Binary Search Tree

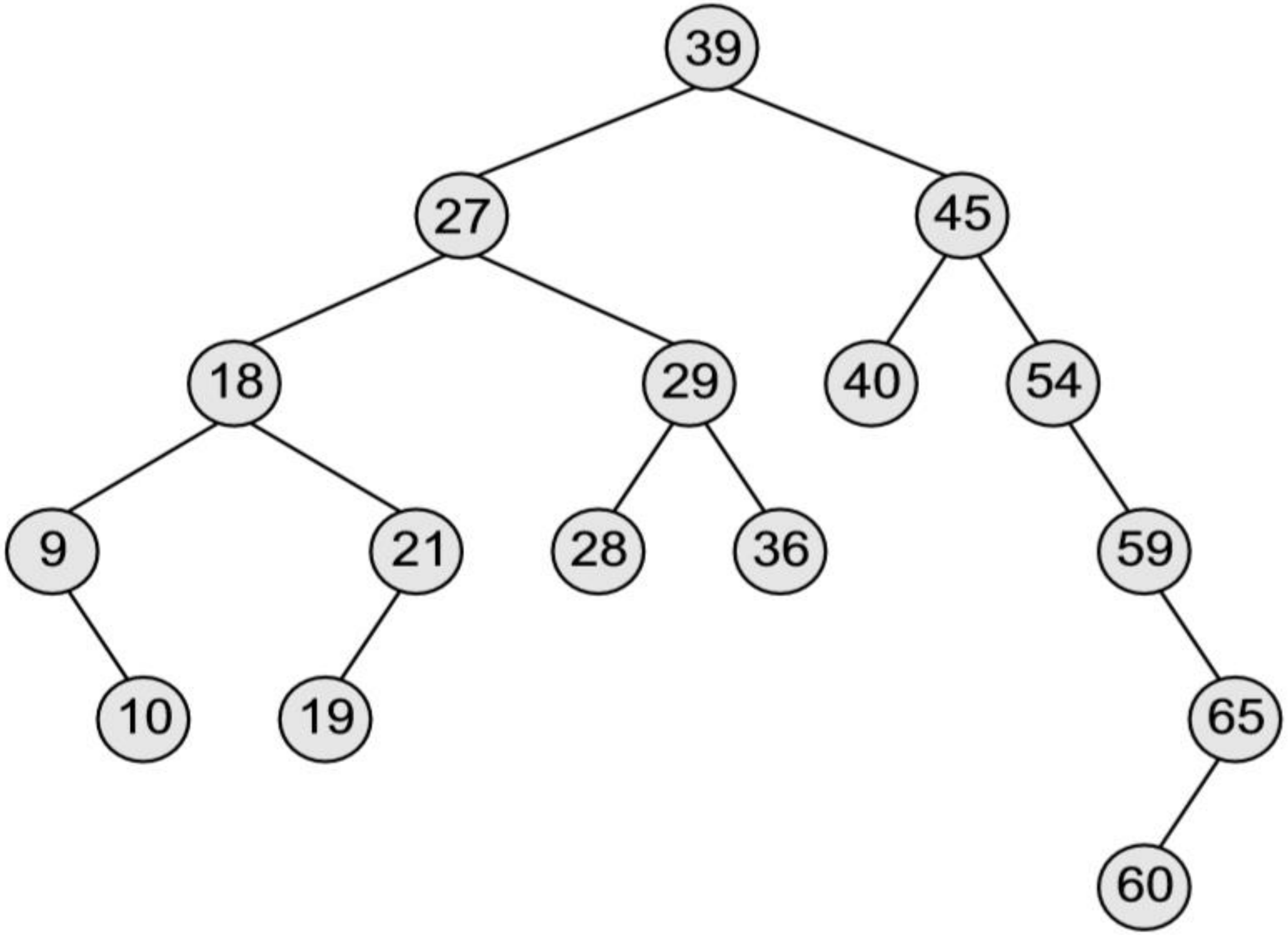
By

Arun Cyril Jose

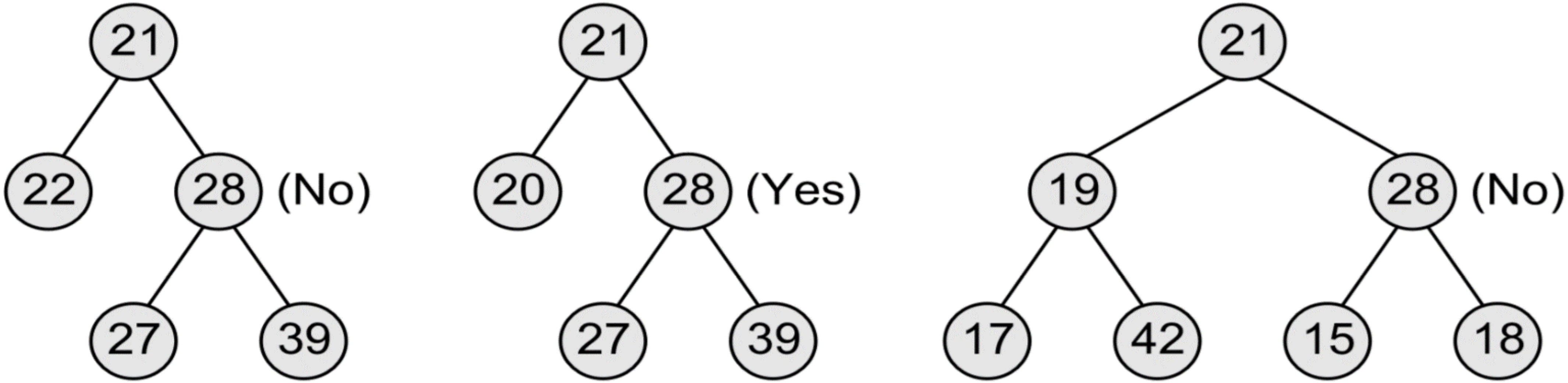
Binary Search Tree

- Variant of binary trees in which the nodes are arranged in an order.
- All the nodes in the **left sub-tree have a value less than that of the root node**.
- All the nodes in the **right sub-tree have a value either equal to or greater than the root node**.
- Same rule is applicable to every sub-tree in the tree.
- May or may not contain duplicate values, depending on its implementation.

Binary Search Tree



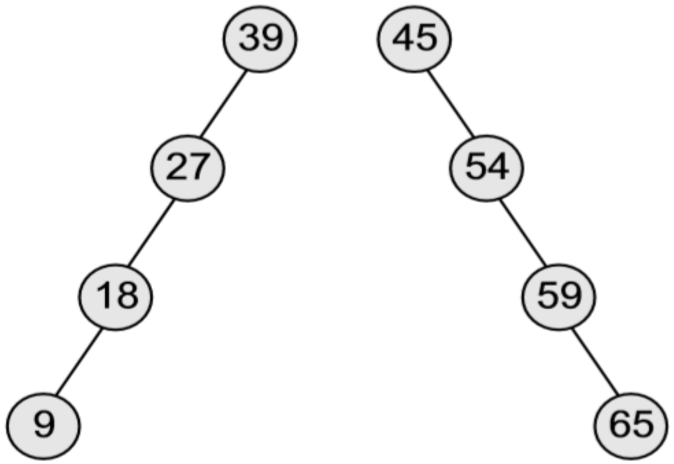
Binary Search Tree



Binary Search Tree

- Time needed to search an element in the tree is greatly reduced.
- Average running time of a search operation is $O(\log_2 n)$.
- Speeds up the insertion and deletion operations.

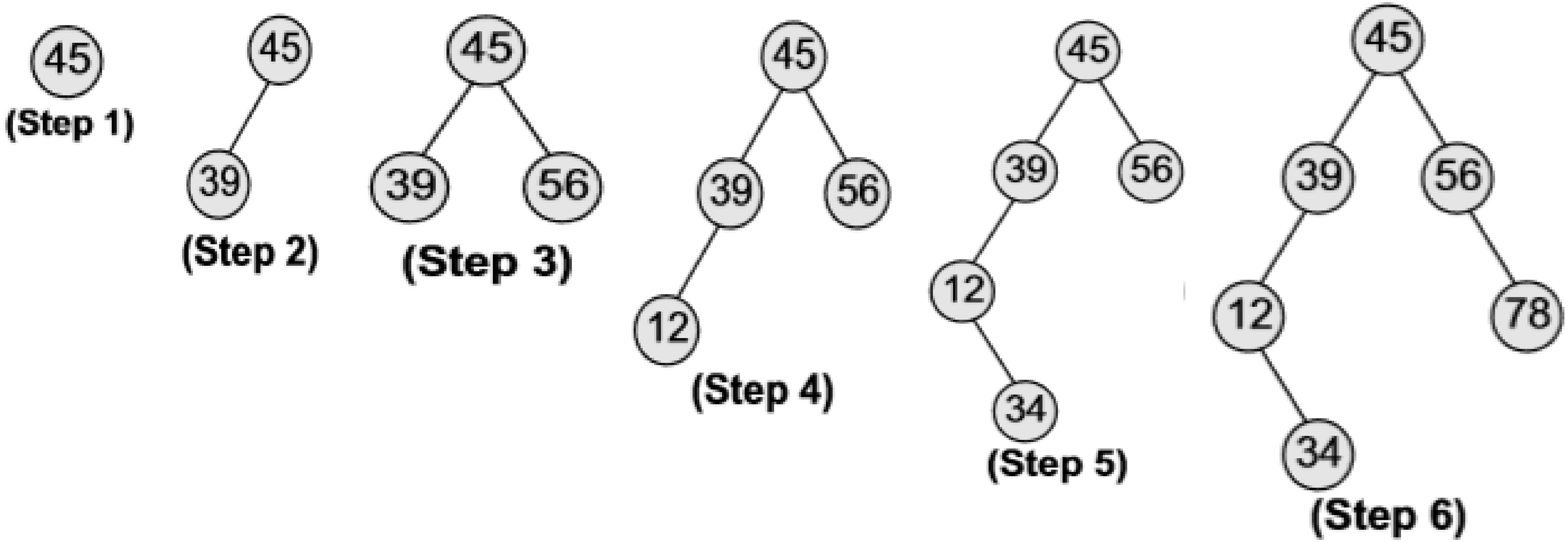
Binary Search Tree



- The left sub-tree of a node N contains values that are less than N’s value.
- The right sub-tree of a node N contains values that are greater than or equal to N’s value.
- Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

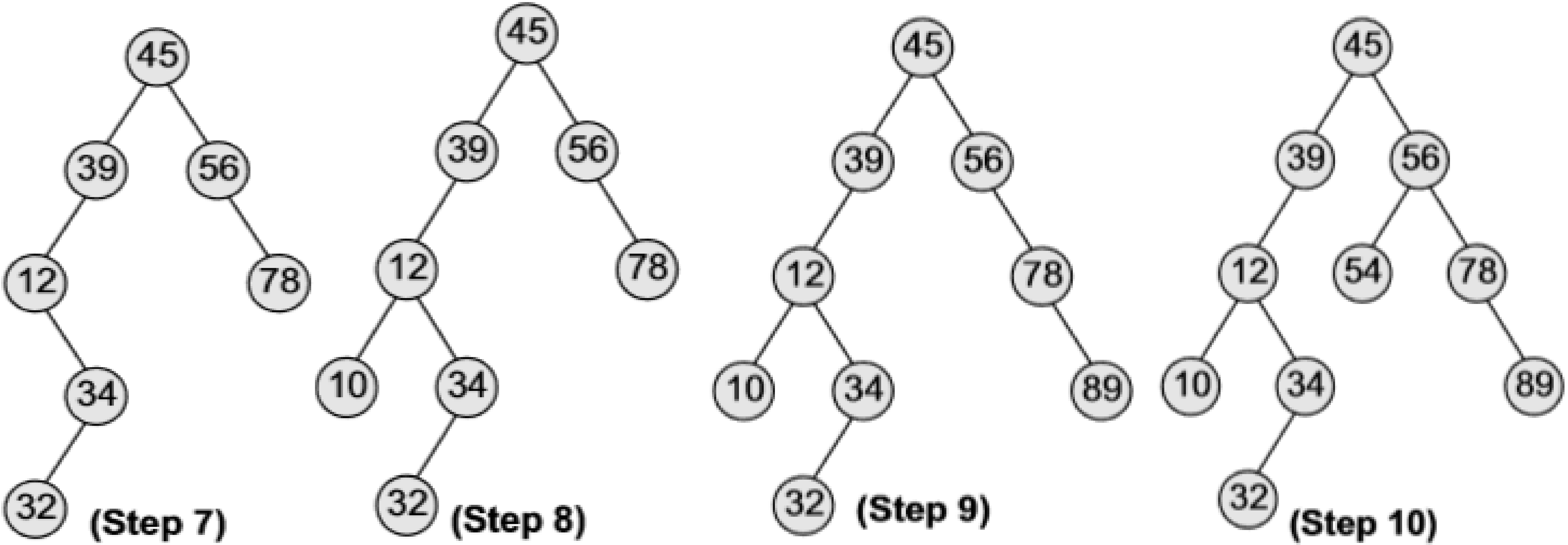
Building a Binary Search Tree

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



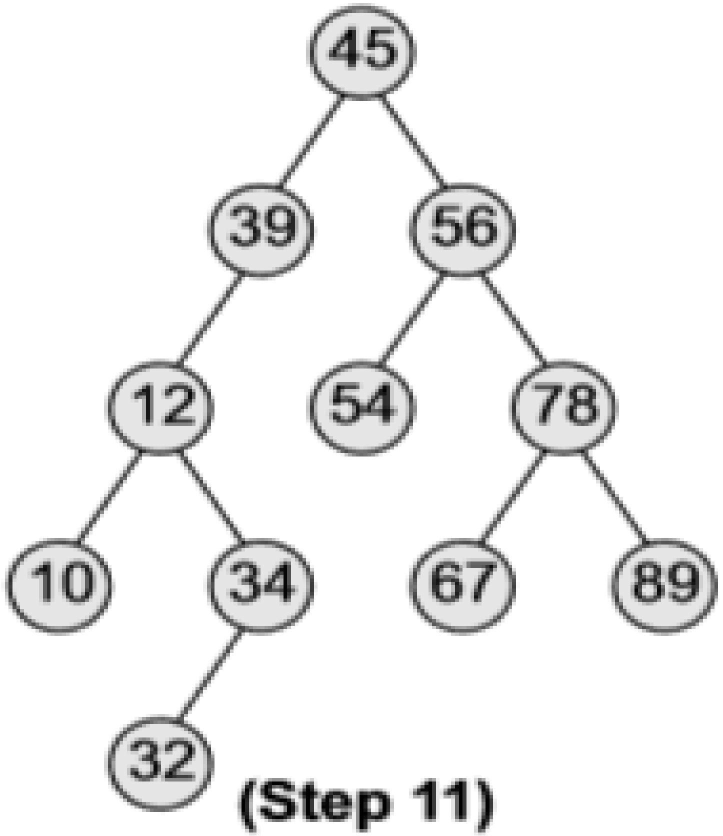
Building a Binary Search Tree

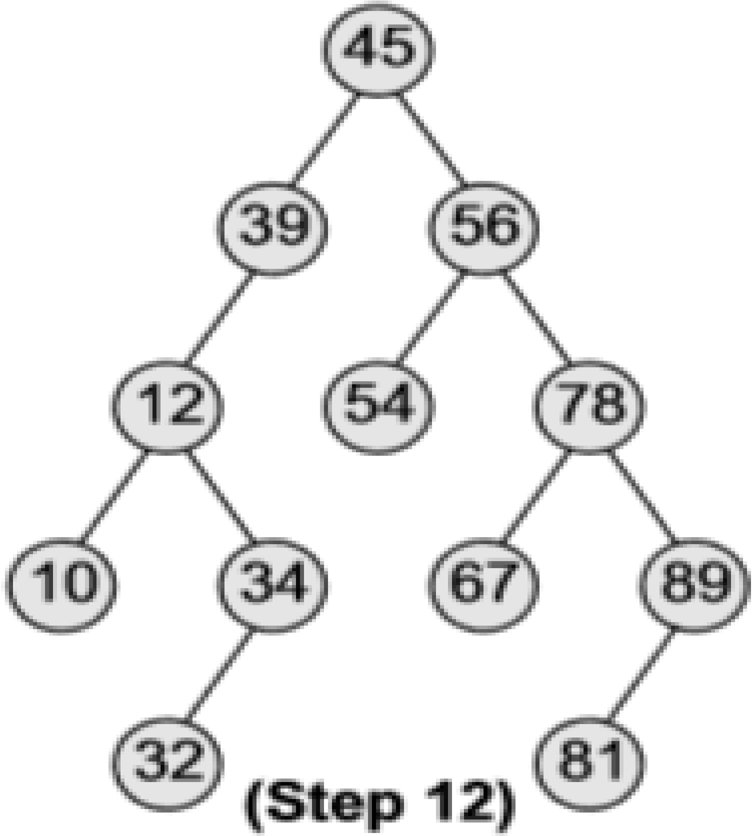
- Data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



Building a Binary Search Tree

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





Searching for a Node in Binary Search Tree

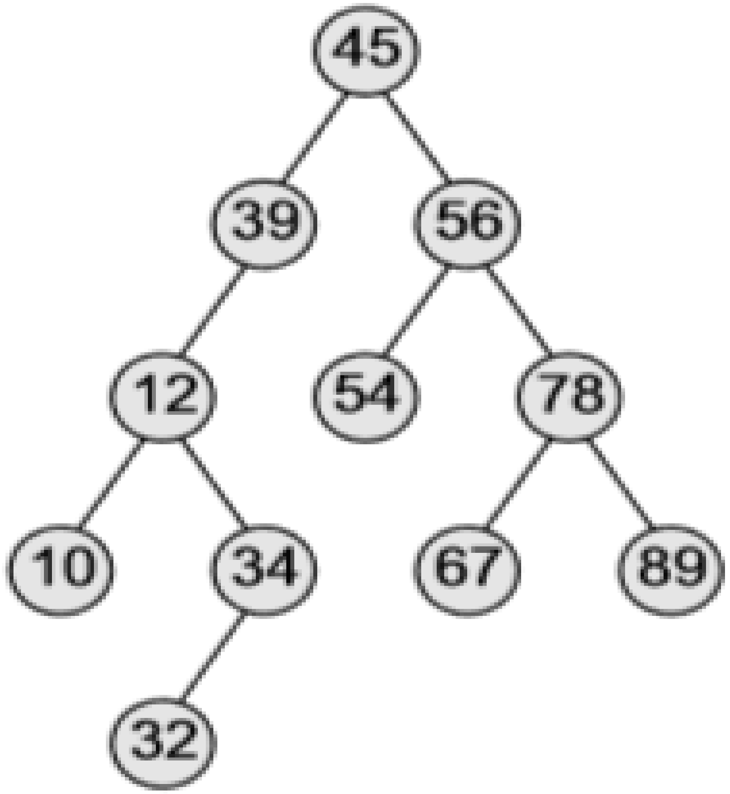
- Recursive algorithm.

```
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
```

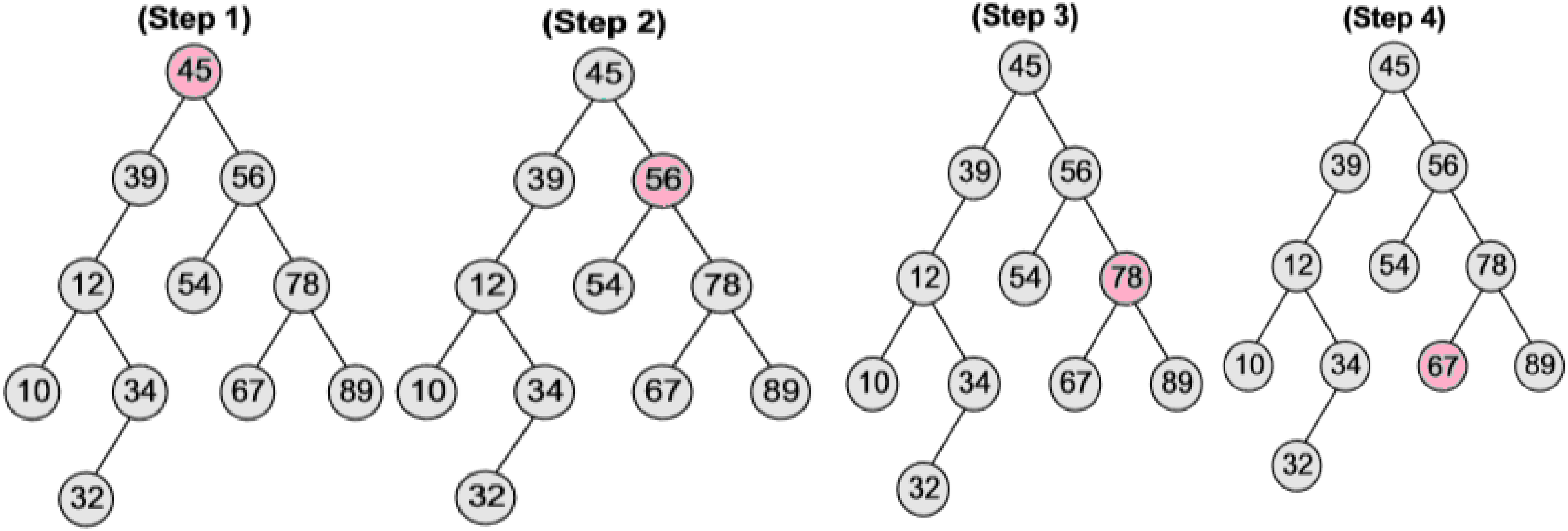
Searching for a Node in Binary Search Tree

- Search for element 67 in the Binary Search Tree.



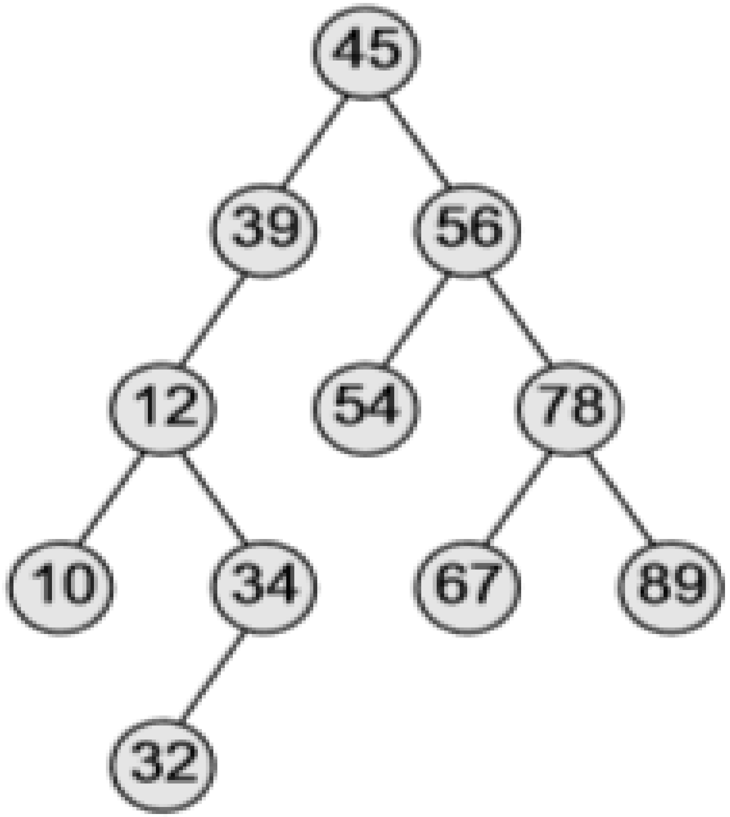
Searching for a Node in Binary Search Tree

- Search for element 67 in the Binary Search Tree.



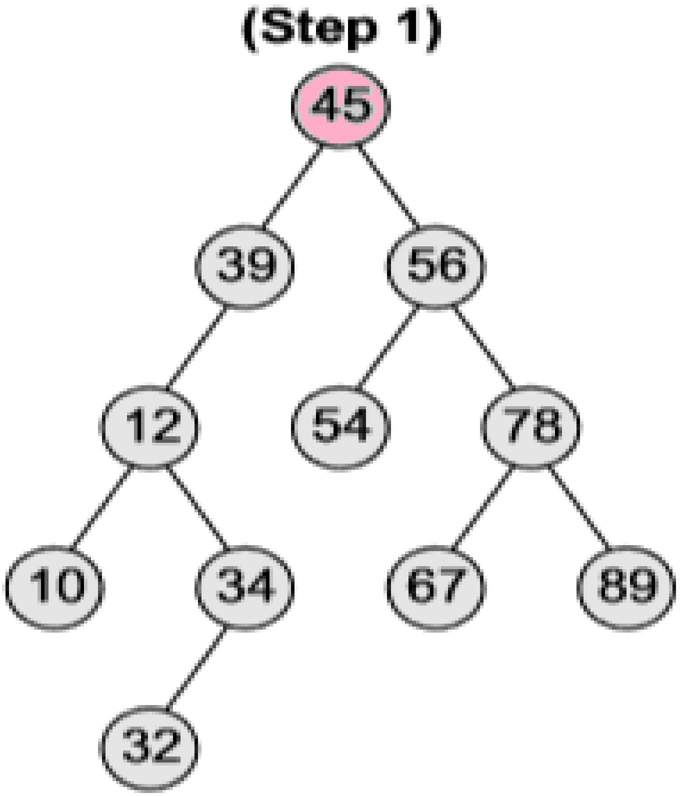
Searching for a Node in Binary Search Tree

- Search for element 12 in the Binary Search Tree.



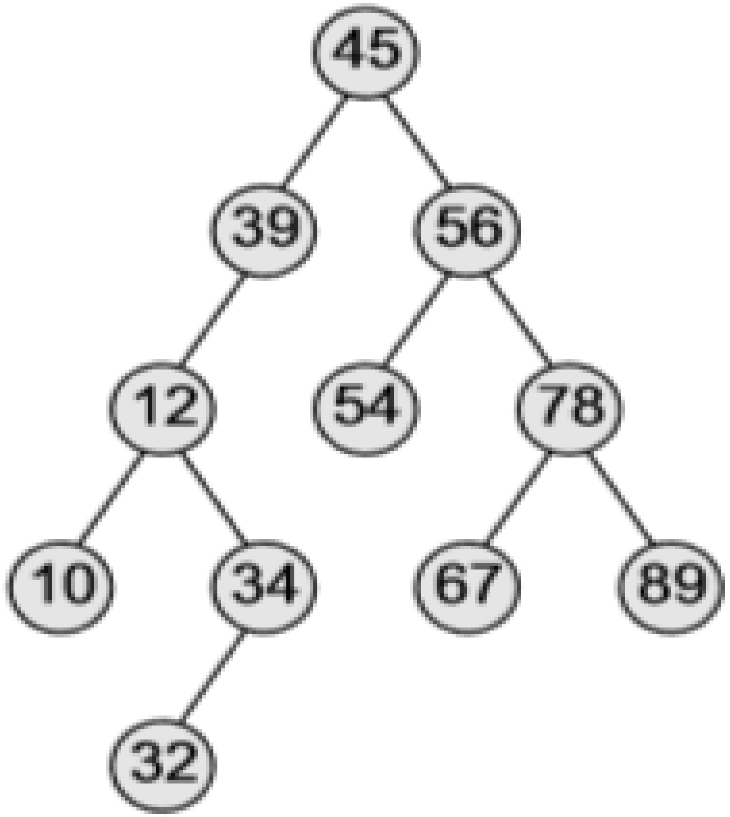
Searching for a Node in Binary Search Tree

- Search for element 12 in the Binary Search Tree.



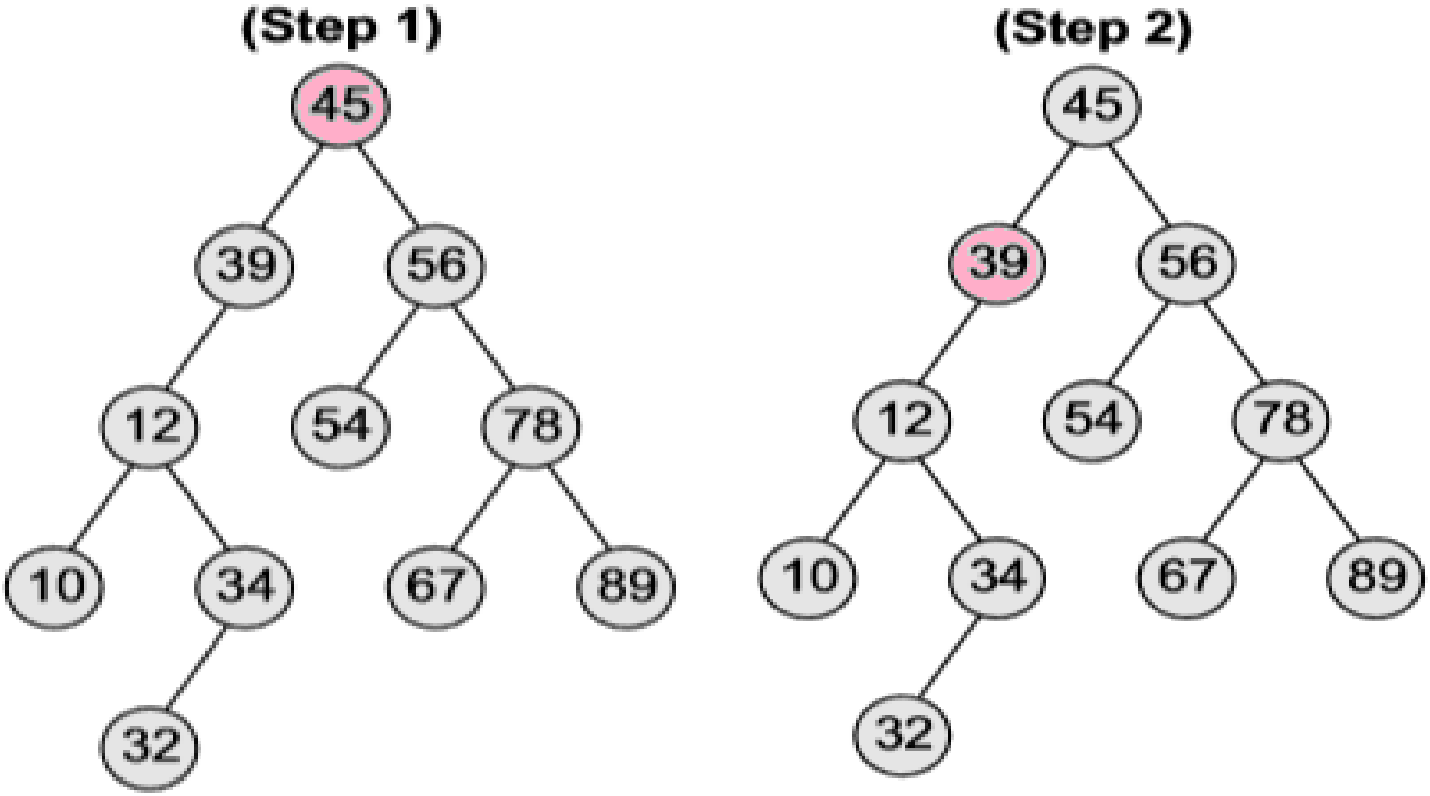
Searching for a Node in Binary Search Tree

- Search for element 40 in the Binary Search Tree.



Searching for a Node in Binary Search Tree

- Search for element 40 in the Binary Search Tree.



Inserting a Node into Binary Search Tree

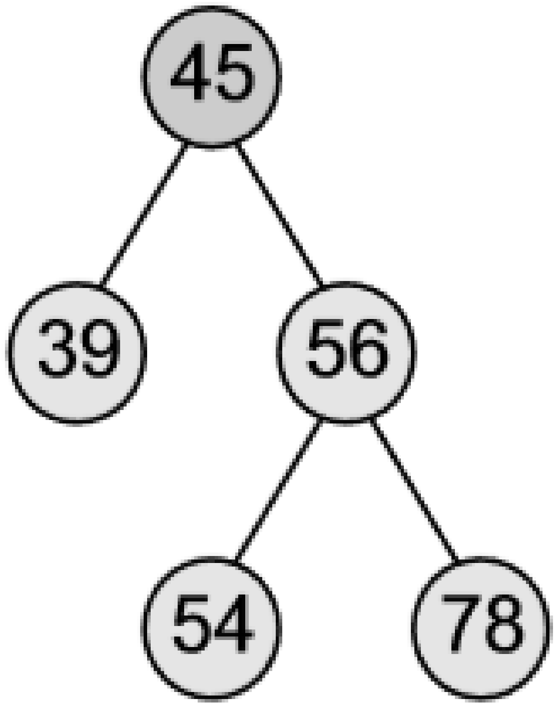
- Recursive algorithm.

```
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
```

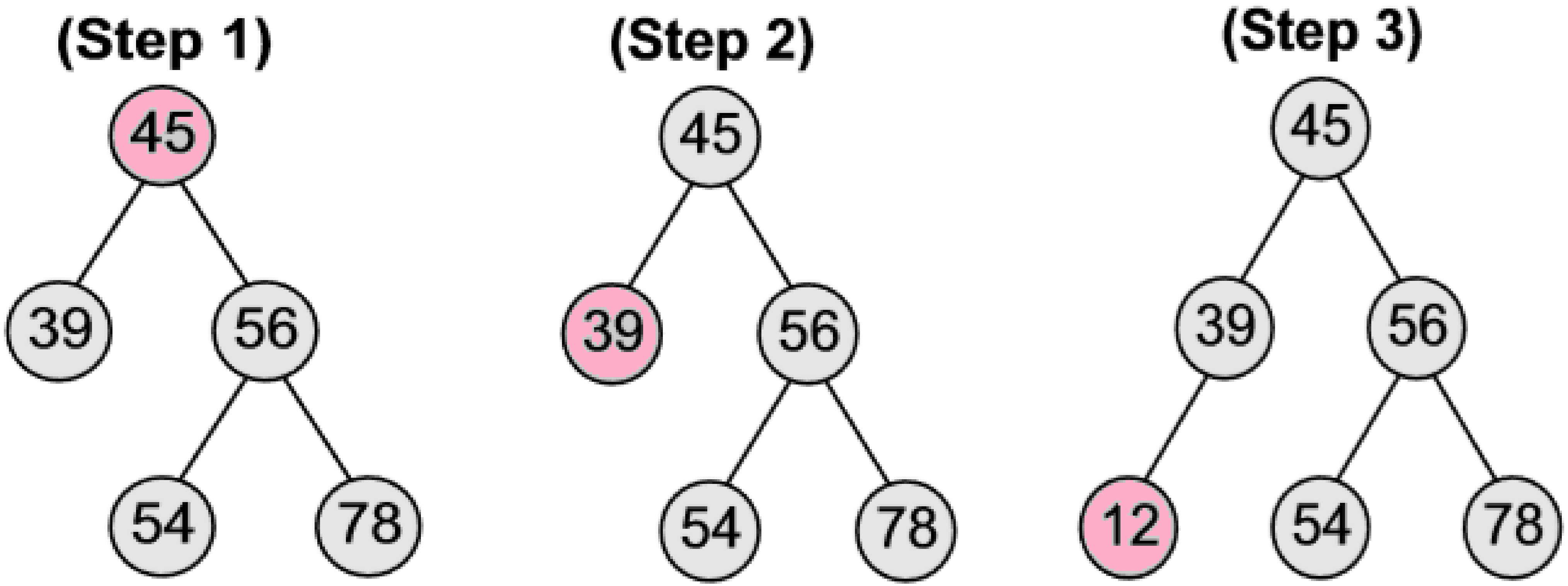
Inserting a Node into Binary Search Tree

- Insert an element 12 into the Binary Search Tree.



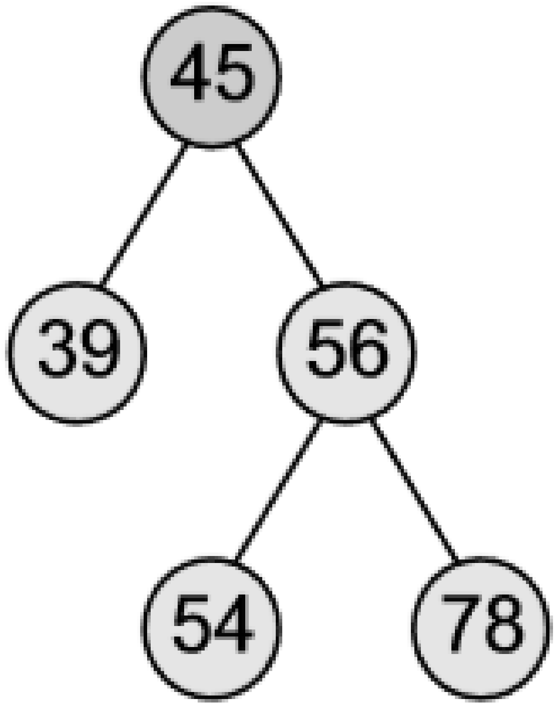
Inserting a Node into Binary Search Tree

- Insert an element 12 into the Binary Search Tree.



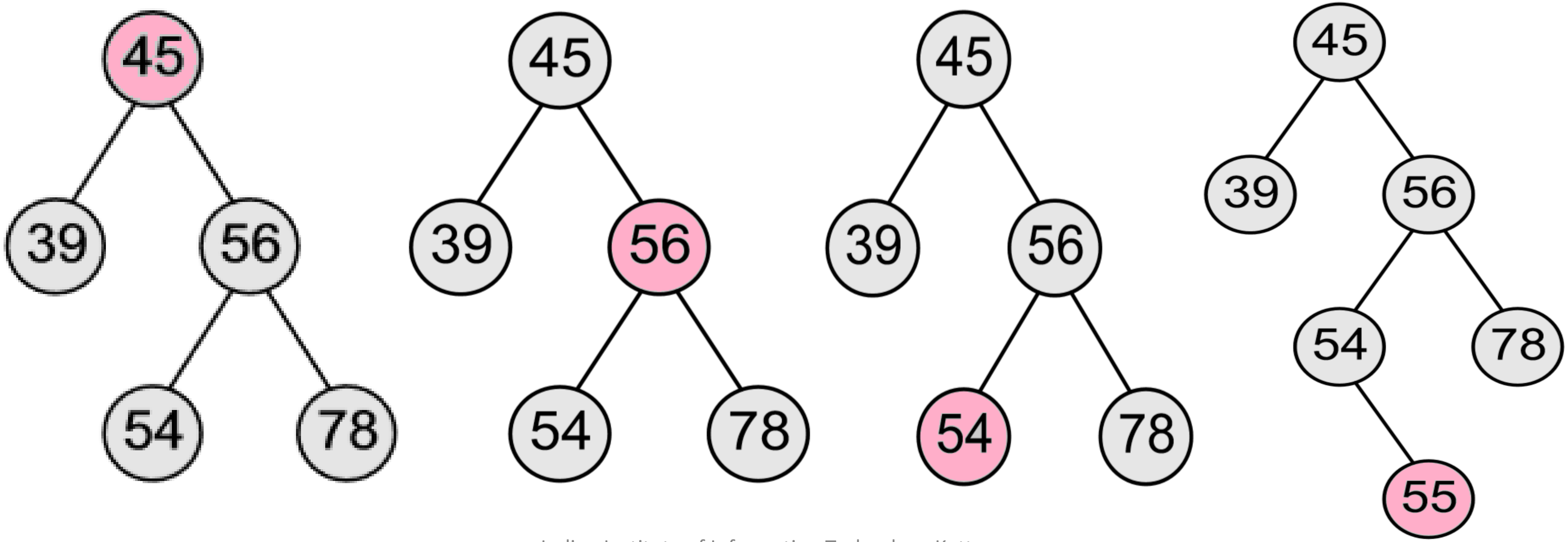
Inserting a Node into Binary Search Tree

- Insert an element 55 into the Binary Search Tree.



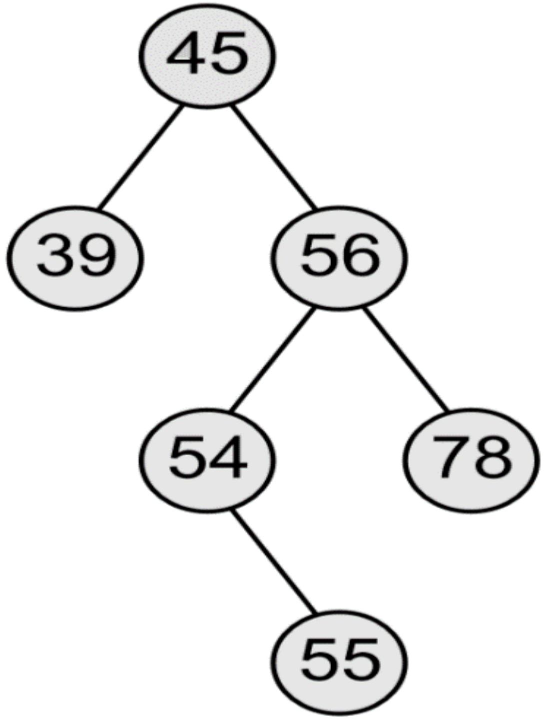
Inserting a Node into Binary Search Tree

- Insert an element 55 into the Binary Search Tree.



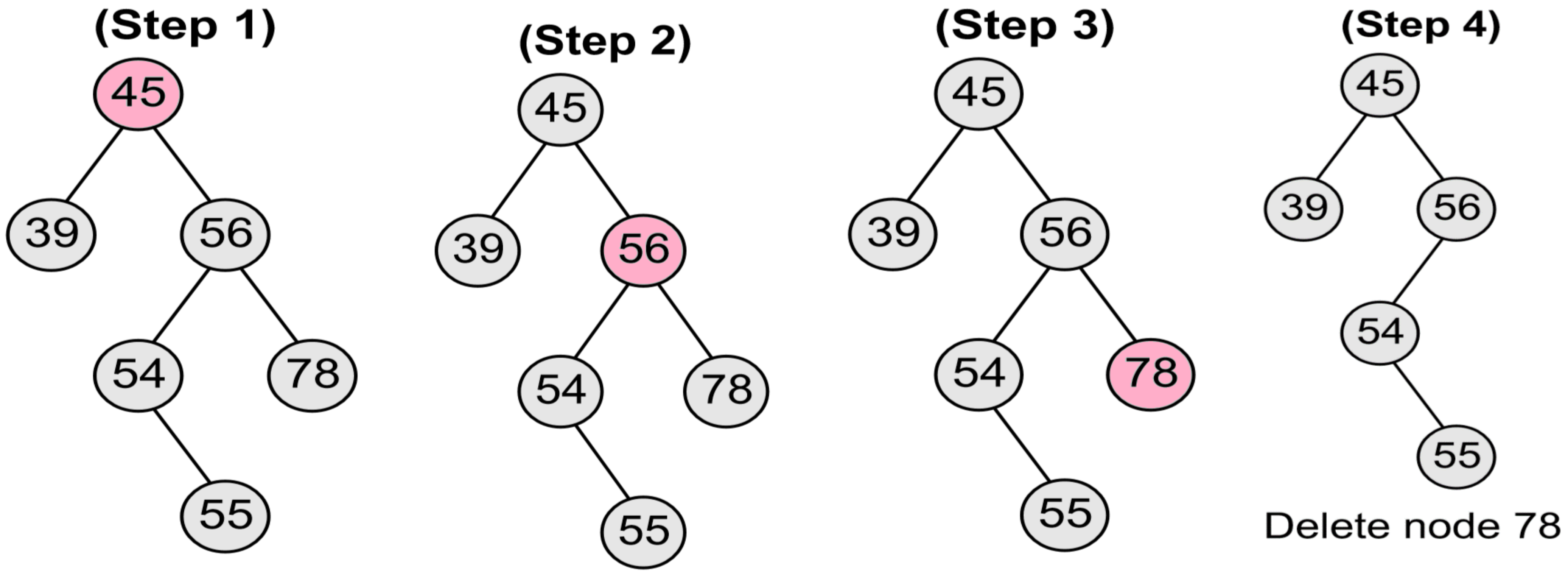
Deleting a Node from Binary Search Tree

- **Case 1:** Deleting a Node that has No Children.
- Delete node 78 from the Binary Search Tree.



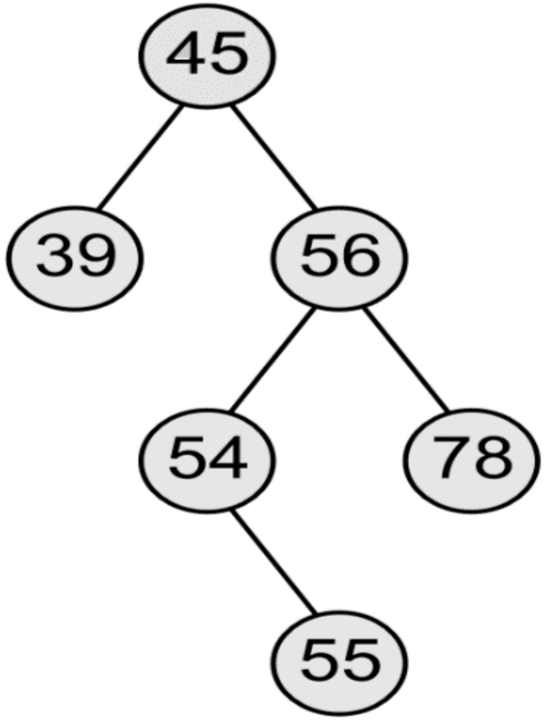
Deleting a Node from Binary Search Tree

- **Case 1:** Deleting a Node that has No Children.
- Delete node 78 from the Binary Search Tree.



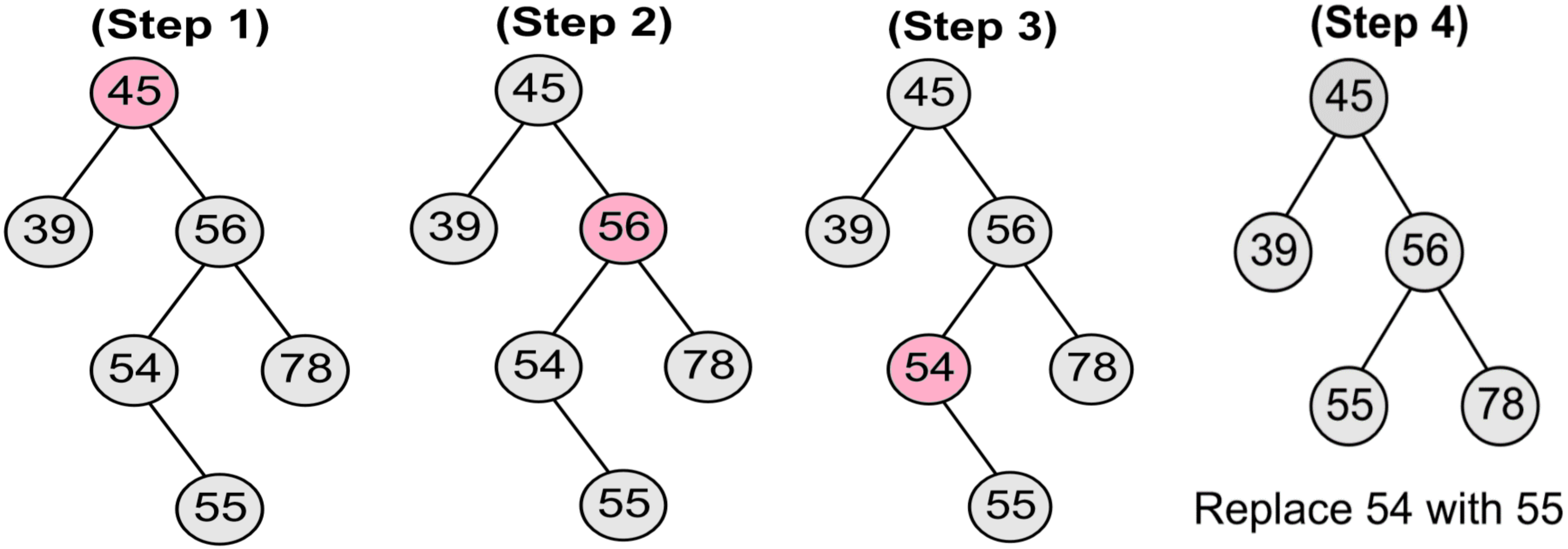
Deleting a Node from Binary Search Tree

- **Case 2:** Deleting a Node with one Child.
- Delete node 54 from the Binary Search Tree.



Deleting a Node from Binary Search Tree

- **Case 2:** Deleting a Node with one Child.
- Delete node 54 from the Binary Search Tree.



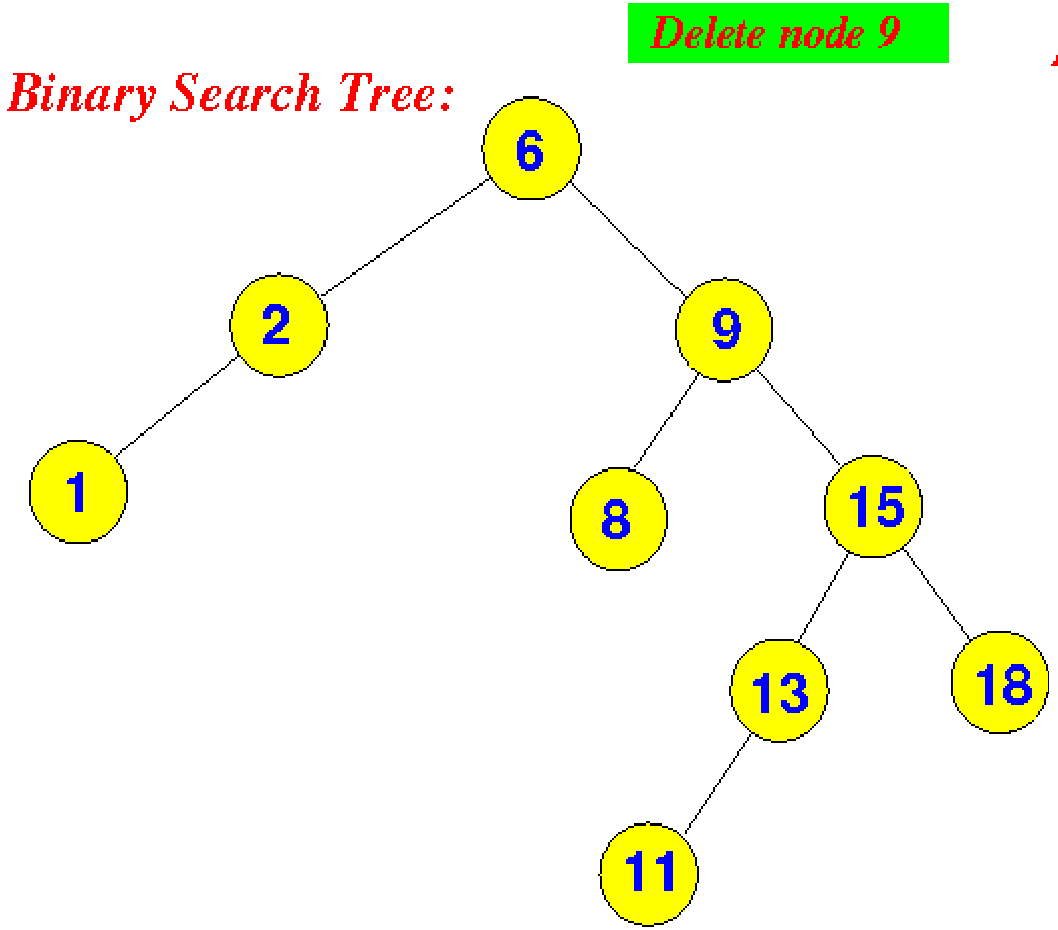
Deleting a Node from Binary Search Tree

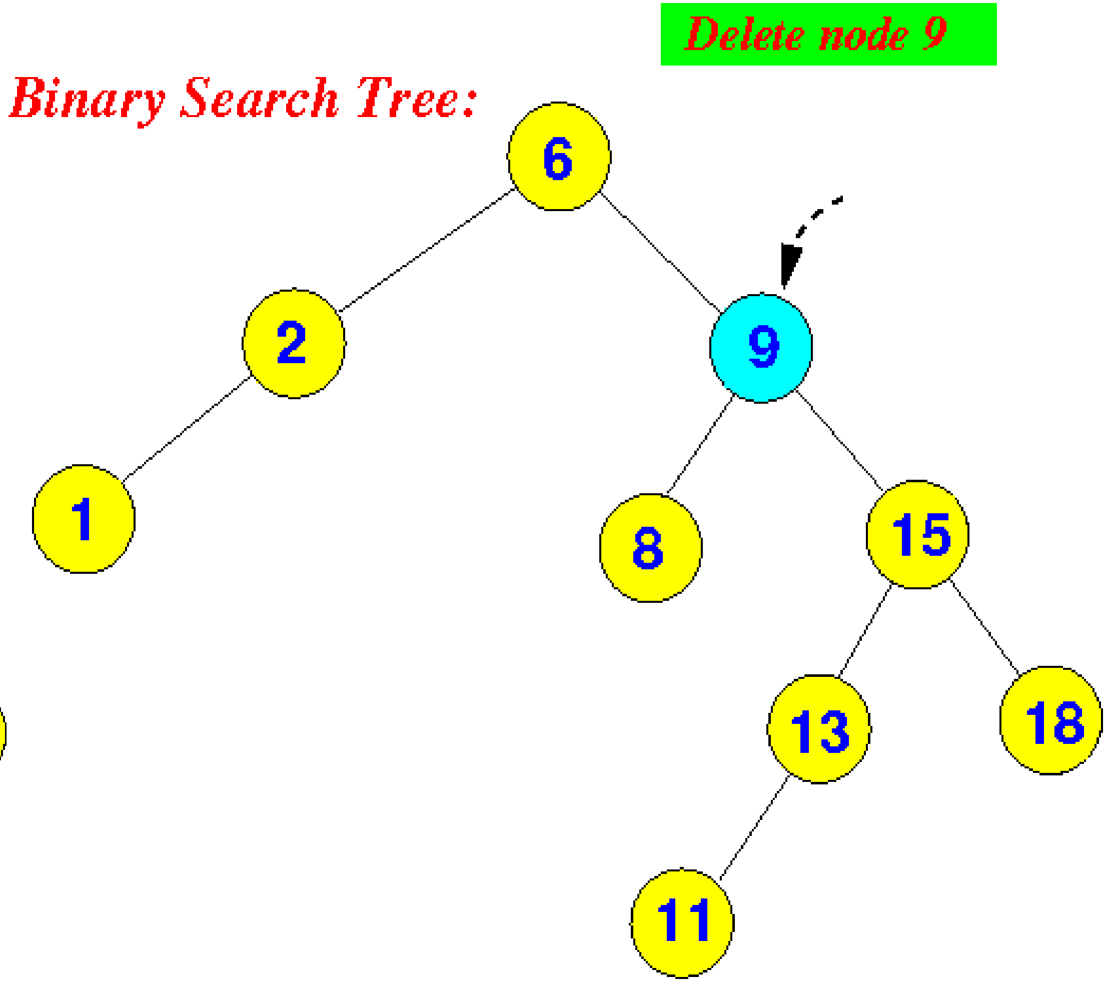
- **Case 3:** Deleting a Node with two Children.

- Replace the node’s value with its:
- In-order predecessor (**largest value in the left sub-tree**)

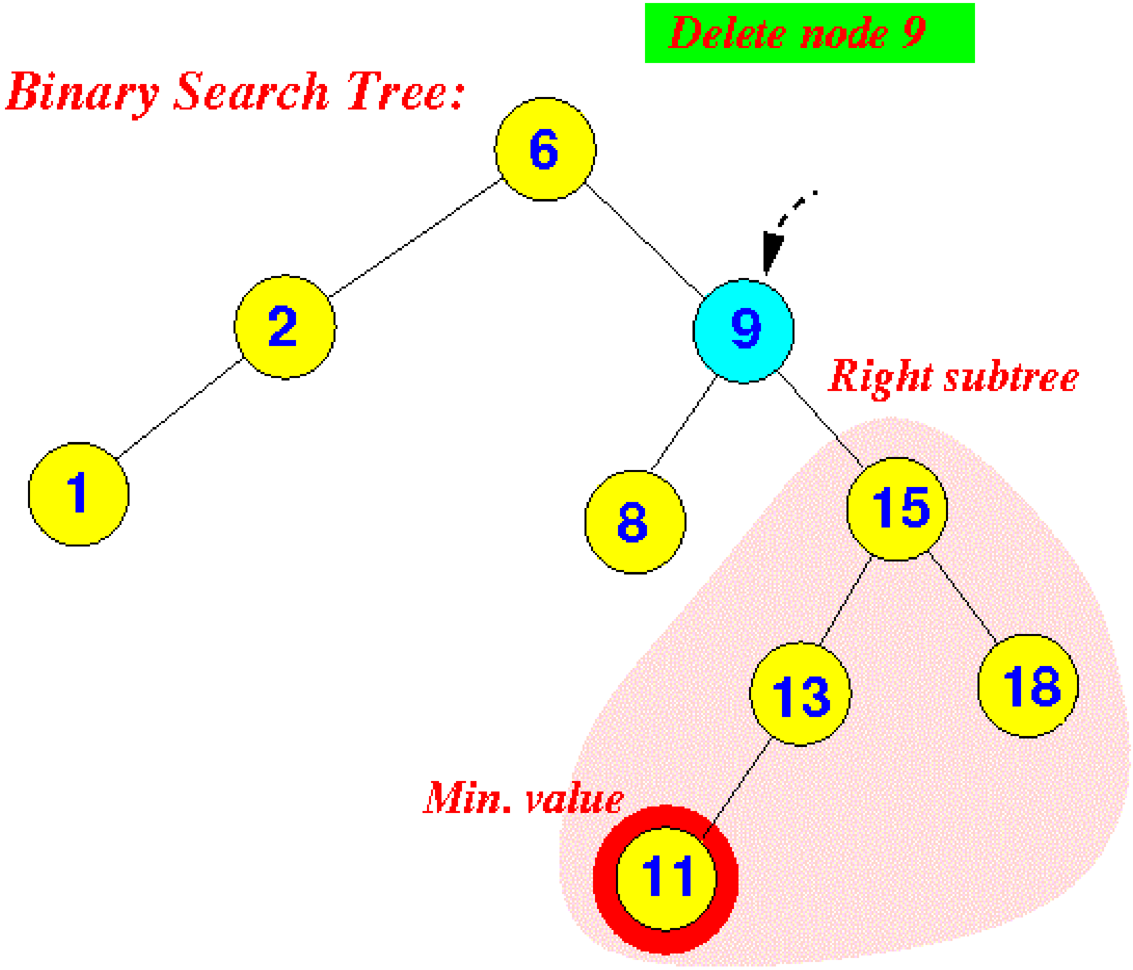
  OR
- In-order successor (**smallest value in the right sub-tree**).

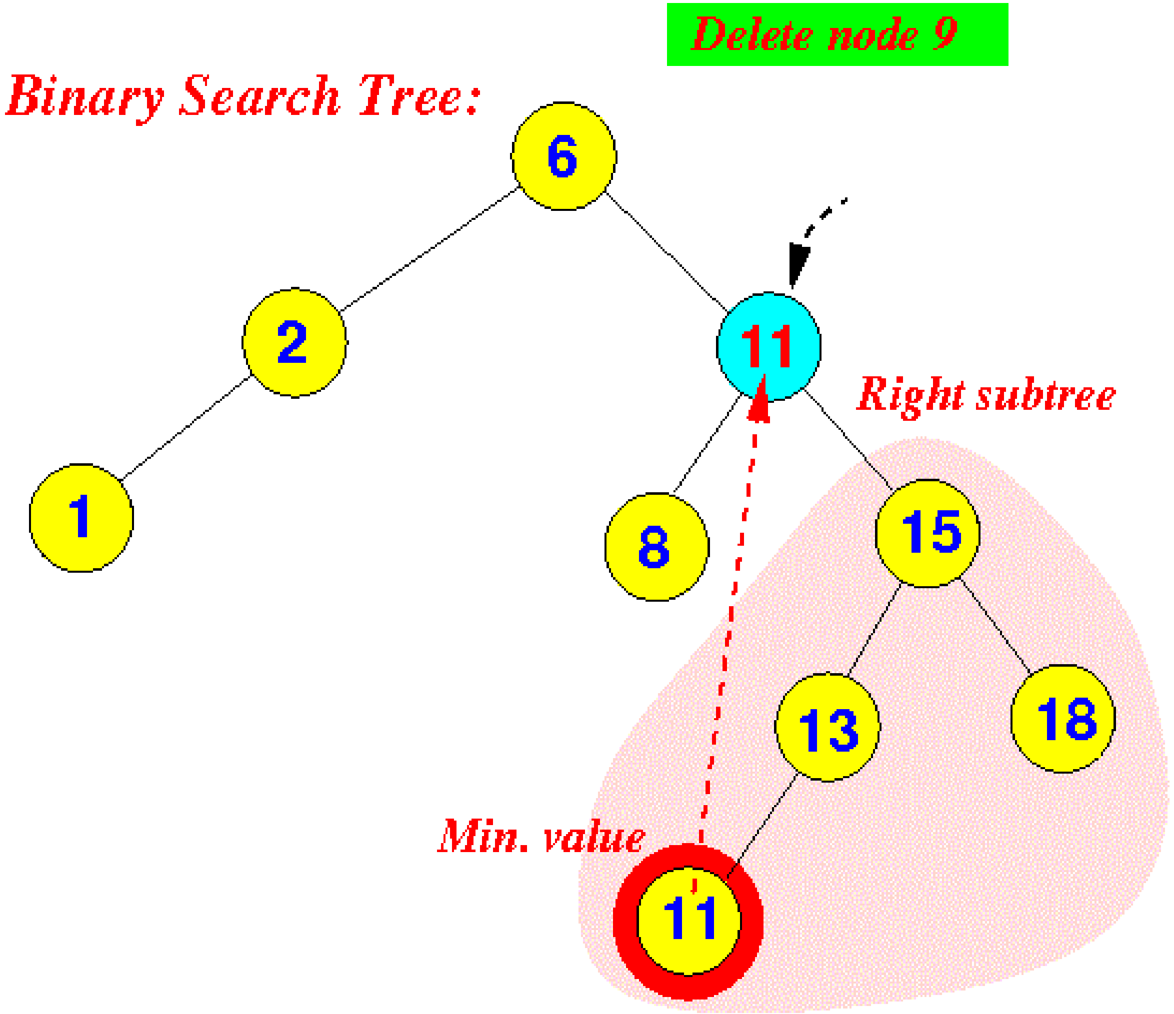
Deleting a Node from Binary Search Tree



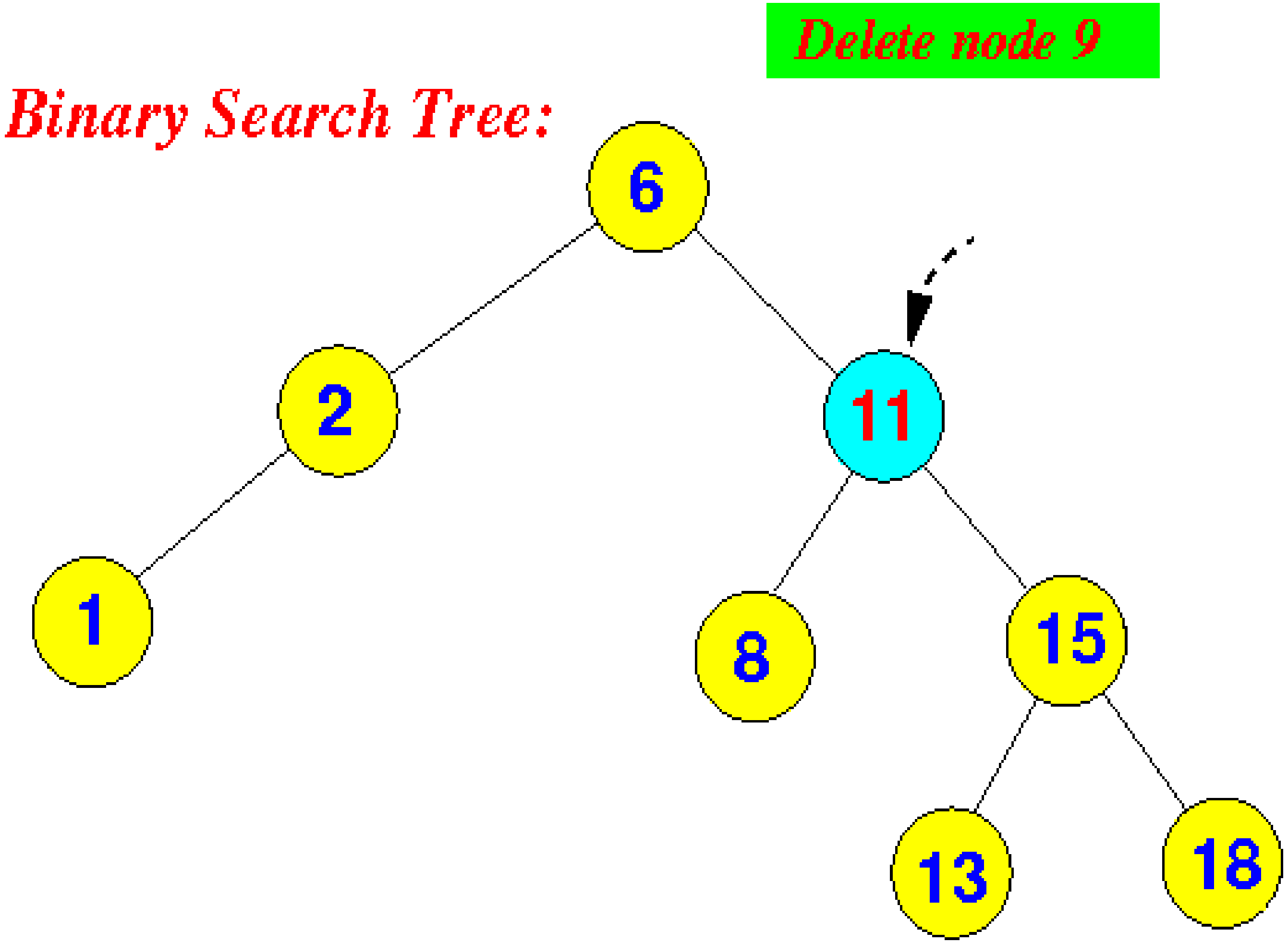


Deleting a Node from Binary Search Tree

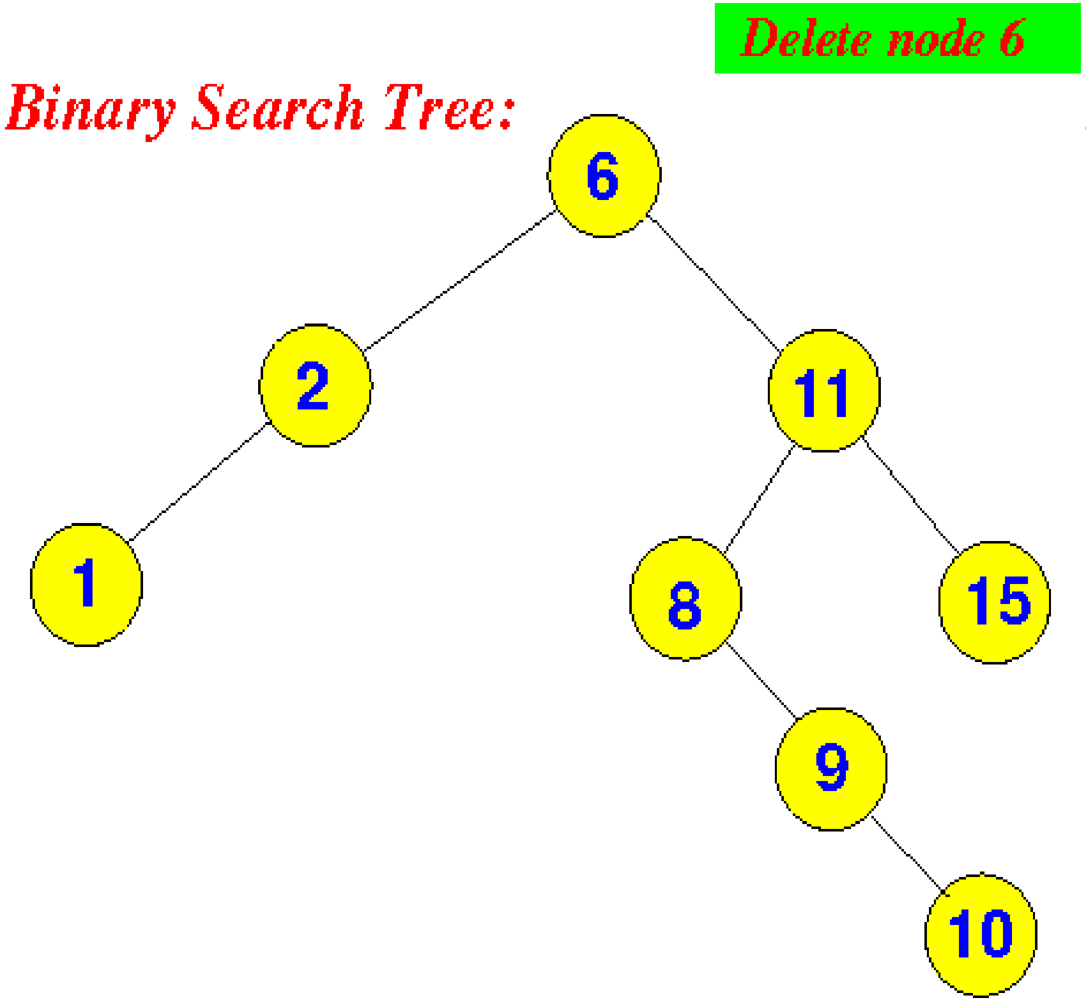


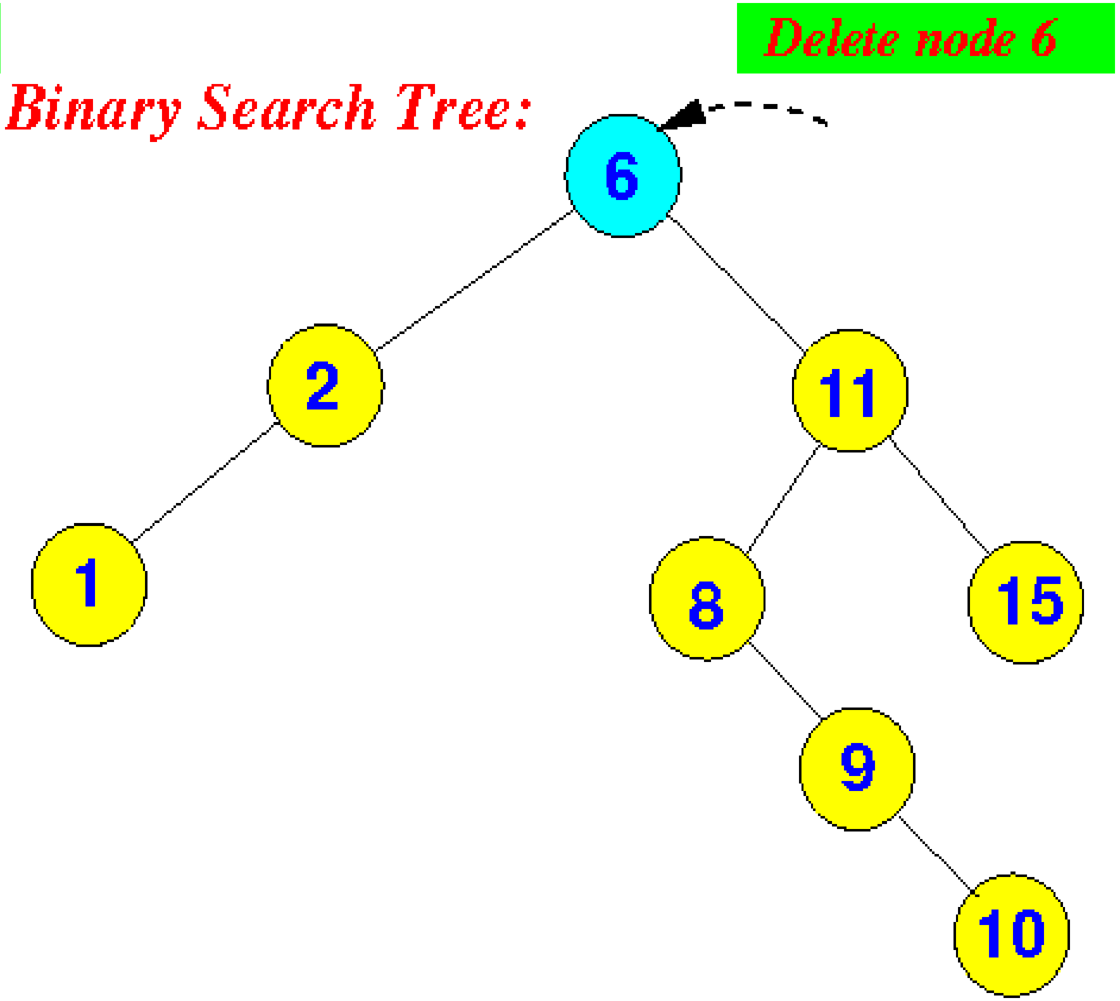


Deleting a Node from Binary Search Tree

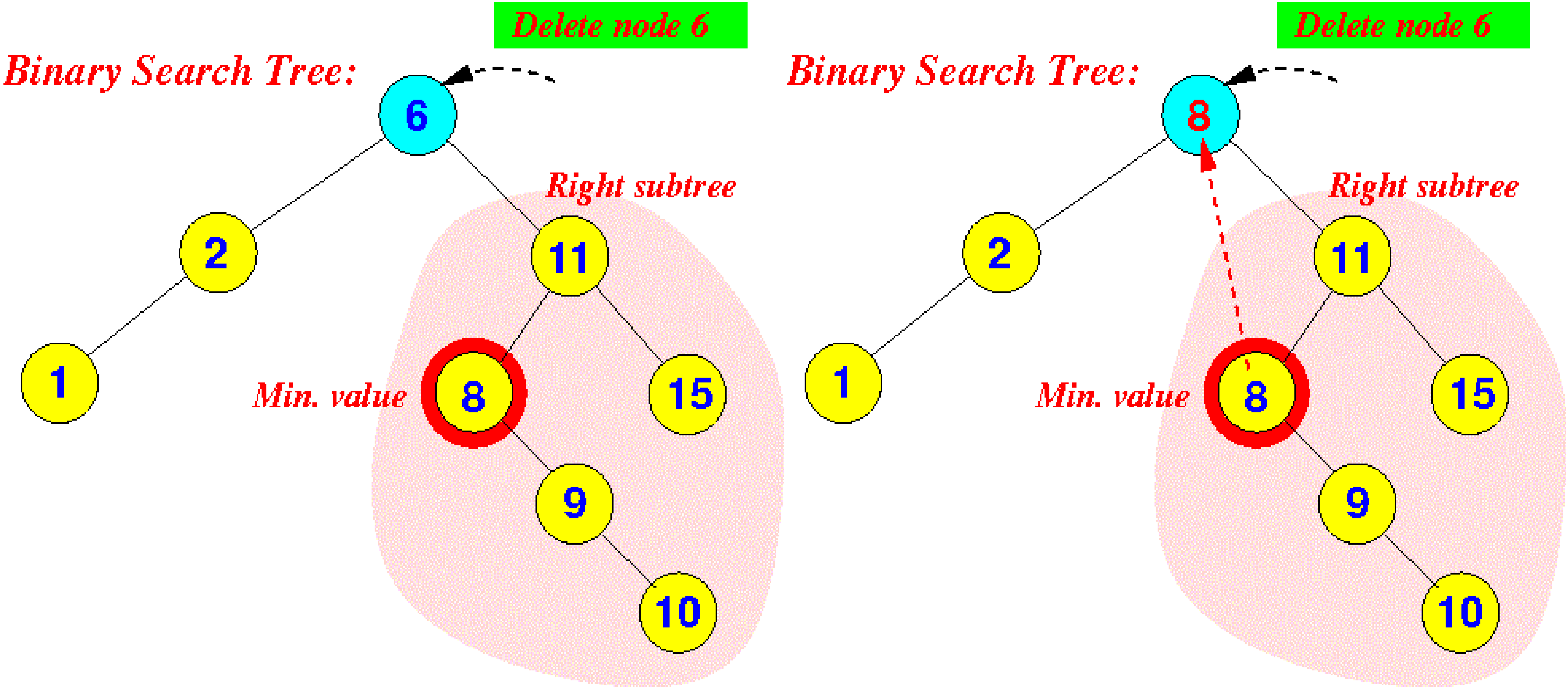


Deleting a Node from Binary Search Tree

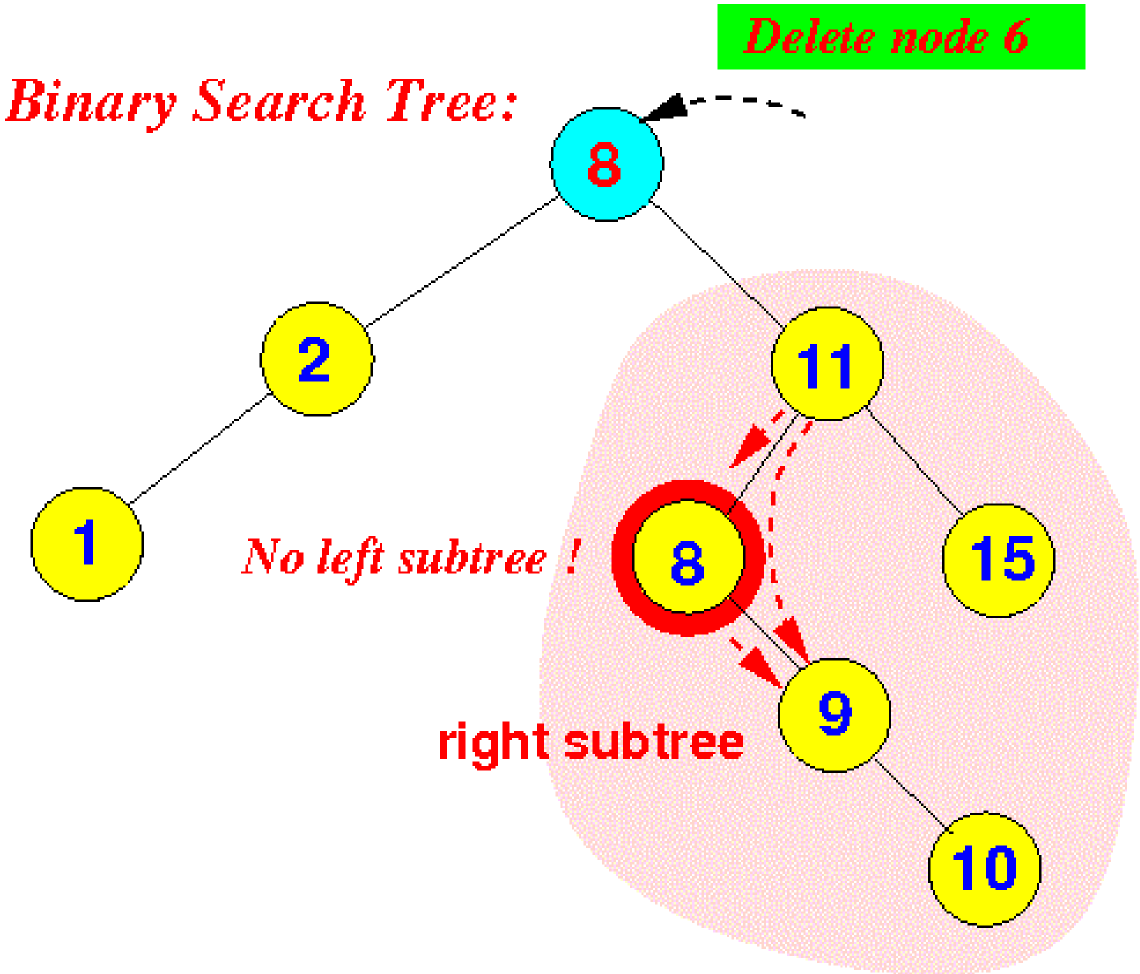


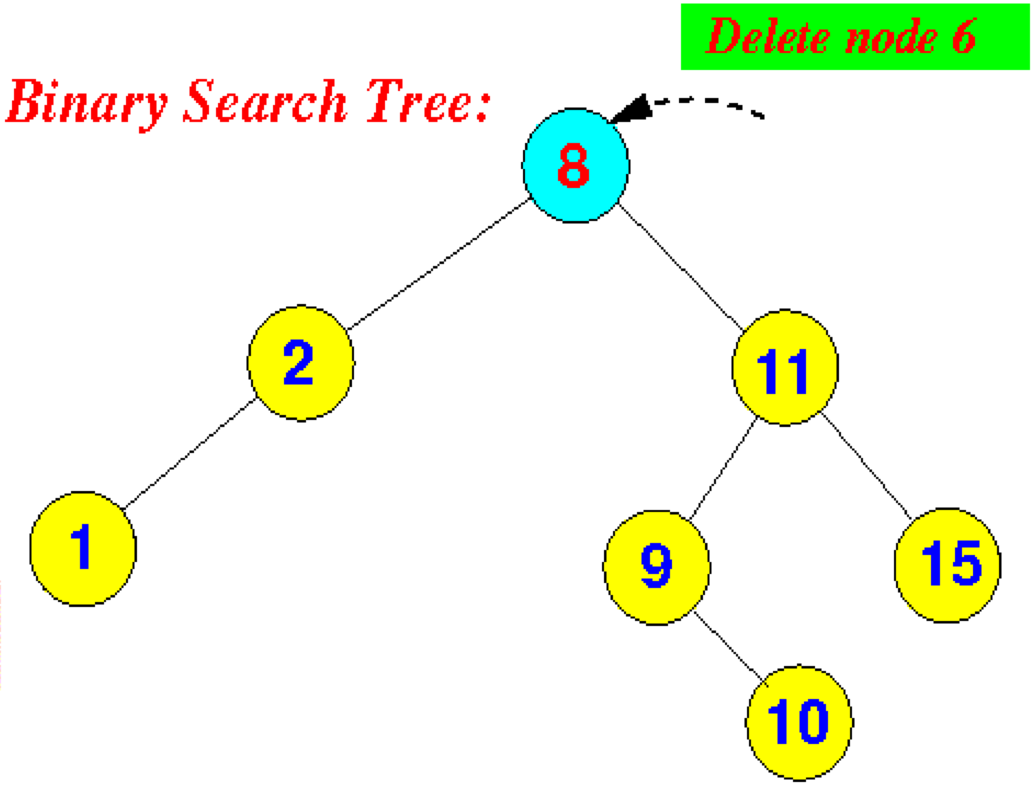


Deleting a Node from Binary Search Tree



Deleting a Node from Binary Search Tree





Deleting a Node from Binary Search Tree

```
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
```

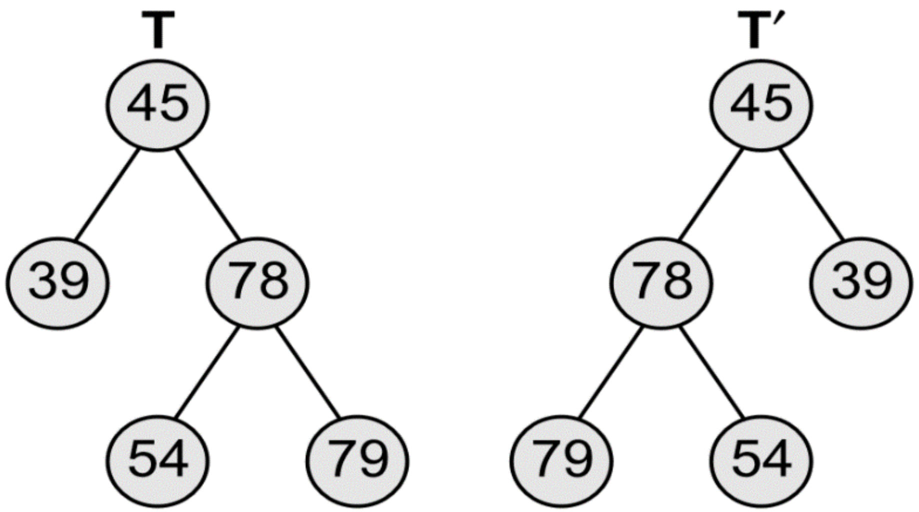
Mirror Image of a Binary Search Tree

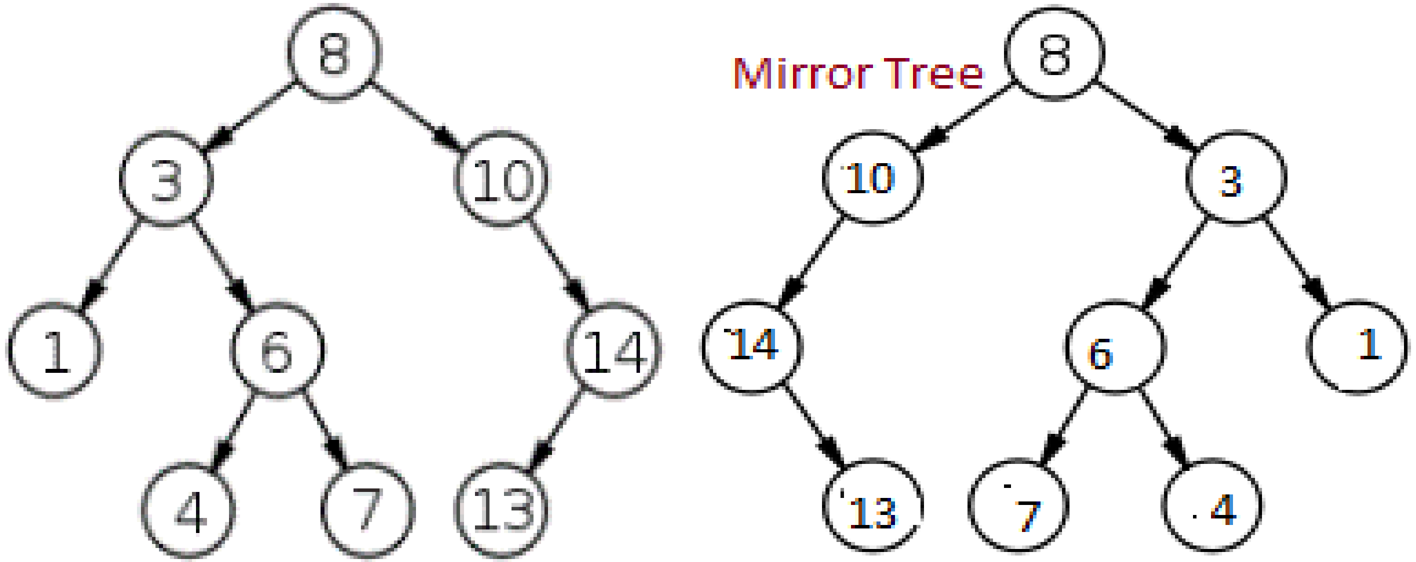
- Obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree.

```
MirrorImage(TREE)

Step 1: IF TREE != NULL
              MirrorImage(TREE -> LEFT)
              MirrorImage(TREE -> RIGHT)
              SET TEMP = TREE -> LEFT
              SET TREE -> LEFT = TREE -> RIGHT
              SET TREE -> RIGHT = TEMP
        [END OF IF]
Step 2: END
```

Mirror Image of a Binary Search Tree





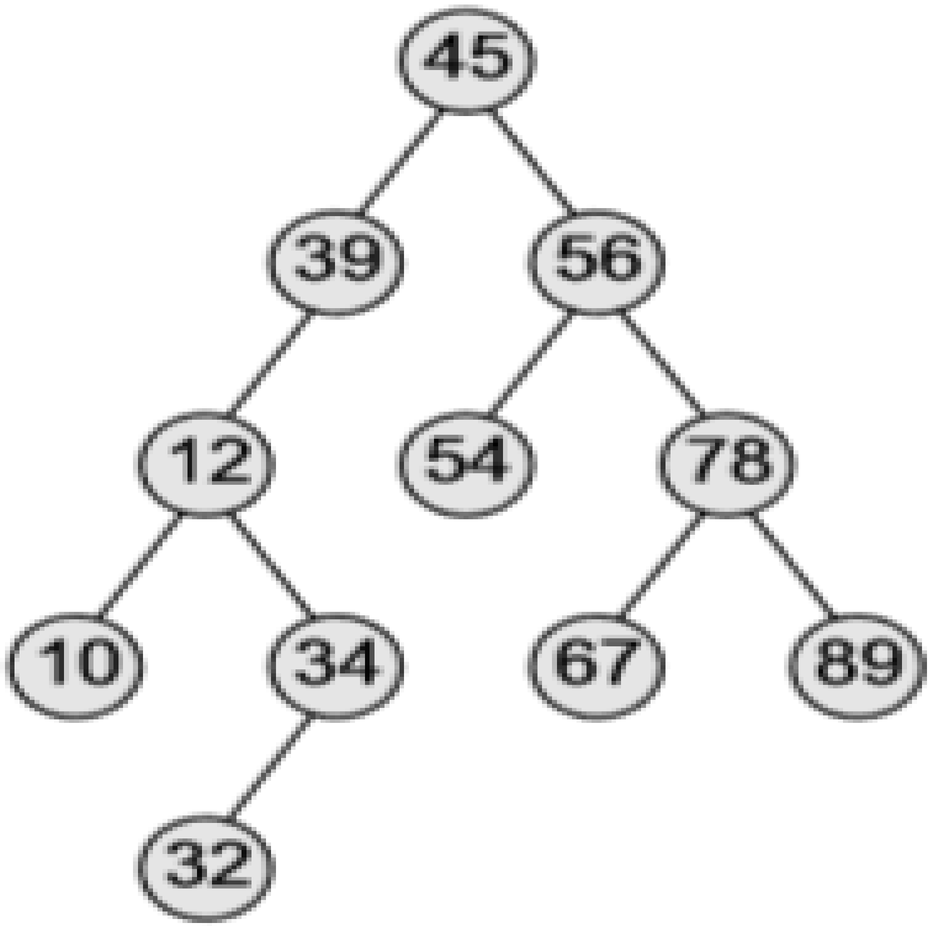
Smallest element in a Binary Search Tree

- Smaller value will occur in the left sub-tree.
- Smallest value is the value of the leftmost node of the left sub-tree.

```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
             Returen TREE
        ELSE
             Return findSmallestElement(TREE -> LEFT)
        [END OF IF]
Step 2: END
```

Smallest element in a Binary Search Tree



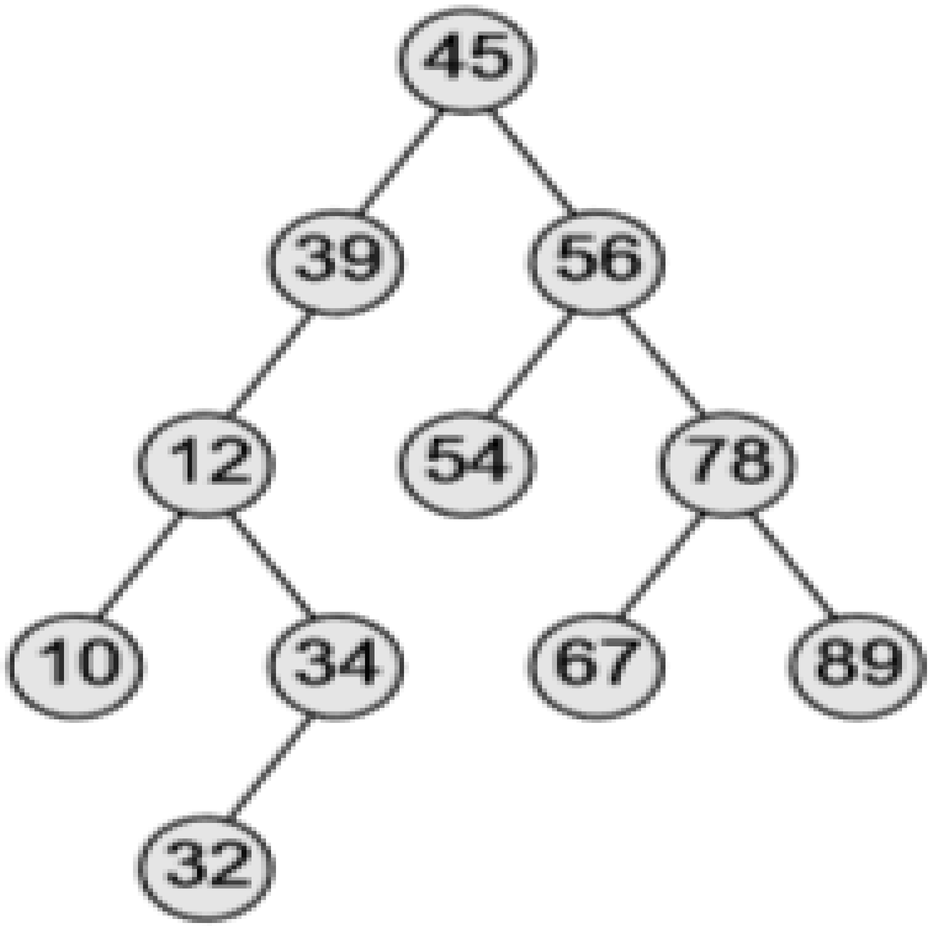
Largest element in a Binary Search Tree

- Largest value is the rightmost node of the right subtree.

```
findLargestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
              Return TREE
        ELSE
             Return findLargestElement(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

Largest element in a Binary Search Tree



Deleting the Binary Search Tree

- First delete the elements/nodes in the left sub-tree and then delete the nodes in the right sub-tree.

```
deleteTree(TREE)

Step 1: IF TREE != NULL
                deleteTree (TREE -> LEFT)
                deleteTree (TREE -> RIGHT)
                Free (TREE)
            [END OF IF]
Step 2: END
```

Deleting the Binary Search Tree