.

3.The tree has a speed advantage when the data in the structure changes rapidly

4. Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists

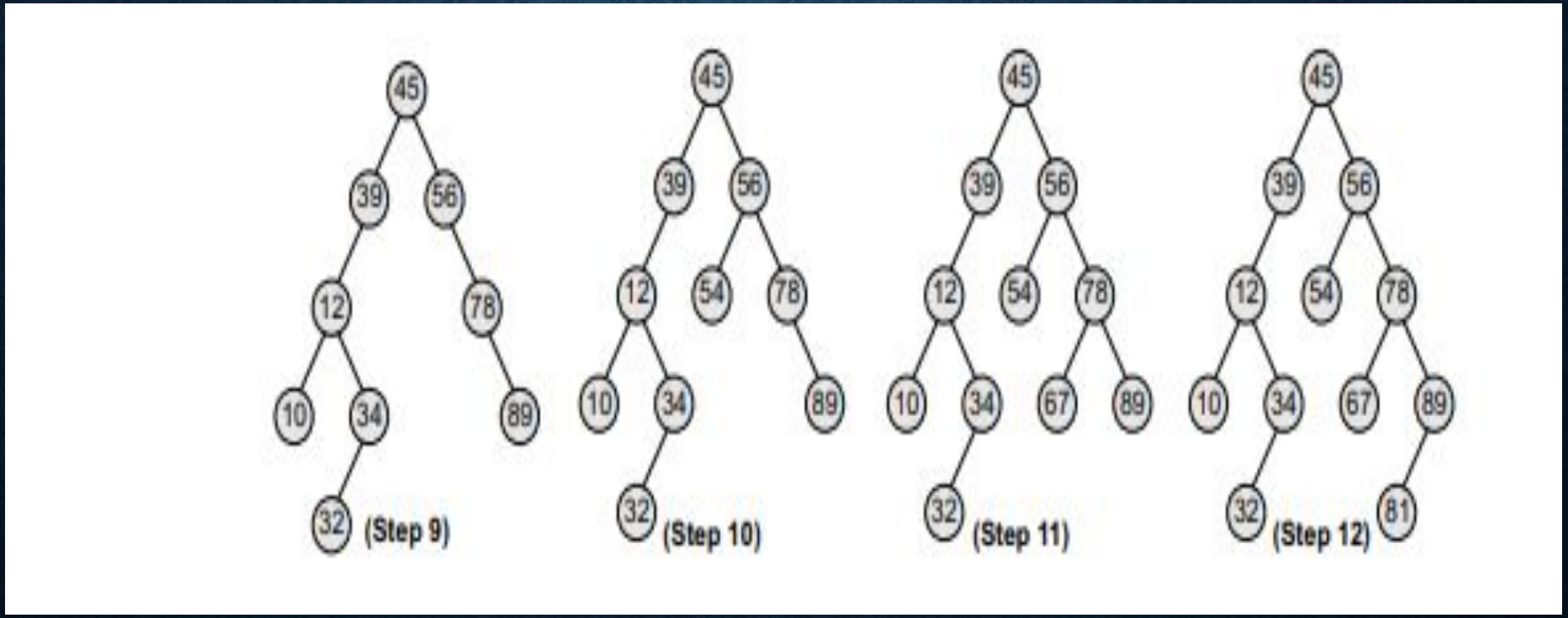
CREATE A BINARY SEARCH TREE

- 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



CREATE A BINARY SEARCH TREE

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



OPERATIONS ON BINARY SEARCH TREES

1. Searching for a node
2. Insertion of a node
3 deletion of a node

SEARCHING FOR A NODE IN A BST

- The searching process begins at the root node.
- The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message.
- However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the current node, it should be recursively called on the right child node.

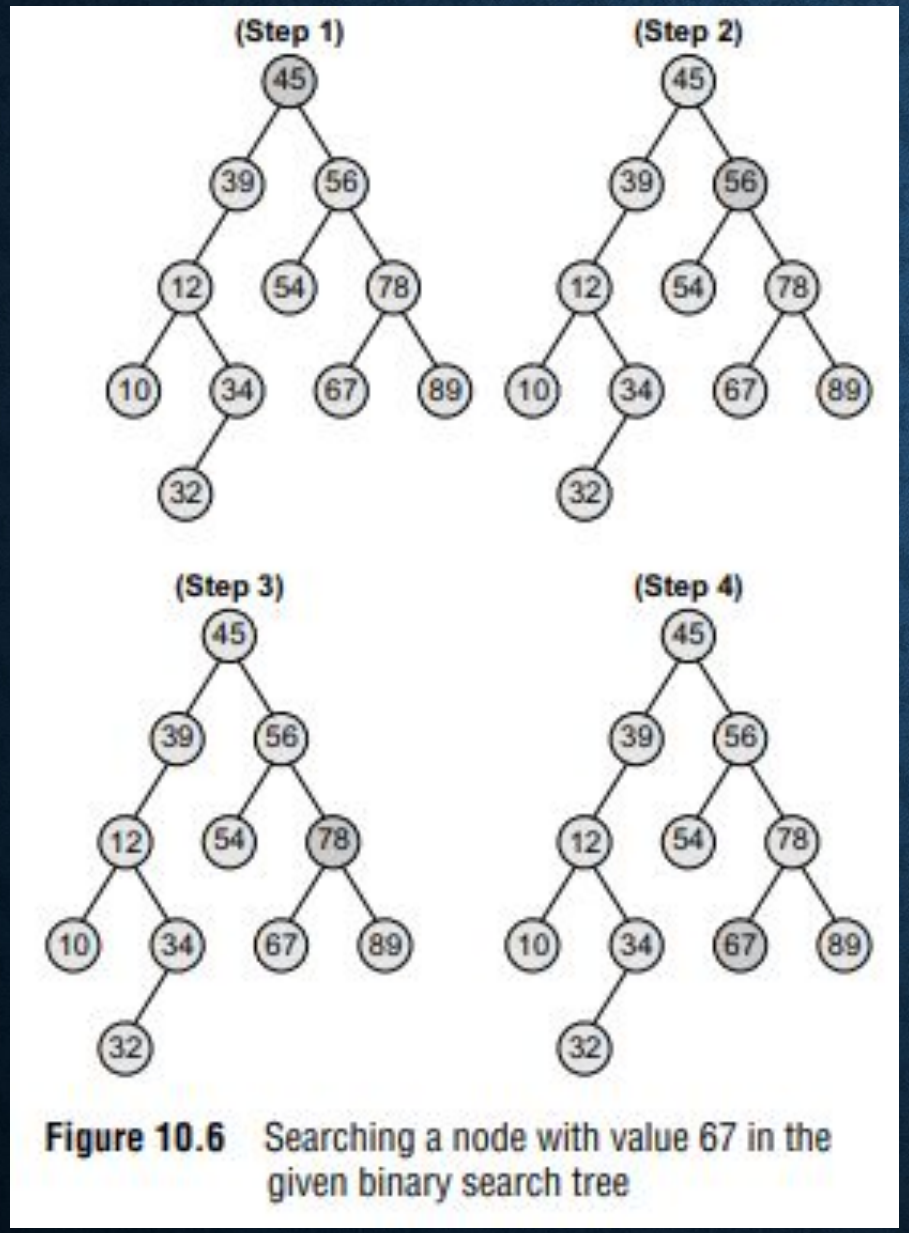


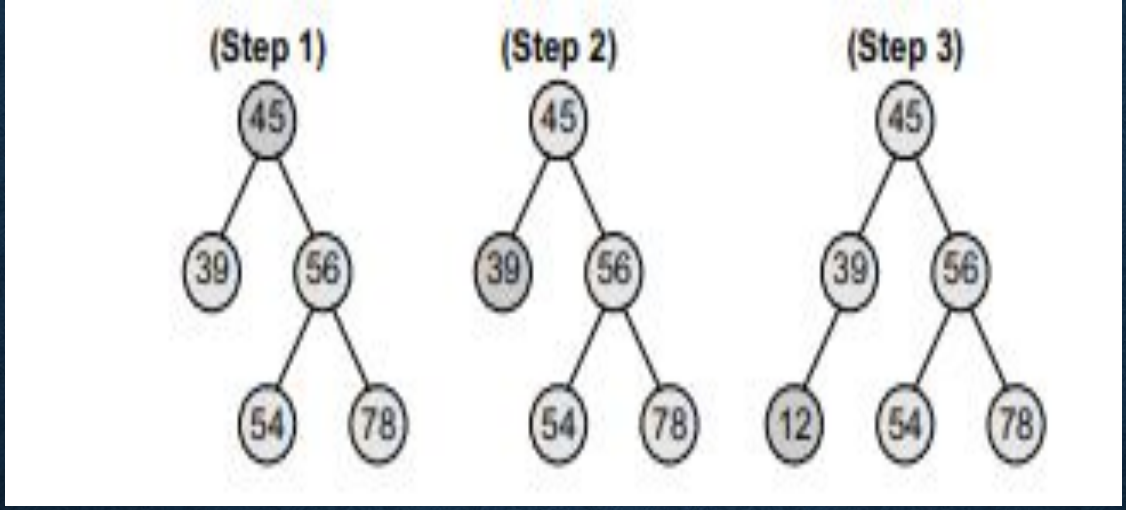
Figure 10.6 Searching a node with value 67 in the given binary search tree

```
struct node* search(struct node* root,
int key)
{
    // Base Cases: root is null or key
is present at root
    if (root == NULL || root->key ==
key)
        return root;


    // Key is greater than root's key
    if (root->key < key)
        return search(root->right,
key);


    // Key is smaller than root's key
    return search(root->left, key);
}
```

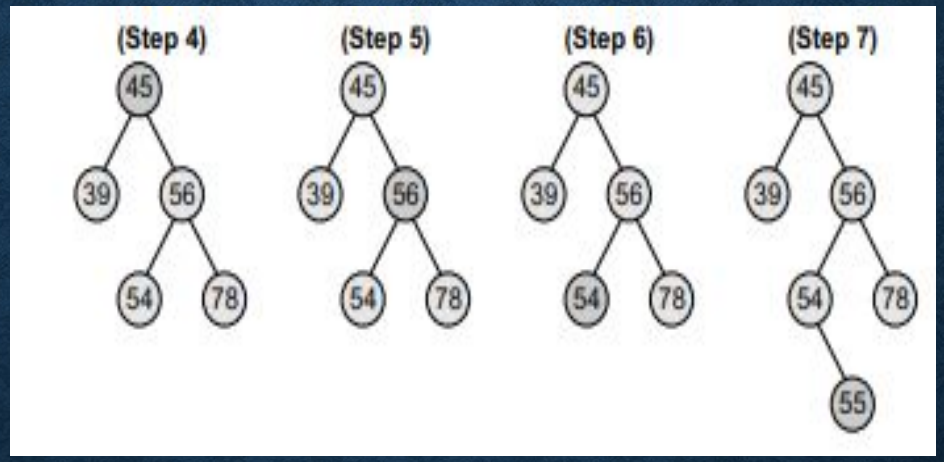
INSERTING A NEW NODE IN A BST

- The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree
- In Step 1 of the algorithm, the insert function checks if the current node of TREE is NULL. If it is NULL, the algorithm simply adds the node, else it looks at the current node's value and then recurs down the left or right sub-tree.
- If the current node's value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed.
- The insert function continues moving down the levels of a binary tree until it reaches a leaf node.
- The new node is added by following the rules of the binary search trees.
- node, the new node iThat is, if the new node's value is greater than that of the parent s inserted

STEPS TO INSERT NODE 12



STEPS TO INSERT NODE 55



INSERTION IN BINARY SEARCH TREE

```
struct node* insert(struct node* node, int key)
{
    // If the tree is empty, return a new node
    if (node == NULL)

            struct node* temp
            = (struct node*)malloc(sizeof(struct node));
         temp->key = key;
         temp->left = temp->right = NULL;
         return temp;

    // Otherwise, recur down the tree
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    // Return the (unchanged) node pointer
    return node;
}
```

DELETING A NODE FROM A BINARY SEARCH TREE

Case 1: Deleting a Node that has No Children: simply remove this node without any issue

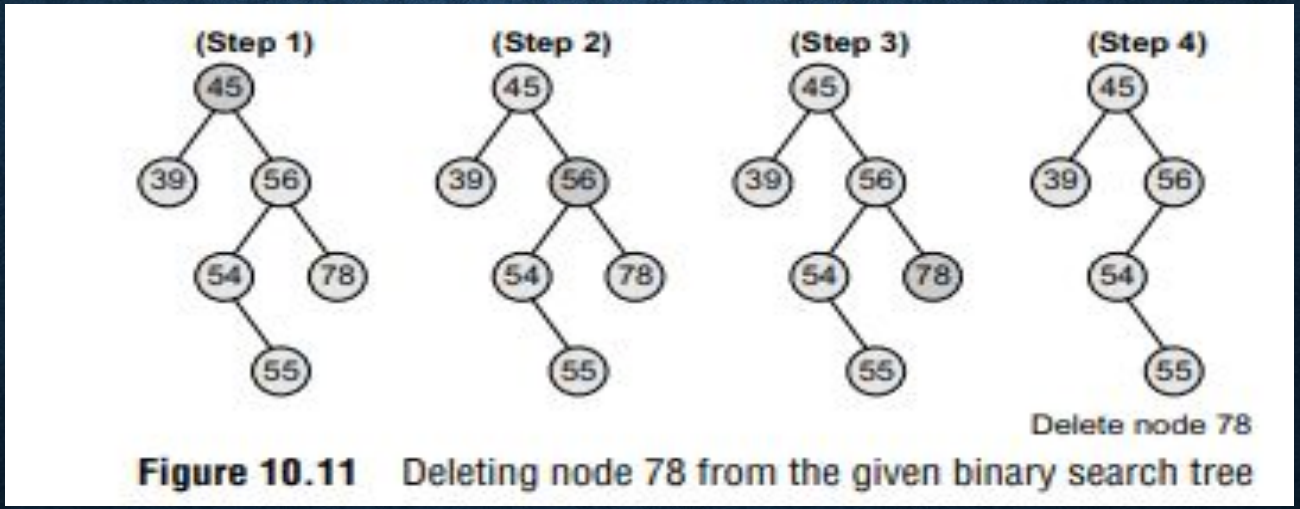


Figure 10.11 Deleting node 78 from the given binary search tree

DELETING A NODE FROM A BINARY SEARCH TREE

Case 2: Deleting a Node with One Child

- To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child.
- Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent

DELETING A NODE FROM A BINARY SEARCH TREE

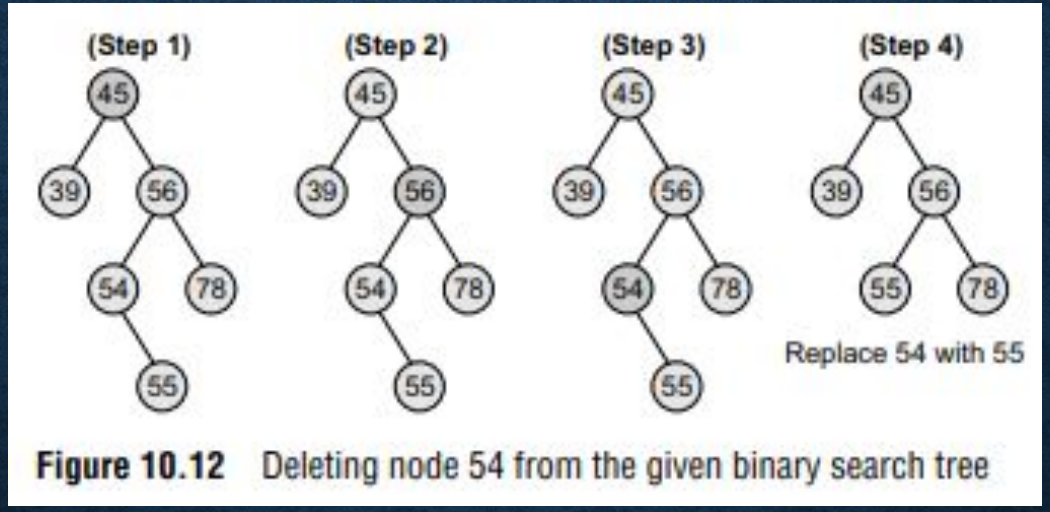


Figure 10.12 Deleting node 54 from the given binary search tree

DELETING A NODE FROM A BINARY SEARCH TREE

Case 3: Deleting a Node with Two Children:

- To handle this case, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases.

DELETING A NODE FROM A BINARY SEARCH TREE

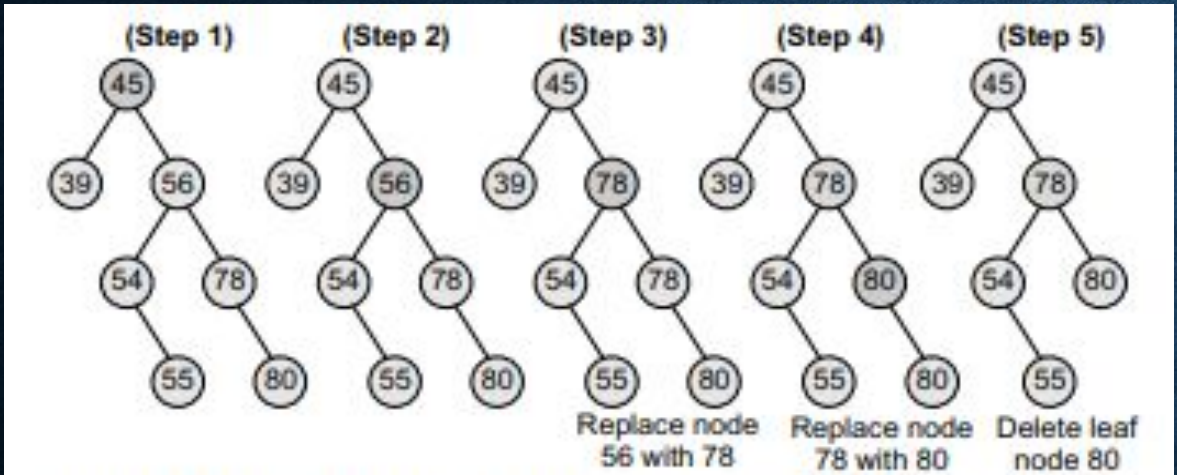


Figure 10.14 Deleting node 56 from the given binary search tree

1. To delete 56 we need to find its in order successor .
2. In order successor of 56 is smallest element in its right subtree that is 78.
3. So replace 56 with 78.
4. Now replace 78 with 80 (because 78 is duplicate so we have to delete 78 and 78 has only one child 80 so case 2 apply .)
5. Now delete 80