Data Structures and Algorithm

Moazzam Ali Sahi

Lecture # 22

Binary Search Tree

Last Lecture

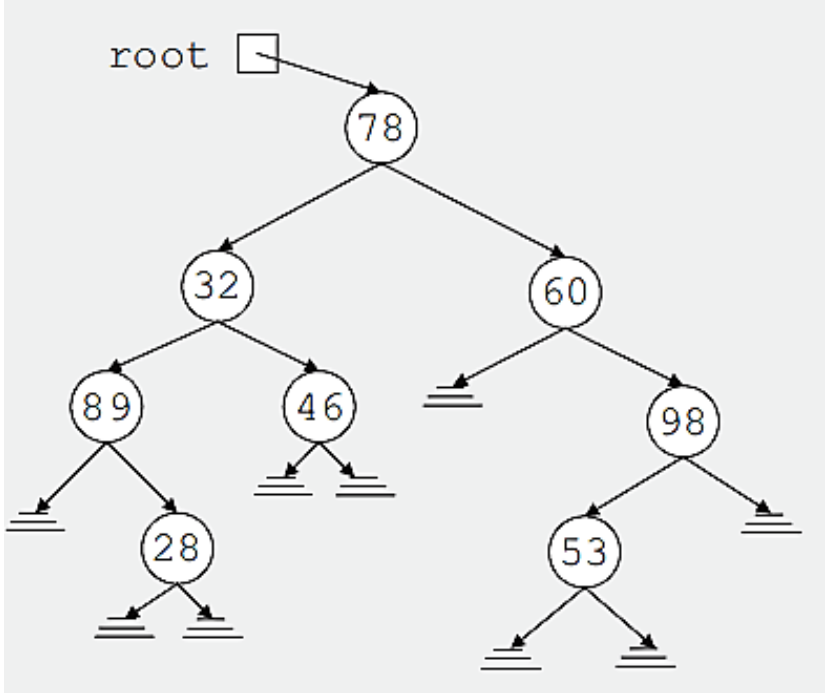
- Expression Trees
- Traversal in Trees

This Lecture

- Binary Search Tree

Binary Search Tree (BST)

- A special type of binary tree



Binary Search Tree (BST)

Definition:

A binary search tree, T, is either empty or the following is true:

i. T has a special node called the root node.

ii. T has two sets of nodes, LT and RT, called the left subtree and right subtree of T, respectively.

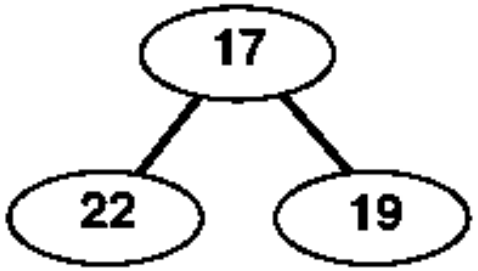
iii. The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree.

iv. L_T and R_T are binary search trees.

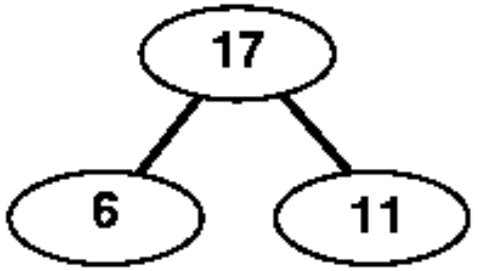
BINARY SEARCH TREE PROPERTIES

- A binary search tree (BST), also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in order.
- In a BST, all nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- Due to its efficiency in searching elements, BSTs are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

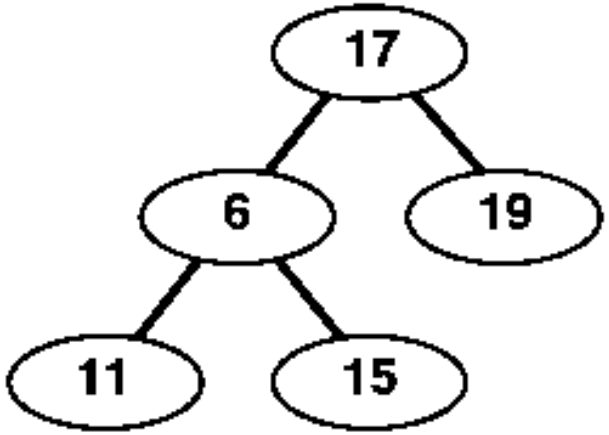
VALID BINARY SEARCH TREES



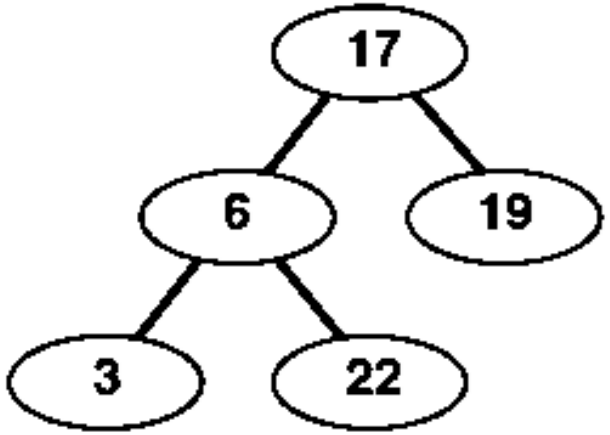
(a)



(b)



(c)



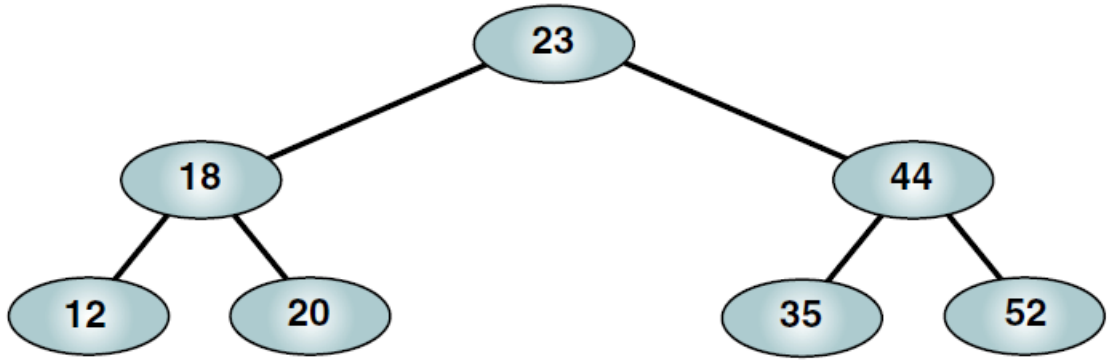
(d)

OPERATIONS OF BINARY SEARCH TREE

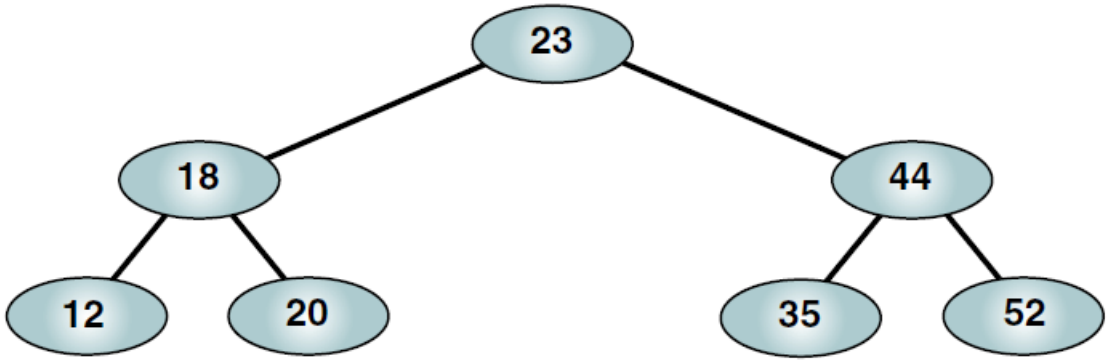
1) Traversal
2) Search the binary search tree for a particular item.
3) Insert an item in the binary search tree.
4) Delete an item from the binary search tree.
5) Find the height of the binary search tree.
6) Find the number of nodes in the binary search tree.
7) Find the number of leaves in the binary search tree.
8) Traverse the binary search tree.
9) Copy the binary search tree.

Traversal in a BST

- The binary tree traversal operations are identical to the ones we had already studied.
- Our interest here is not in the operation itself but rather in the results it produces.
- Let's begin by traversing a tree



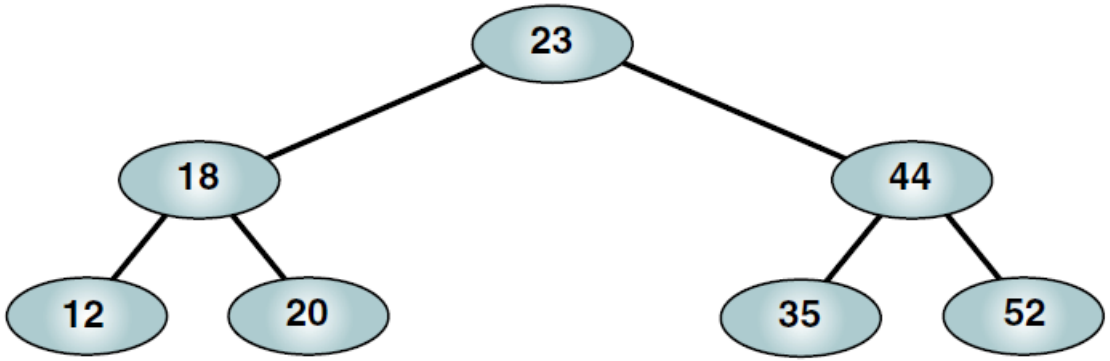
Pre-Order Traversal in a BST



- If we traverse the tree using a preorder traversal, we get the results shown below.

23 18 12 20 44 35 52

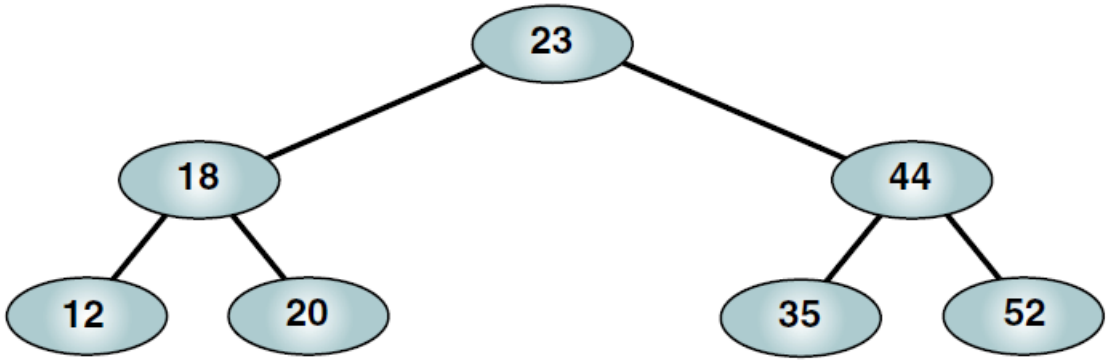
Post-Order Traversal in a BST



- If we traverse the tree using a postorder traversal, we get the results shown below.

12 20 18 35 52 44 23

In-Order Traversal in a BST



- If we traverse the tree using a Inorder traversal, we get the results shown below.

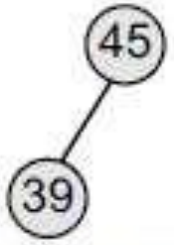
12 18 20 23 35 44 52

The inorder traversal of a binary search tree produces a sequenced list.

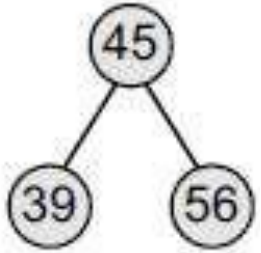
Creating a Binary Search from Given Values

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

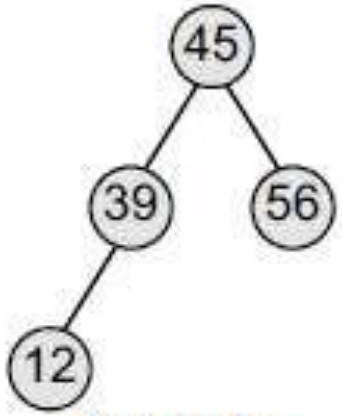
(Step 1)



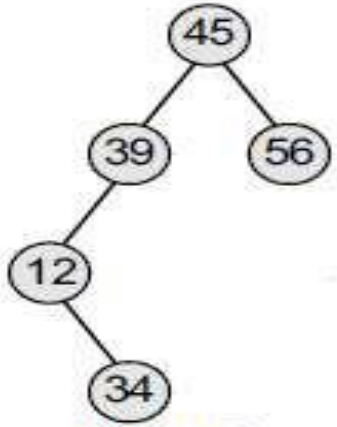
(Step 2)



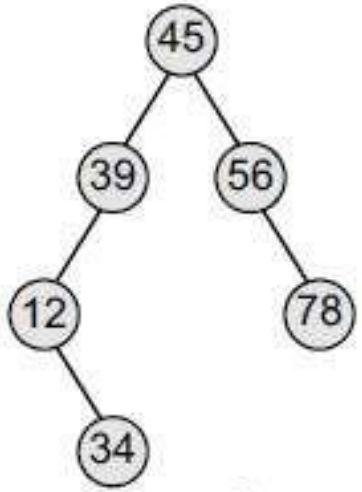
(Step 3)



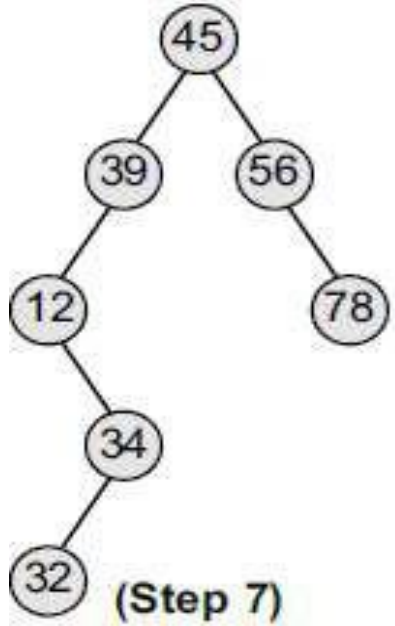
(Step 4)



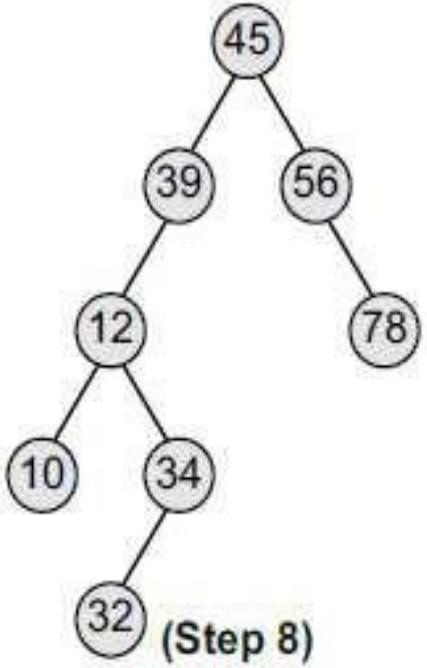
(Step 5)



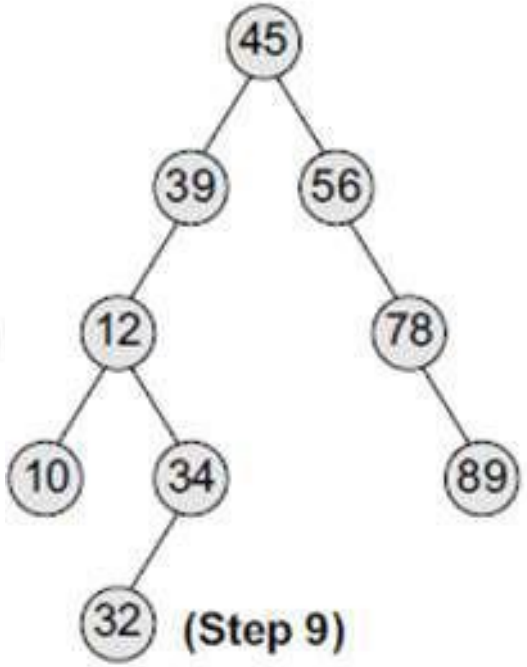
(Step 6)



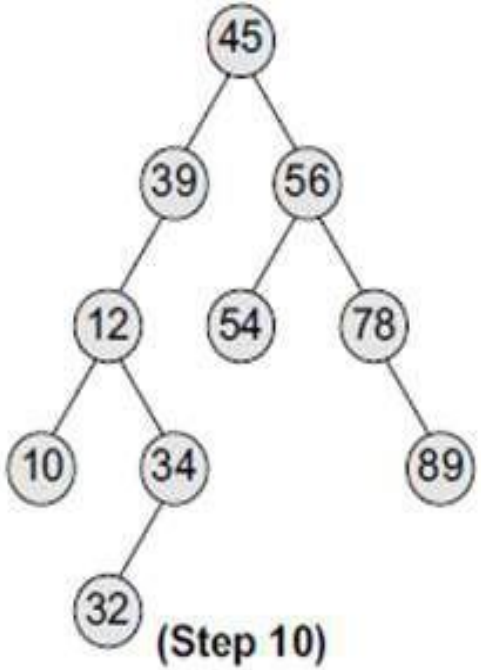
(Step 7)



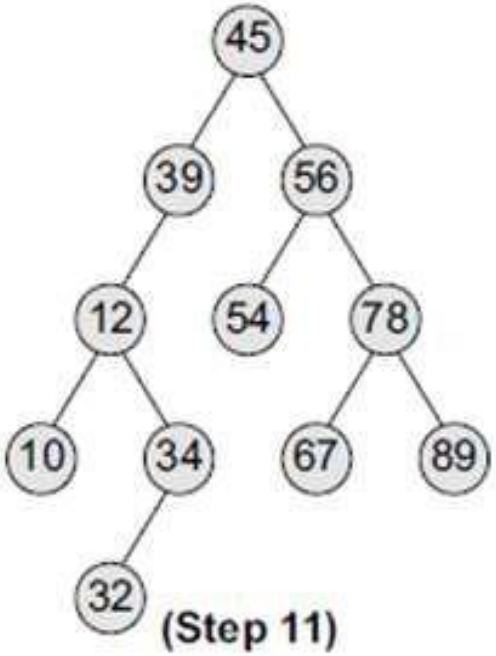
(Step 8)



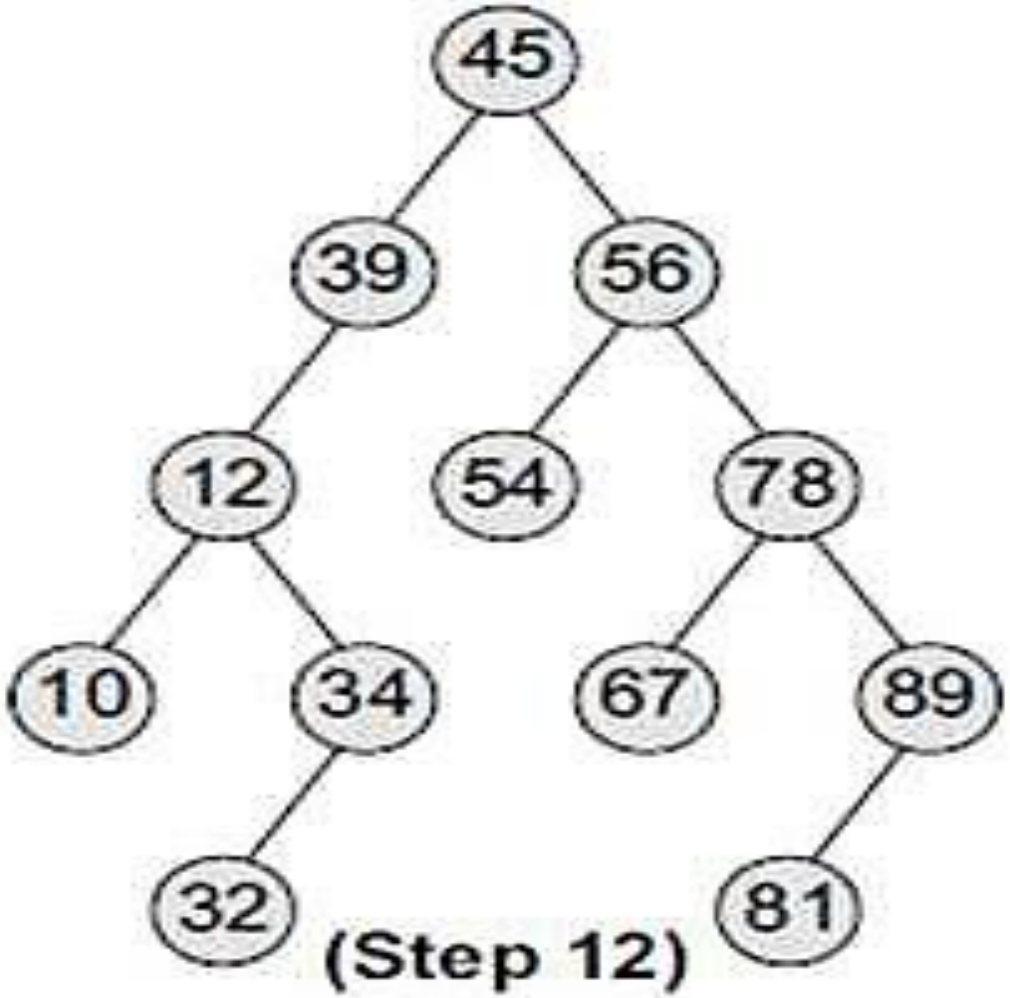
(Step 9)



(Step 10)



(Step 11)



(Step 12)

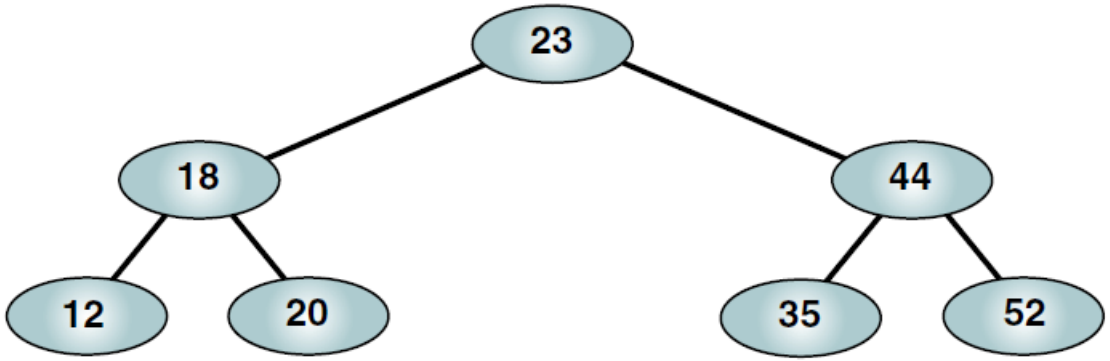
Searching for a Value in a BST

- The search function is used to find whether a given value is present in the tree or not.
- The function first checks if the BST is empty. If it is, then the value we are searching for is not present in the tree, and 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 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 node, it should be recursively called on the right child node.

Searching for a Value in a BST

- Now we study three search algorithms:
  - Find the smallest node
  - Find the largest node
  - Find a requested node (BST search).

Searching [Find the Smallest Node]



- Note that the node with the smallest value (12) is the far-left leaf node in the tree.
- To find smallest node operation, therefore, simply follows the left branches until we get to a leaf.

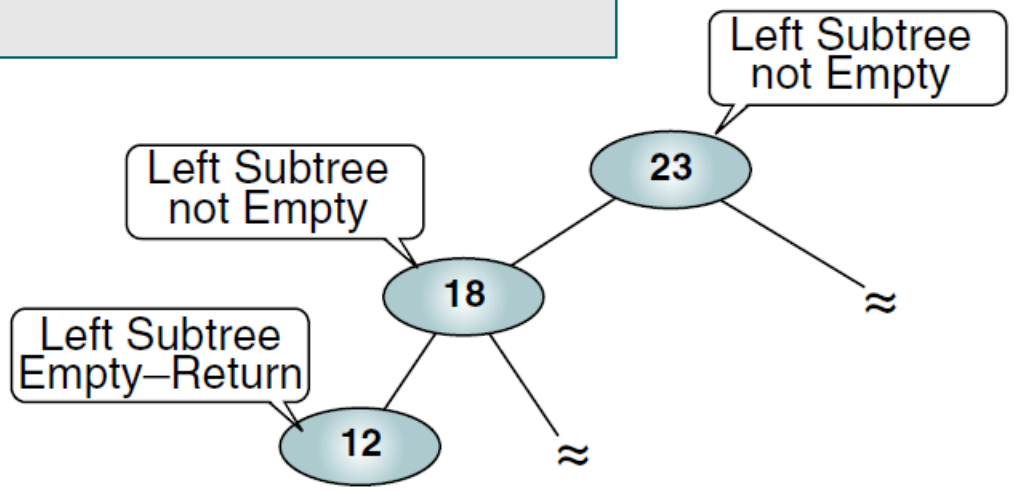
Searching [Find the Smallest Node - -ALGORITHM]

```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
 1 if (left subtree empty)
   1   return (root)
 2 end if
 3 return findSmallestBST (left subtree)
end findSmallestBST
```

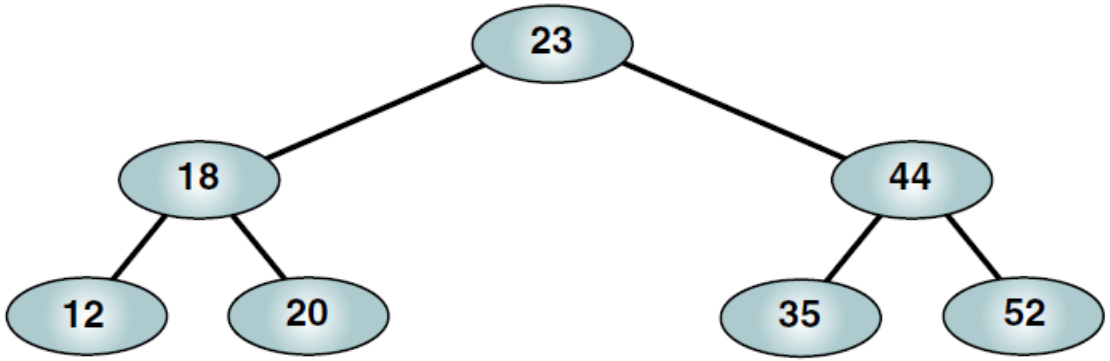
Can you write the code for it?

Searching [Find the Smallest Node - -WORKING]

```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
 1  if (left subtree empty)
    1   return (root)
 2  end if
 3  return findSmallestBST (left subtree)
end findSmallestBST
```



Searching [Find the Largest Node]



- Note that the node with the smallest value (52) is the far-right leaf node in the tree.
- To find largest node operation, therefore, simply follows the right branches until we get to a leaf.

Searching [Find the Largest Node - -ALGORITHM]

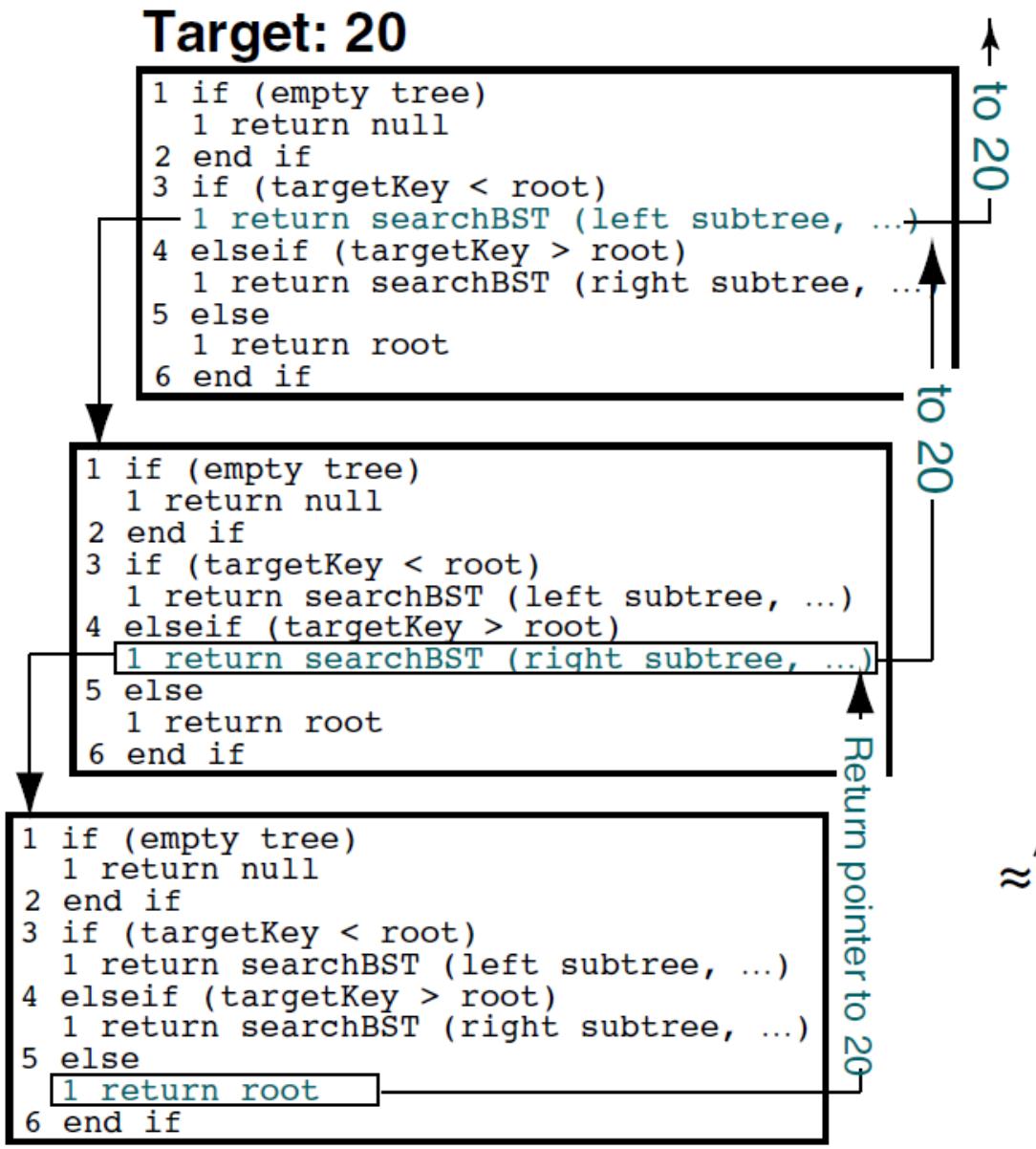
```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
 1 if (left subtree empty)
   1   return (root)
 2 end if
 3 return findSmallestBST (left subtree)
end findSmallestBST
```

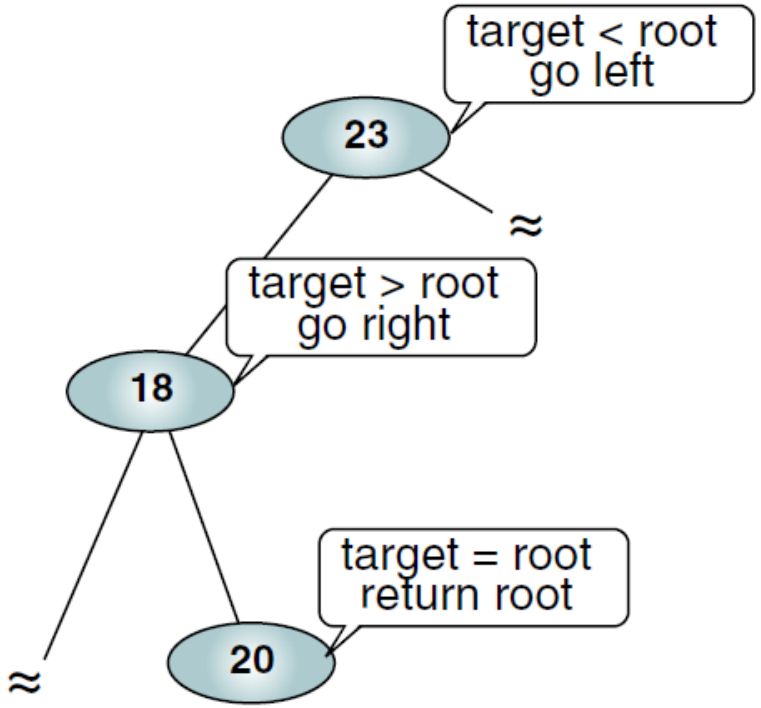
Do it yourself, this algo is for reference

Searching [Find a value]

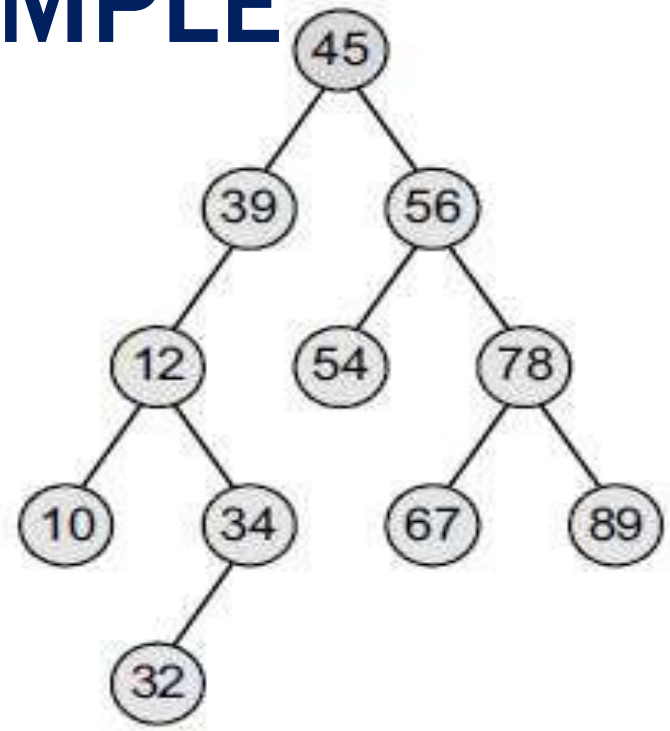
```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Searching [In Action]





EXAMPLE

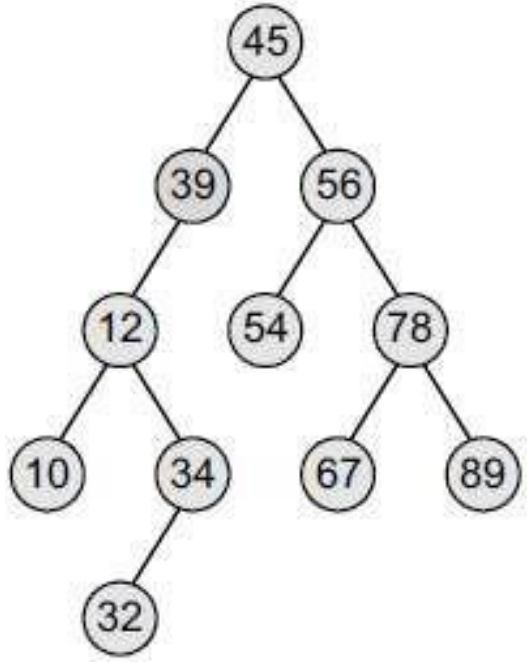


Searching a node with the value 40 in the given binary search tree

VALUE NOT FOUND

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

VALUE FOUND



Algorithm to Inserting a node /Value in BST

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

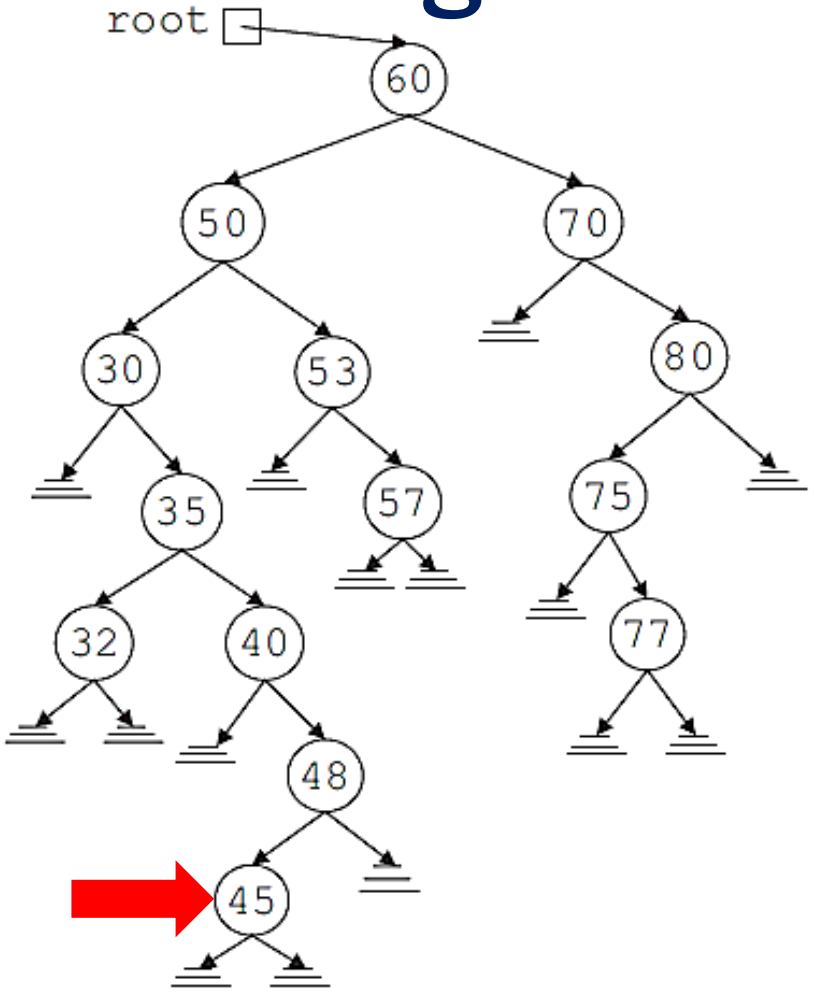
DELETING A NODE IN A BINARY SEARCH TREE

DELETNG A NODE

- Deleting a node in a binary search tree can be done in THREE cases

a) Deleting a node with “no” children's
b) Deleting a node having “one” child
c) Deleting a node having “two” child's

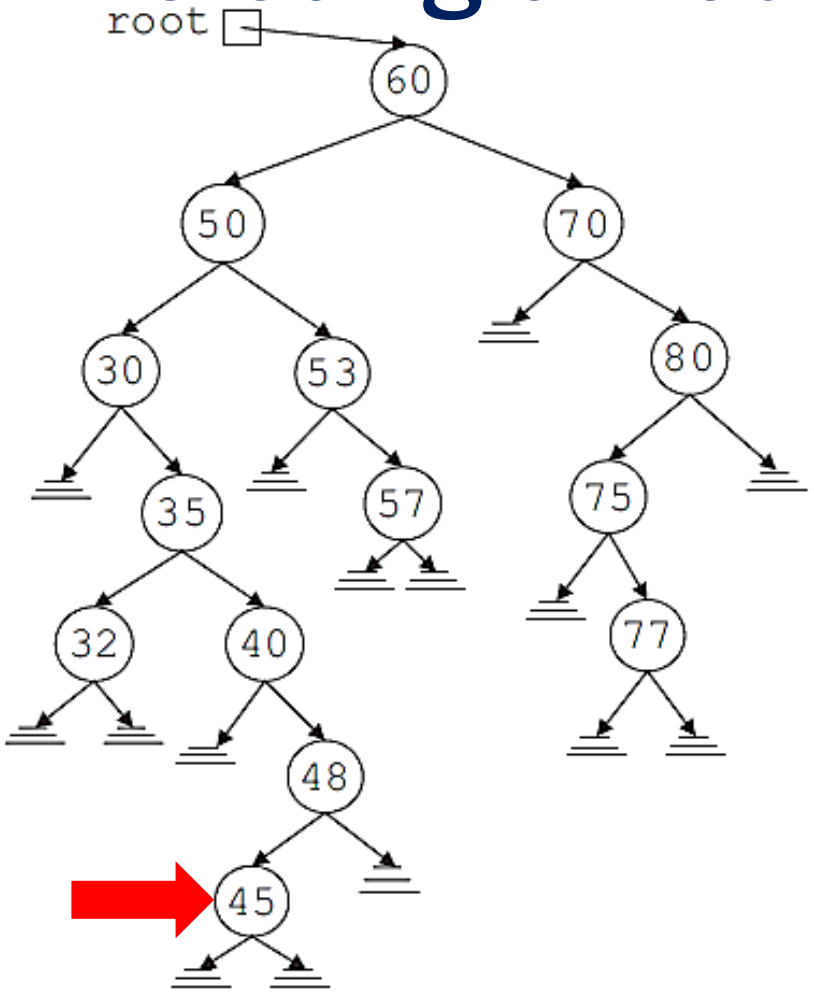
Deleting a Node



Case 1: The node to be deleted has no left and right subtrees; that is, the node to be deleted is a leaf.

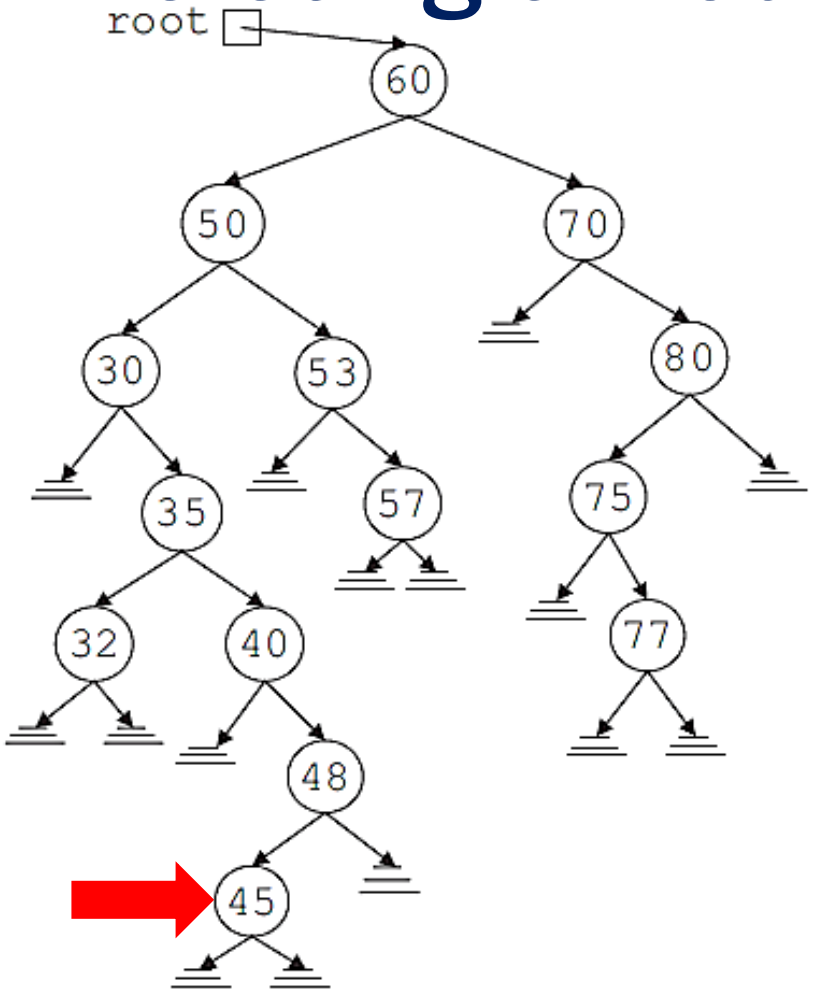
For example, the node with info 45 is a leaf.

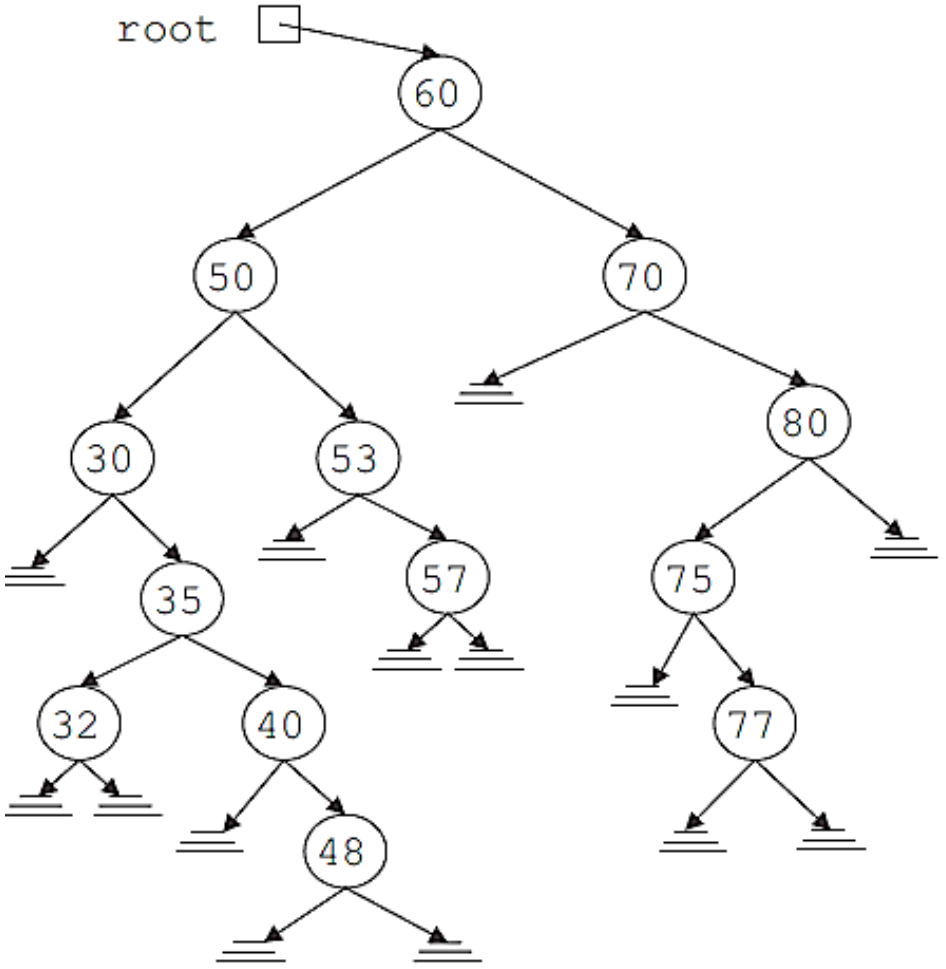
Deleting a Node Case 1



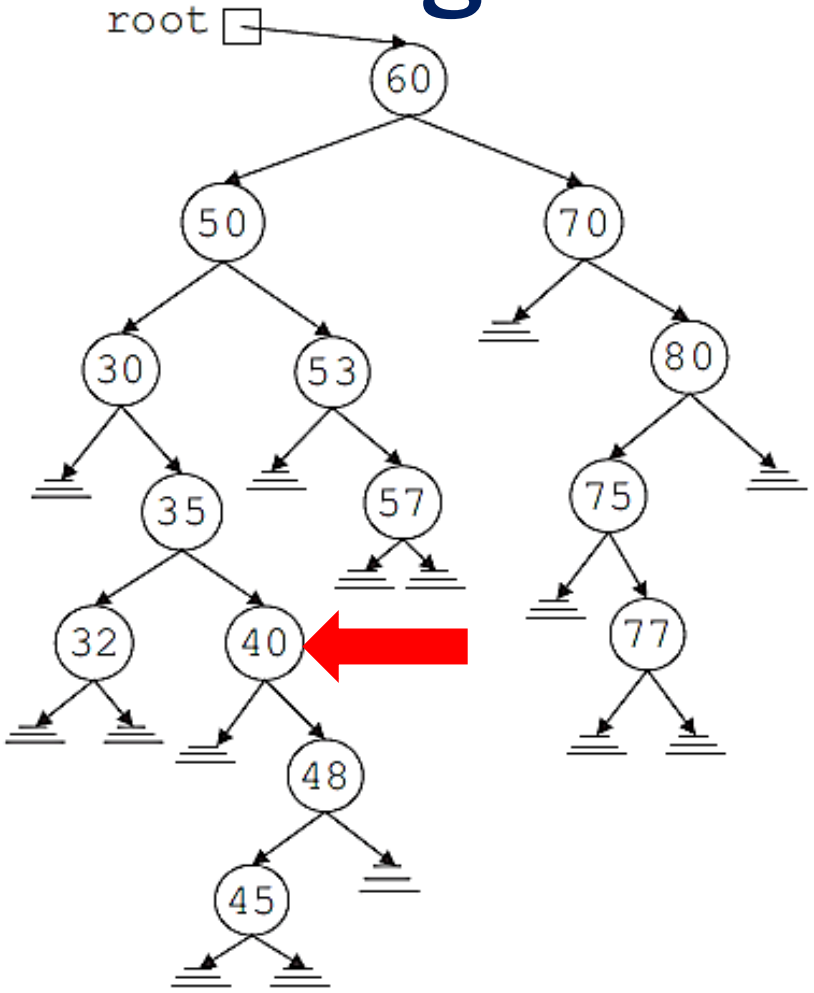
- To delete 45 from the binary search tree.
- We search the binary tree and arrive at the node containing 45.
- Because this node is a leaf and is the left child of its parent, we can simply set the ***left->link*** of the parent node to **NULL** and deallocate the memory occupied by this node.

Deleting a Node Case 1-Resolved



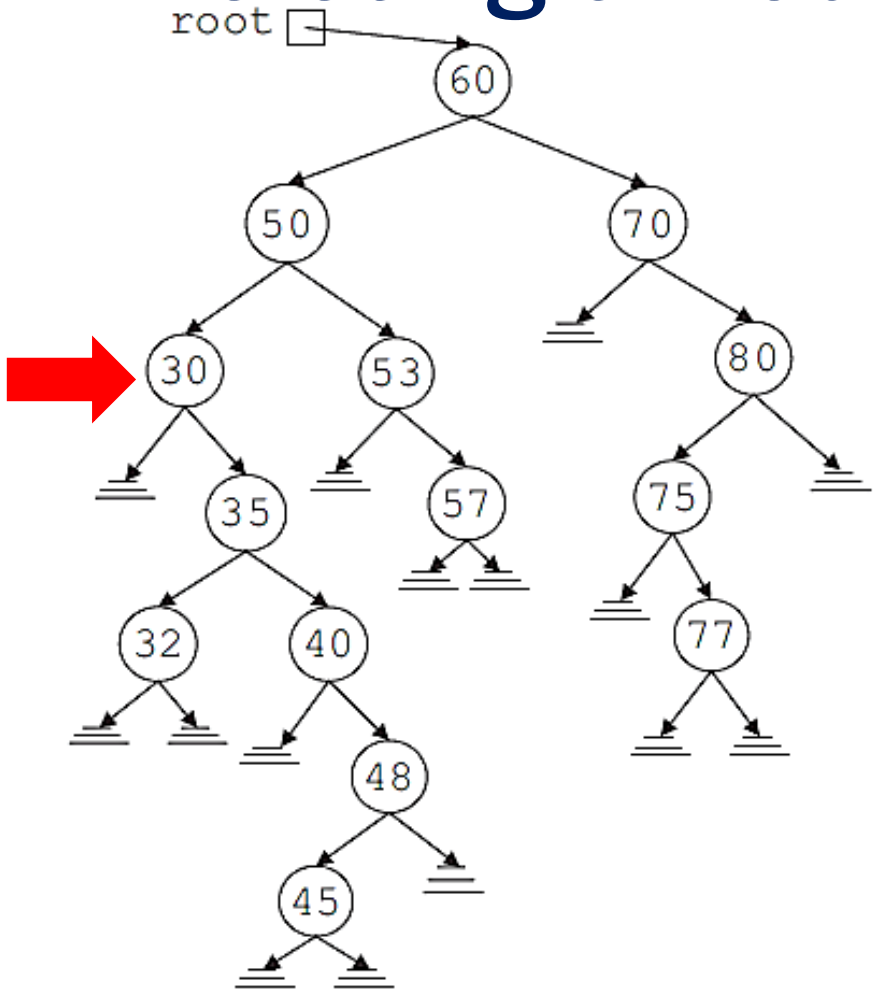


Deleting a Node



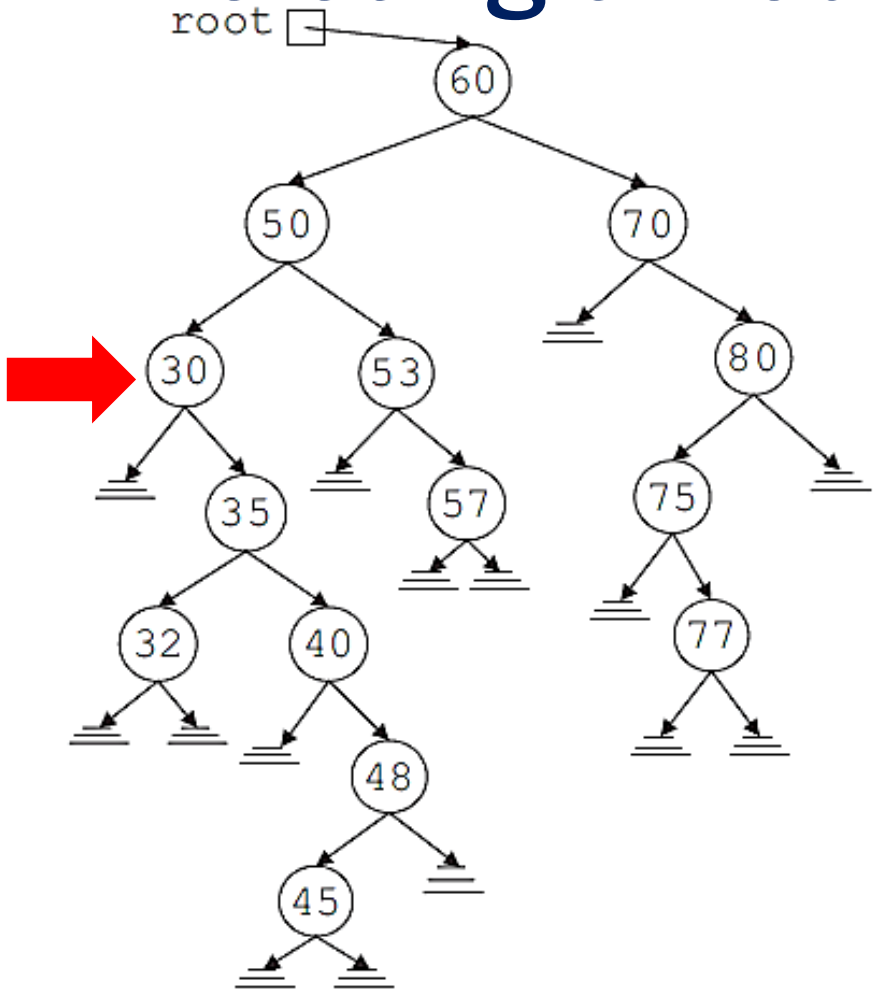
Case 2: The node to be deleted has no left subtree; that is, the left subtree is empty, but it has a nonempty right subtree.
For example, the left subtree of node with info 40 is empty and its right subtree is nonempty.

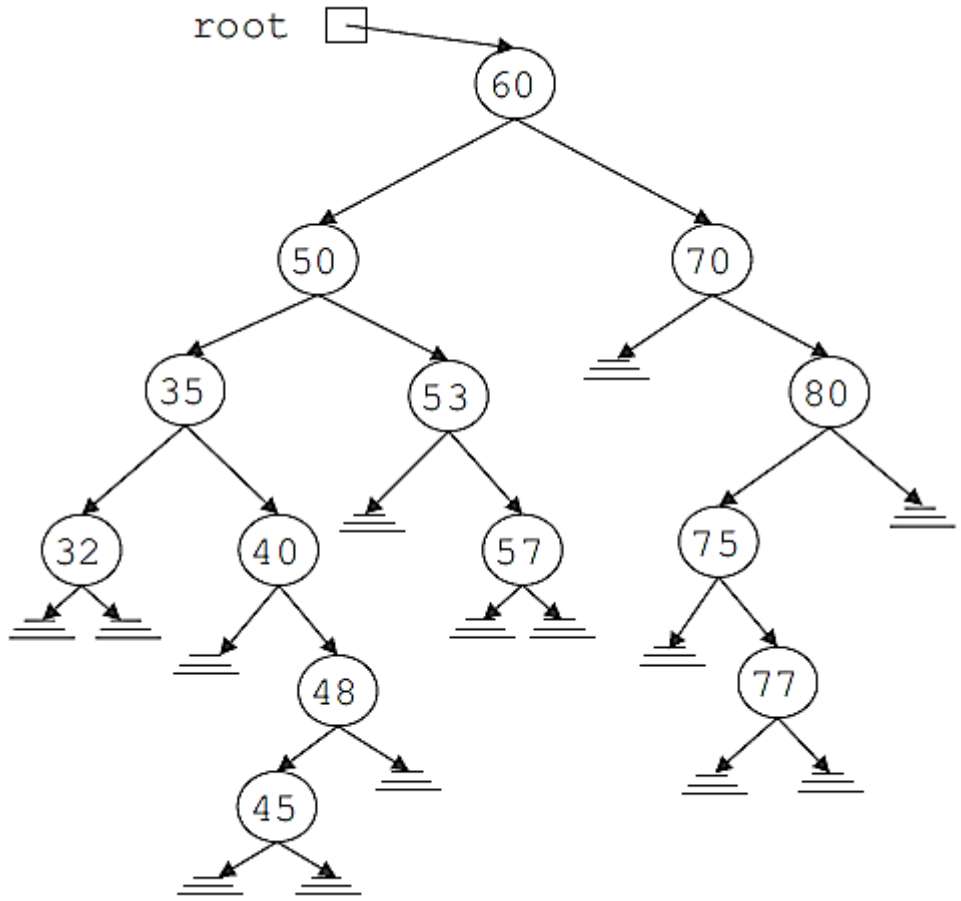
Deleting a Node Case 2



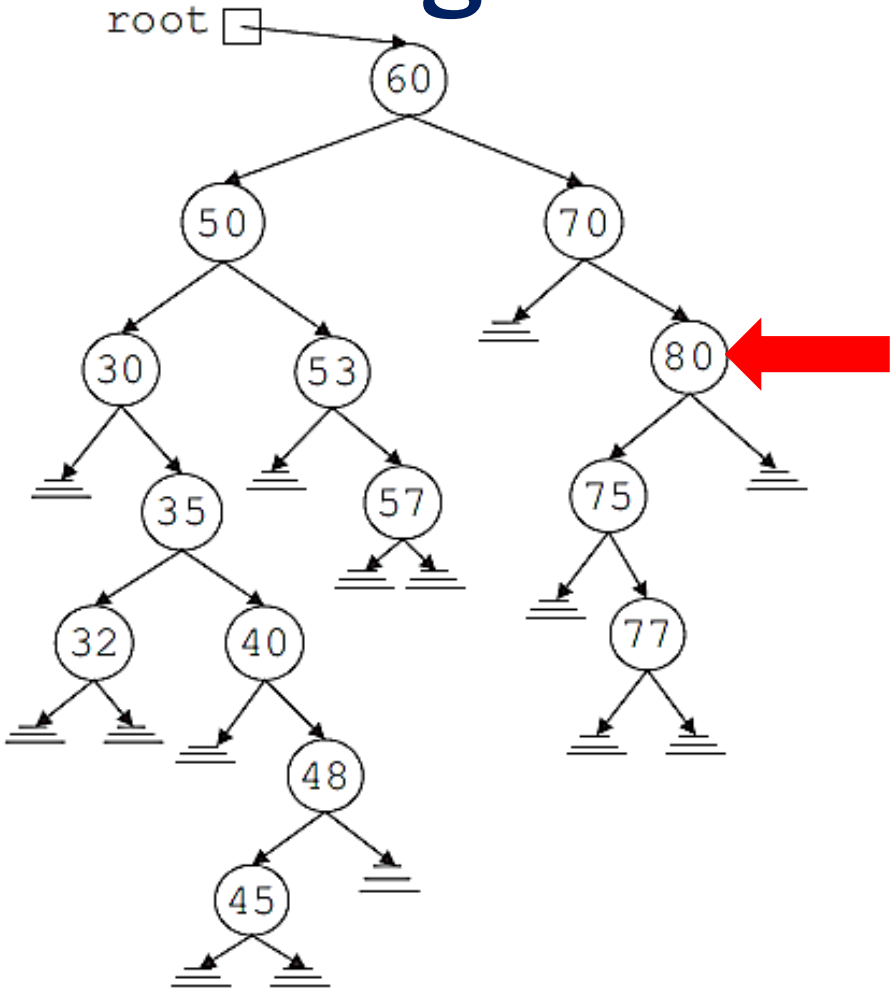
- To delete 30 from the binary search
- The node to be deleted has no left subtree. Because 30 is the left child of its parent node,
- Make the ***left->link*** of the parent node point to the right child of 30 and deallocate the memory occupied by 30

Deleting a Node Case 2-Resolved



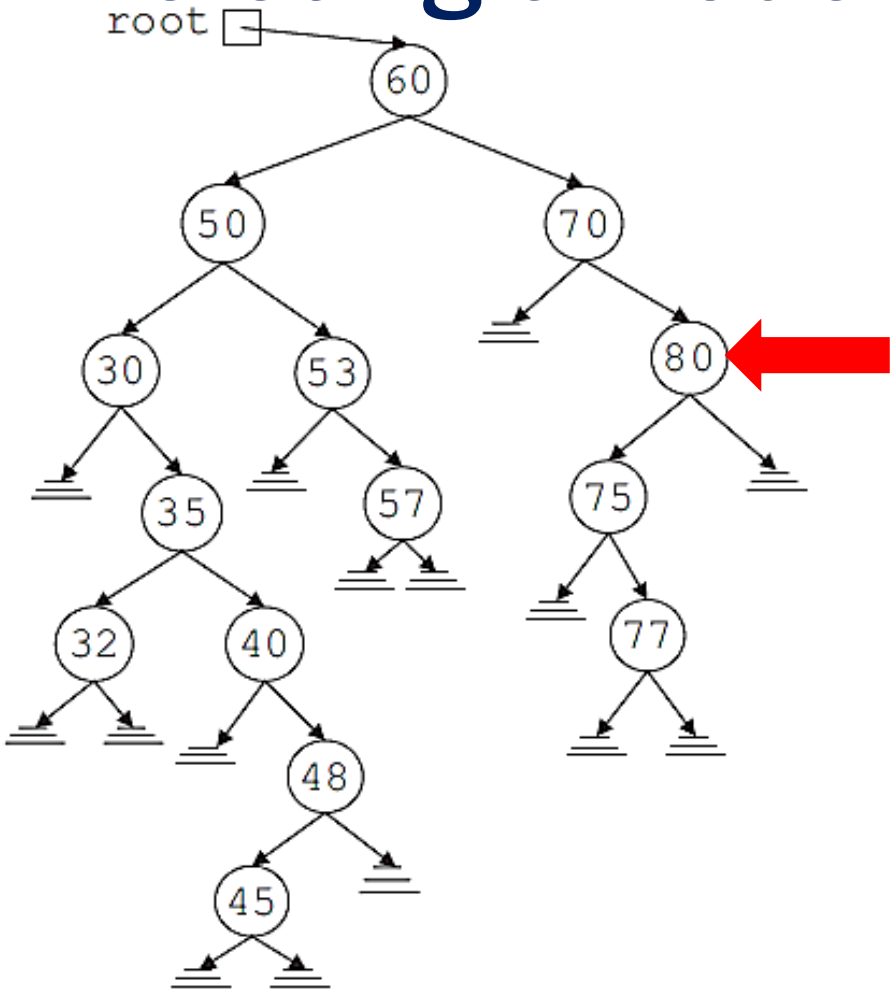


Deleting a Node



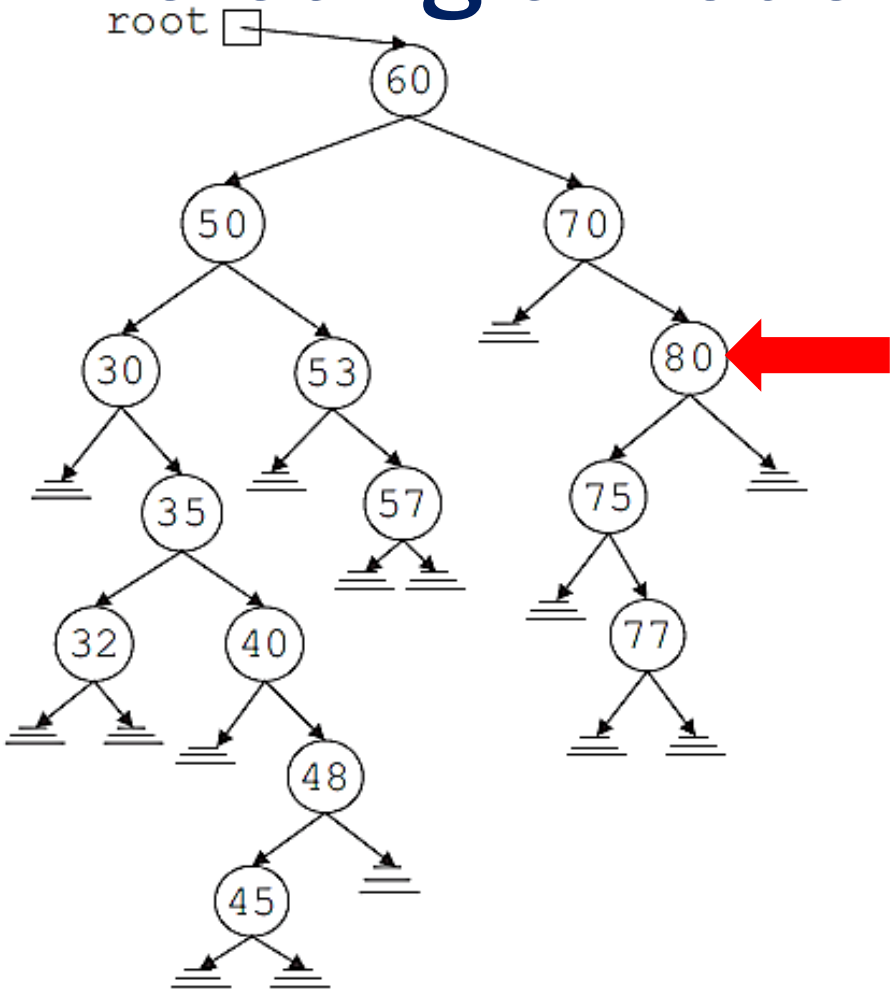
Case 3: The node to be deleted has no right subtree; that is, the right subtree is empty, but it has a nonempty left subtree.
For example, the left subtree of node with info 80 is empty and its right subtree is nonempty.

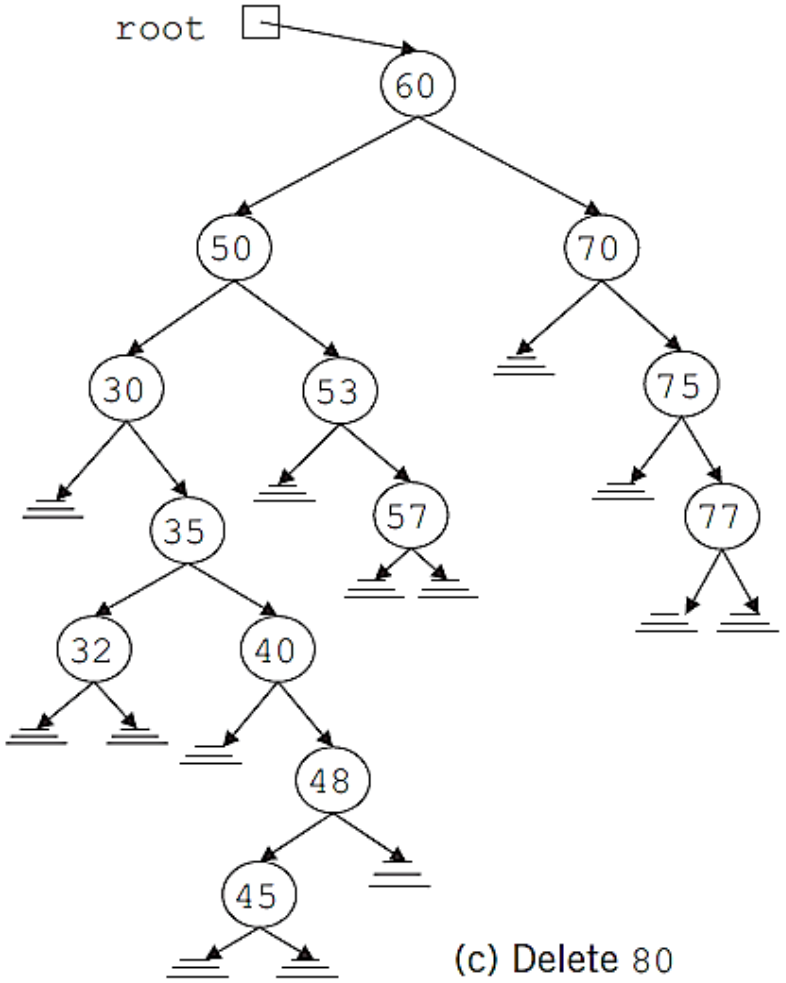
Deleting a Node Case 3



- To delete 80 from the binary search tree.
- The node containing 80 has no right child and is the right child of its parent.
- Thus, we make the ***right->link*** of the parent of 80—that is, 70—point to the left child of 80.

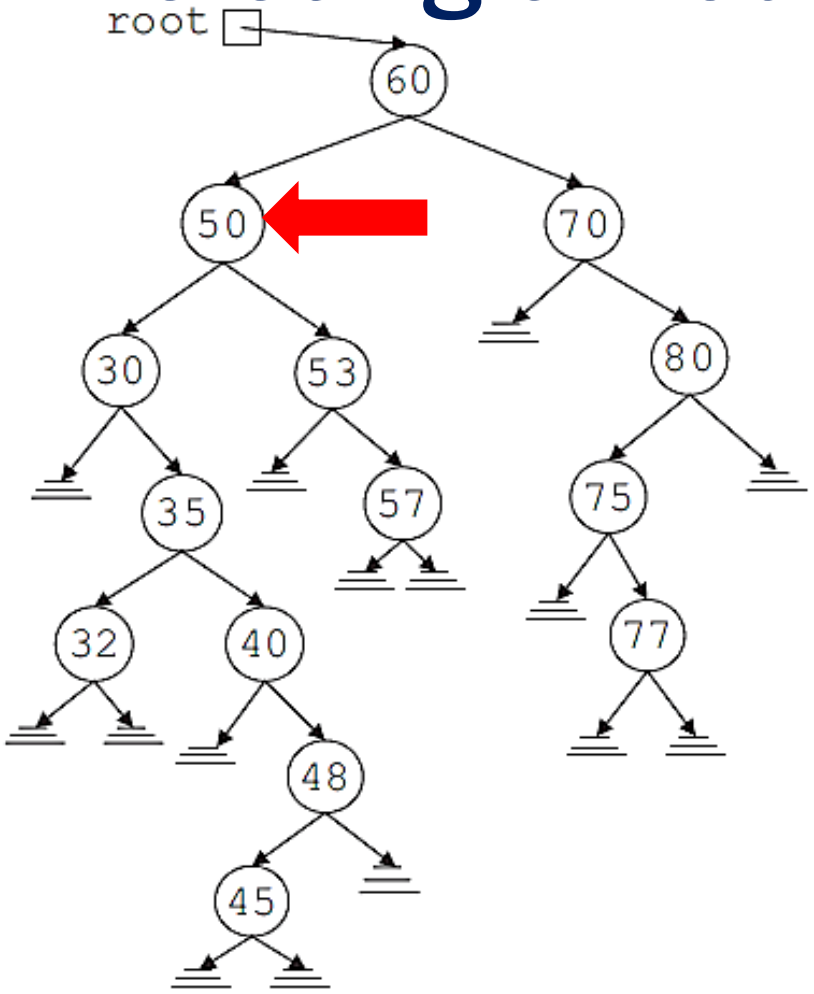
Deleting a Node Case 3-Resolved





(c) Delete 80

Deleting a Node Case 4



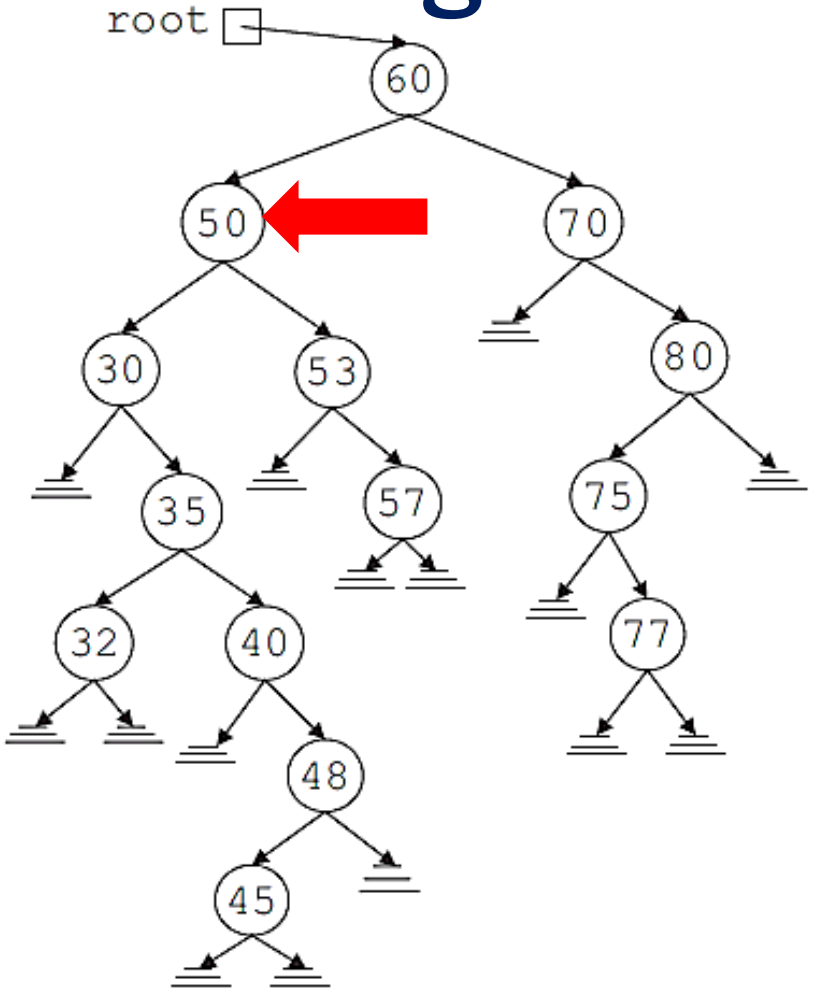
Case 4: The node to be deleted has nonempty left and right subtrees.
For example, the left and the right subtrees of node with info 50 are nonempty.

This can be done in one of two ways:

1) Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data or
2) Find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.

Case 4-Resolved

Deleting a Node



- The node with info 50 has a nonempty left subtree and a nonempty right subtree.
  - First reduce this case to either Case 2 or Case 3 as follows.
  - To reduce it to Case 3—that is, the node to be deleted has no right subtree.
    - Find the immediate predecessor of 50 in this binary tree, which is 48.
    - This is done by first going to the left child of 50 and then locating the rightmost node of the left subtree of 50.
    - To do so, we follow the ***right->link*** of the nodes.
    - Because the binary search tree is finite, we eventually arrive at a node that has no right subtree.
    - Next, we swap the info in the node to be deleted with the info of its immediate predecessor.
    - In this case, we swap 48 with 50.
    - This reduces to the case wherein the node to be deleted has no right subtree.

Algorithm for Deleting a node in BST

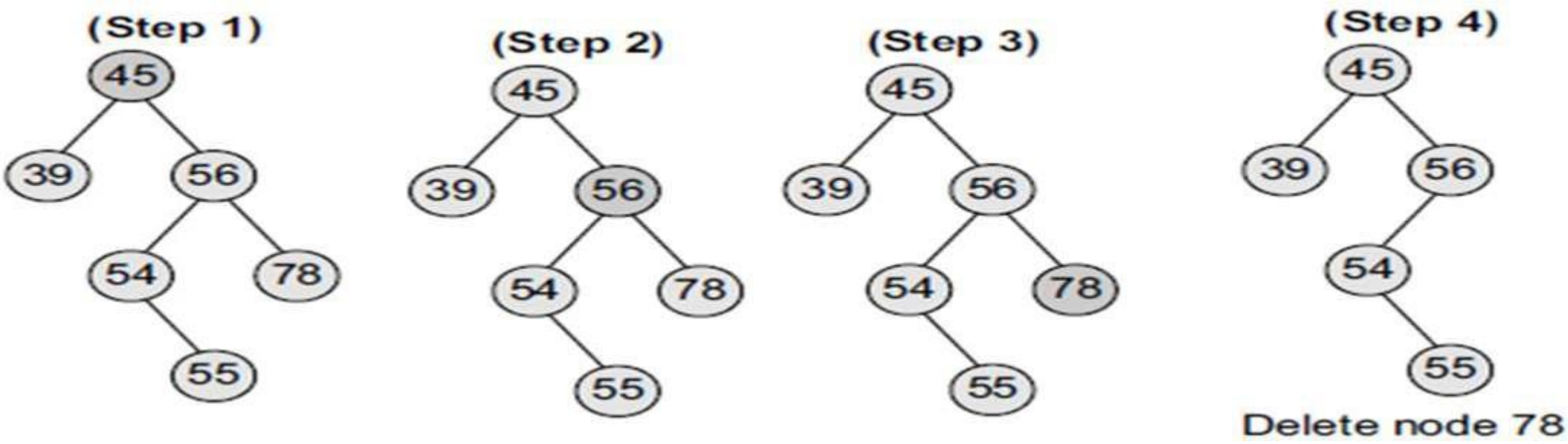
```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
              SET TREE = TREE -> LEFT
          ELSE
              SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Deleting a Value from a BST

- The delete function deletes a node from the binary search tree.
- However, care should be taken that the properties of the BSTs do not get violated and nodes are not lost in the process.
- The deletion of a node involves any of the three cases.
  - Case 1: Deleting a node that has no children.
- Case 2: Deleting a node with one child (either left or right).
  - To handle the deletion, the node's child is set to be the child of the node's parent.
  - Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
  - Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.

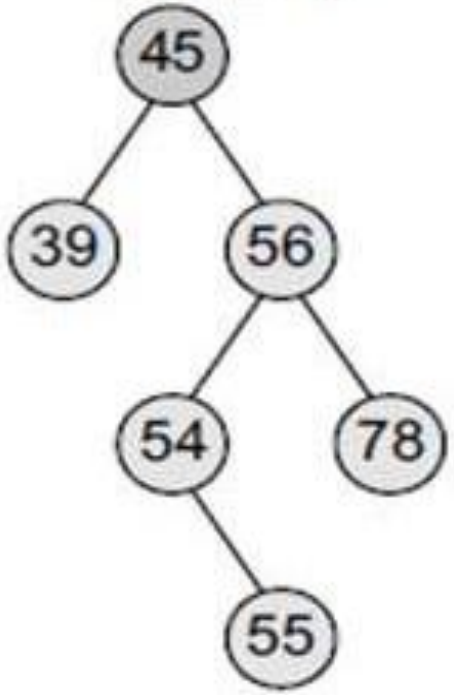
- ***Case 3:*** Deleting a node with two children.
- To handle this case of deletion, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
- The in-order predecessor or the successor can then be deleted using any of the above cases.

Deleting node 78 from the given binary search tree (No child)

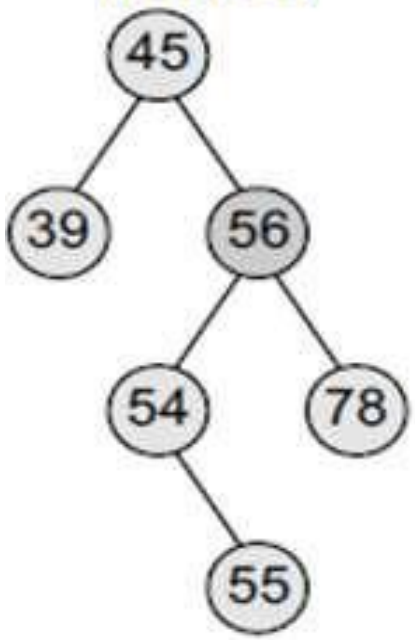


Deleting node 54 from the given binary search tree (one child)

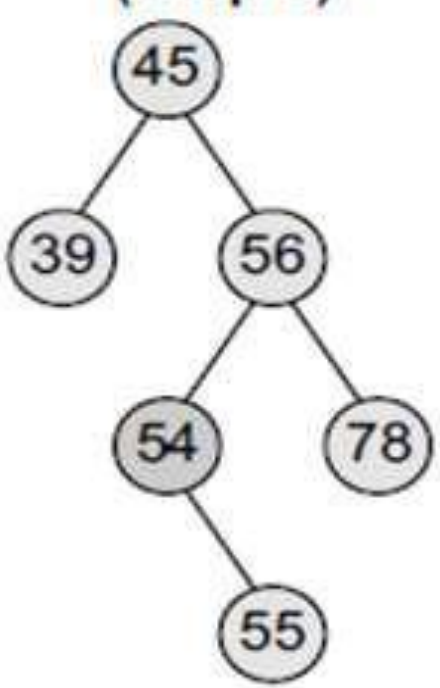
(Step 1)



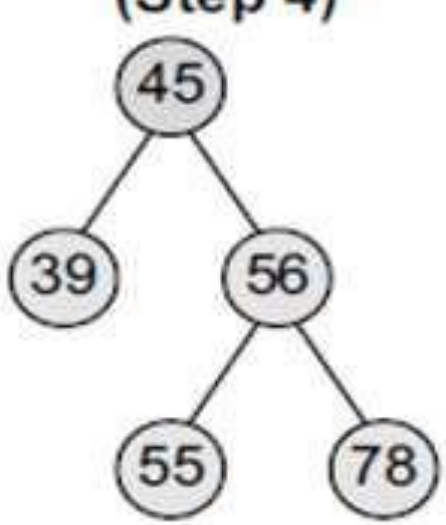
(Step 2)



(Step 3)

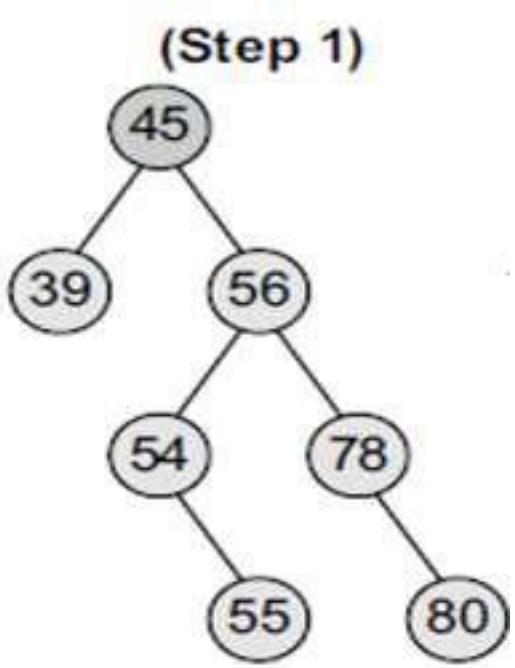


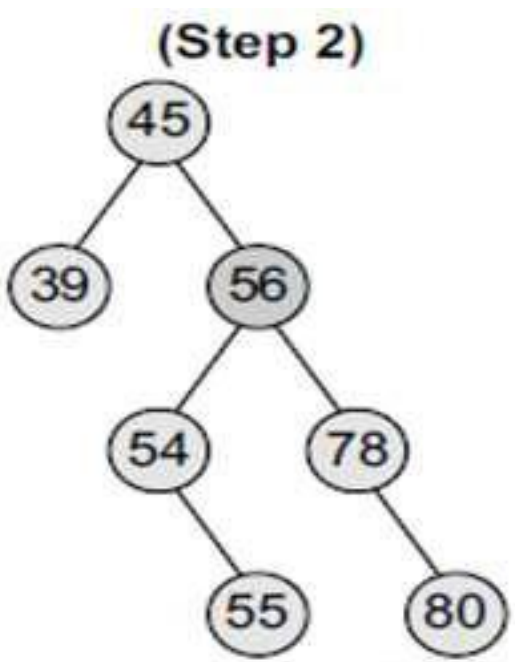
(Step 4)

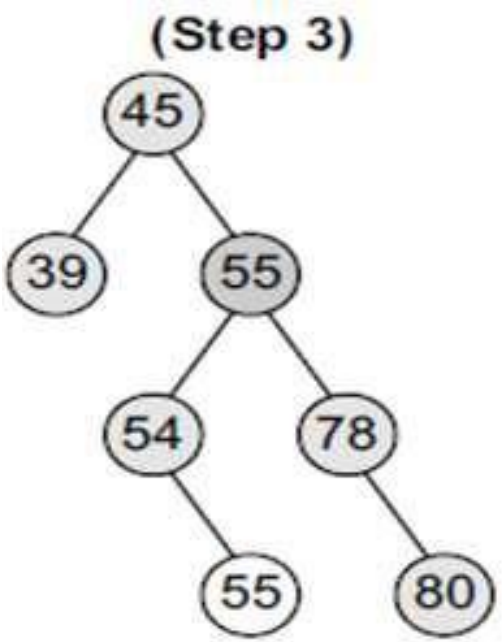


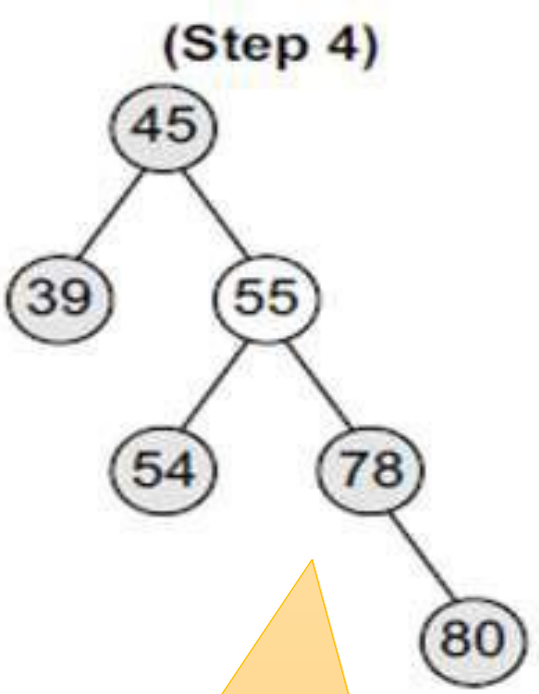
Replace 54 with 55

Deleting node 56 from the given binary search tree (Two child)





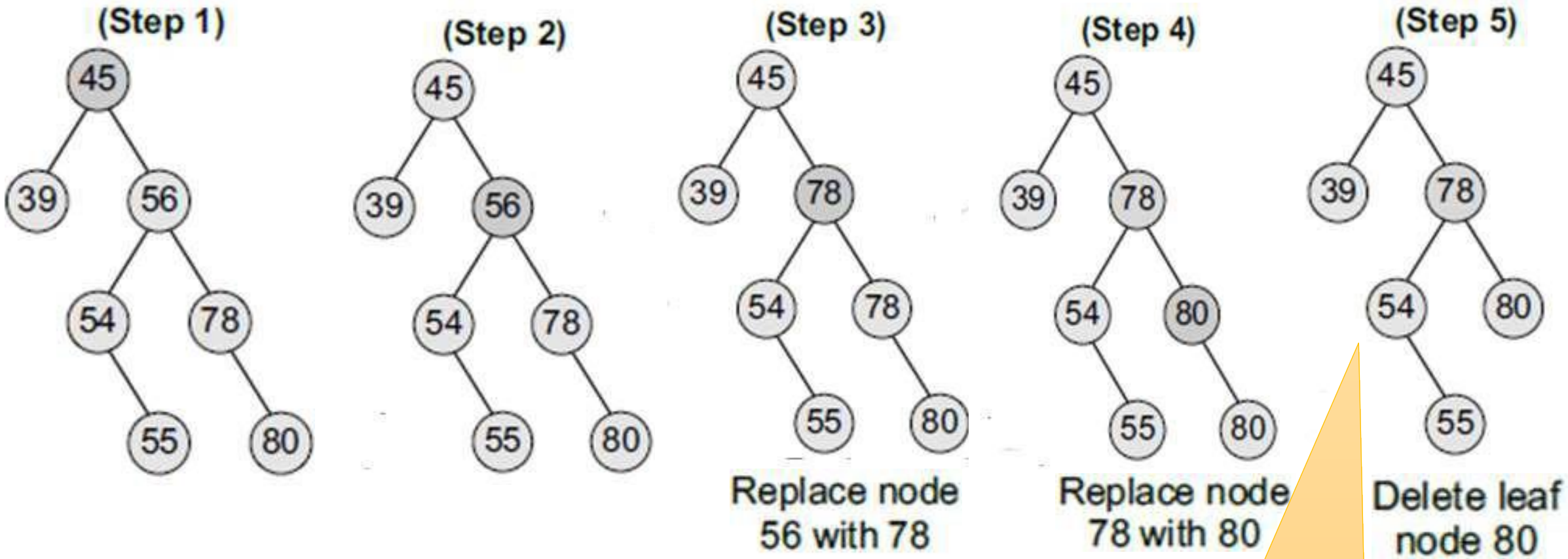




- Can replace the parent with the child
  - Maximum value from the left sub tree

After replacing the Maximum value from the left sub tree

Deleting node 56 from the given binary search tree (Two child)



- Can replace the parent with the child
- Minimum value from the Right Sub Tree

After replacing the Minimum value from the Right sub tree

What else can be done with BSTs

- Height
- Number of nodes
- Number of Internal Nodes
- Number of External Nodes

Delete a Node

```
struct Node *removeNode(struct Node *root, int val)
{
        If the node becomes NULL, it will return NULL
    Two possible ways which can trigger this case
        1. If we send the empty tree. i.e root == NULL
        2. If the given node is not present in the tree.

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

Delete a Node

If root->key < val. val must be present in the right subtree So, call the above remove function with root->right

```
if(root->data < val)
    root->right = removeNode(root->right,val);
```

if root->key > val. val must be present in the left subtree So, call the above function with root->left

```
else if(root->data > val)
    root->left = removeNode(root->left,val);
```

Delete a Node

This part will be executed only if the root->key == val The actual removal starts from here

```
else
 {
    Case 1: Leaf node. Both left and right reference is NULL
    Replace the node with NULL by returning NULL to the calling pointer.
    Free the node
    if(root->left == NULL && root->right == NULL)
    {
        free(root);
        return NULL;
    }
```

Delete a Node

Case 2: Node has right child.

Replace the root node with root->right and free the right node

```
else if(root->left == NULL)
{
    struct Node *temp = root->right;
    free(root);
    return temp;
}
```

Delete a Node

Case 3: Node has left child.

Replace the node with root->left and free the left node

```
else if(root->right == NULL)
{
    struct Node *temp = root->left;
    free(root);
    return temp;
}
```

Delete a Node

Case 4: Node has both left and right children.

Find the min value in the right subtree

Replace node value with min.

And again call the remove function to delete the node which has the min value.

Since we find the min value from the right subtree call the remove function with root->right.

```
        else      {
          int rightMin = getRightMin(root->right);
          root->data = rightMin;
          root->right = removeNode(root->right,rightMin);
      } }
    //return the actual root's address
    return root;   }
```