Efficient Binary Trees

Binary Search Trees

- 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.

Binary Search Tree

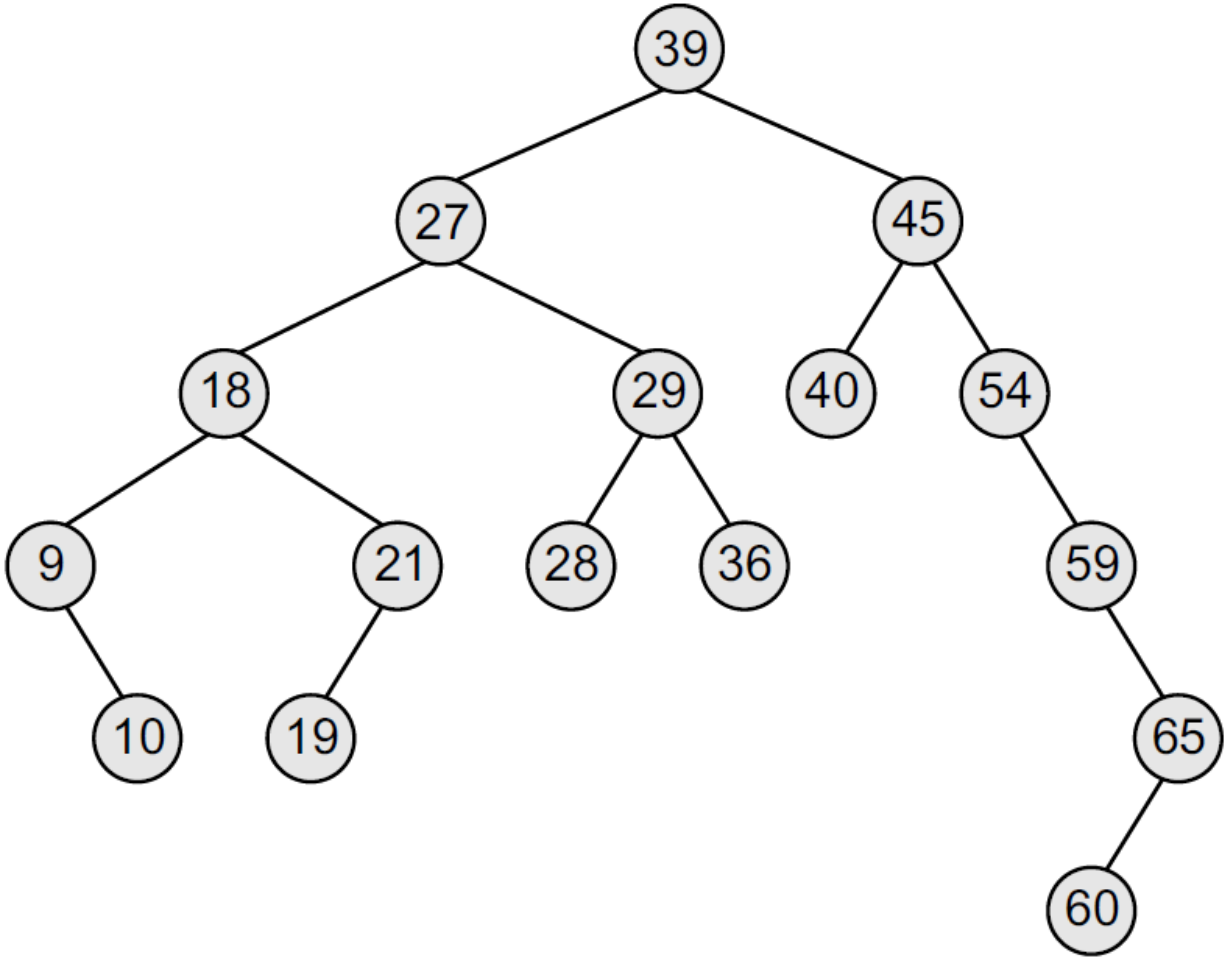


Figure 10.1 Binary search tree

Binary Search Tree

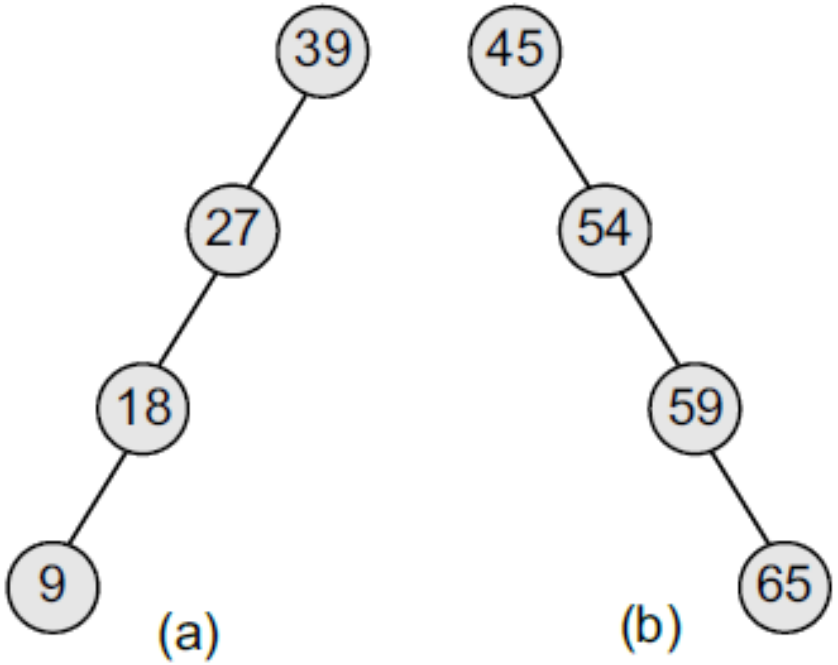


Figure 10.2 (a) Left skewed, and (b) right skewed binary search trees

Creating a BST from Given Values

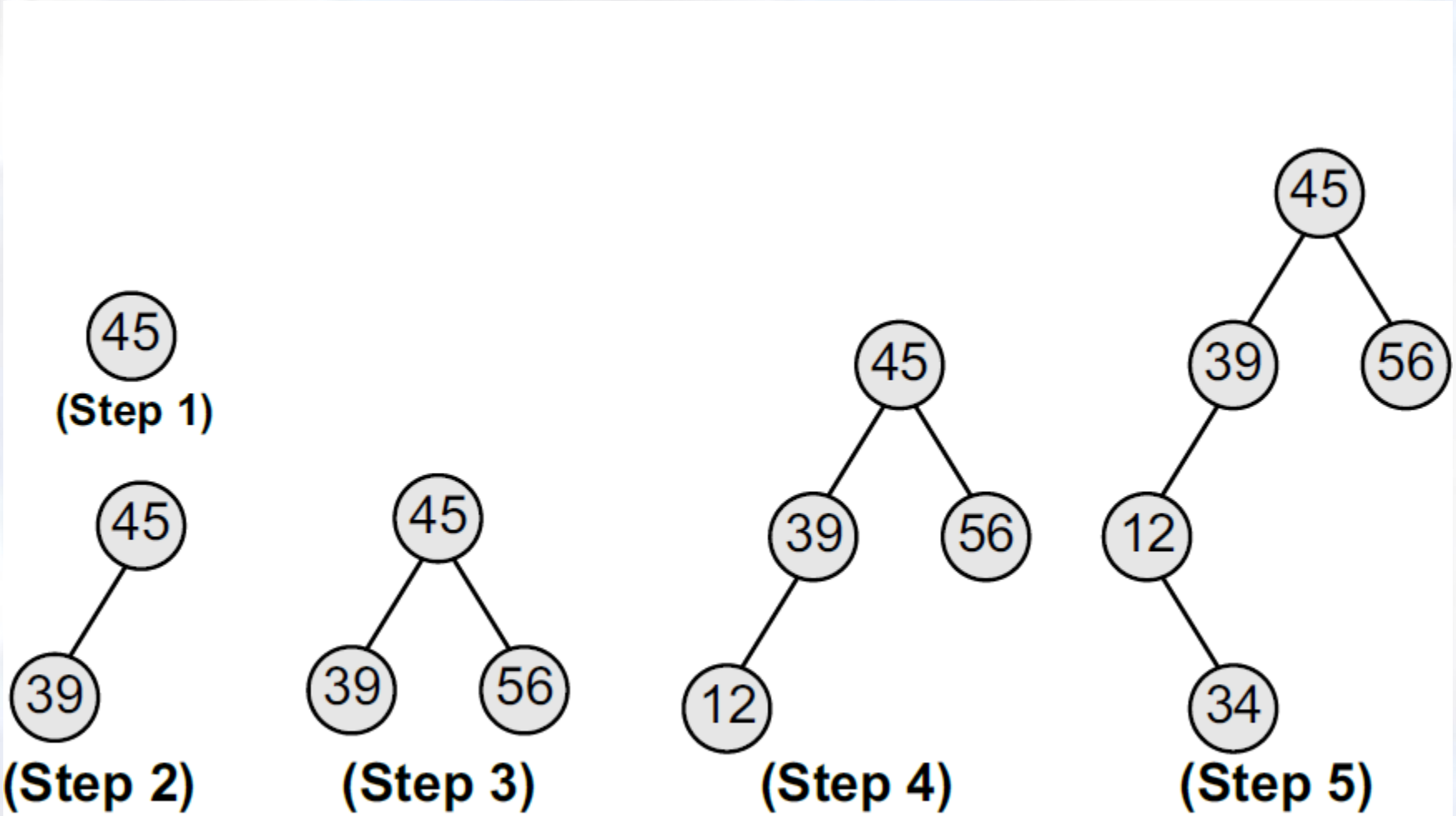
Create a BST from the key/data values given in following order:

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

BST??

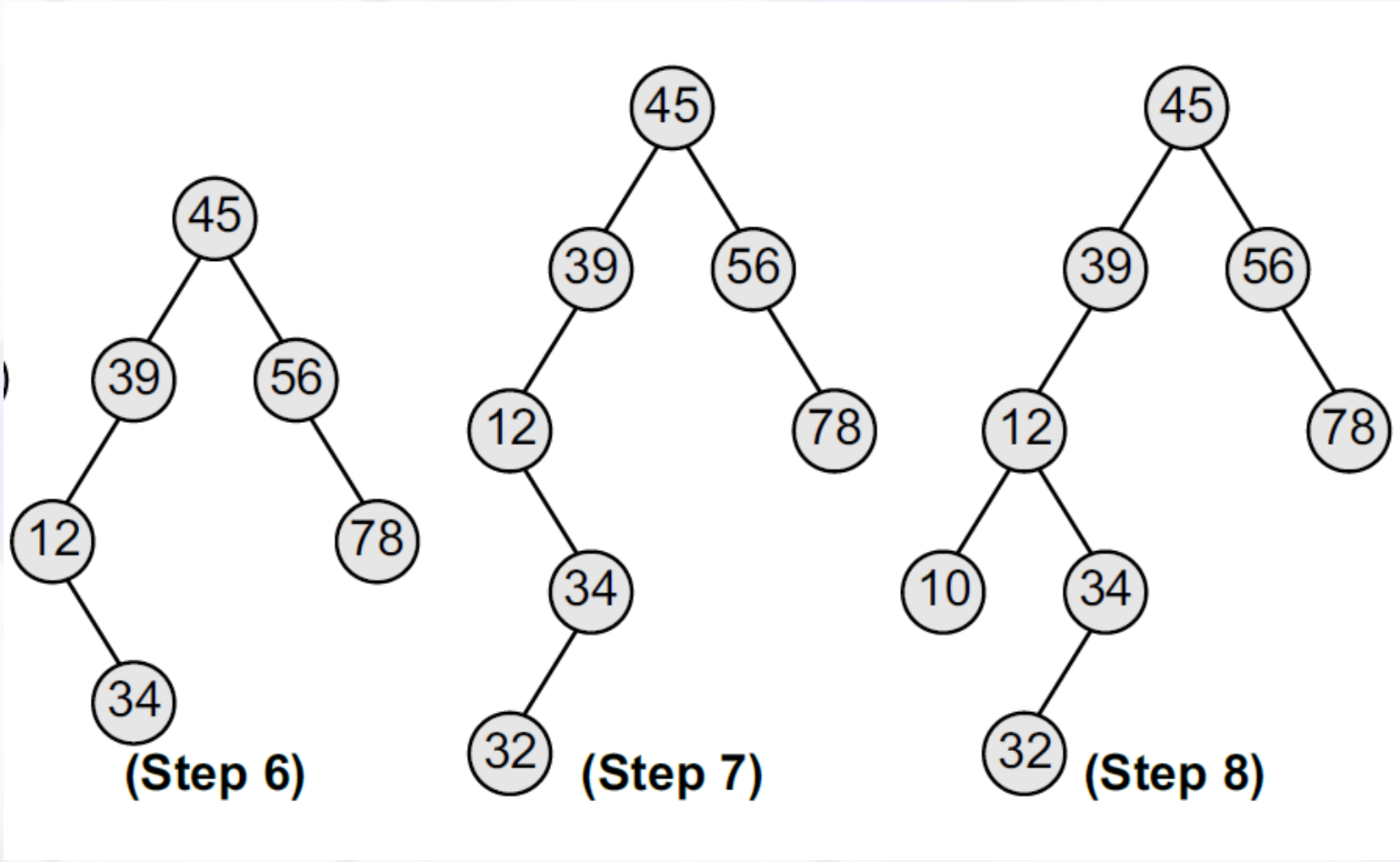
Creating a BST from Given Values

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



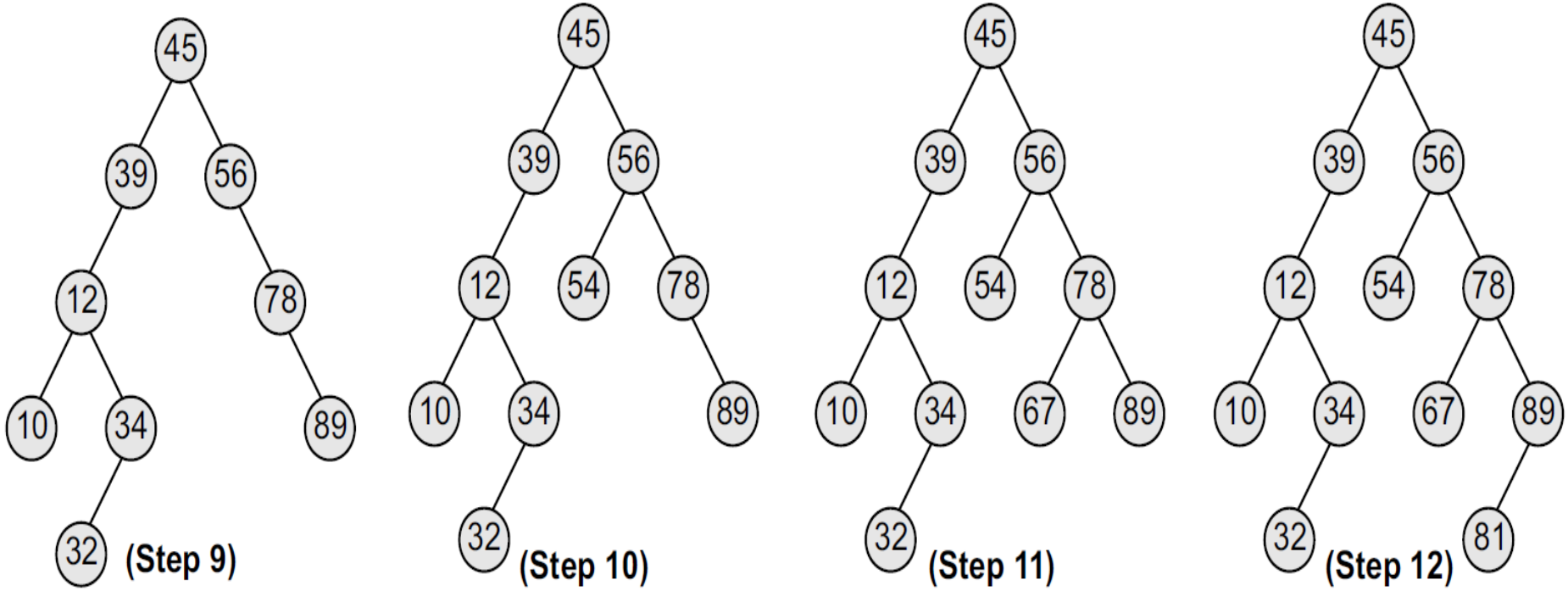
Creating a BST from Given Values

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



Creating a BST from Given Values

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



Code Implementation of BST

- Non-recursive
- Recursive

```
struct  nodetype{
        int key;
        struct nodetype *left;
        struct nodetype *right;
};
```

Traversal in BST

- In-order Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

Figure 9.17 Algorithm for in-order traversal

Traversal in BST

- Pre-order Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            Write TREE -> DATA
Step 3:            PREORDER(TREE -> LEFT)
Step 4:            PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

Figure 9.16 Algorithm for pre-order traversal

Traversal in BST

- Postorder Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

Figure 9.18 Algorithm for post-order traversal

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.

Algorithm to Search a Value in a BST

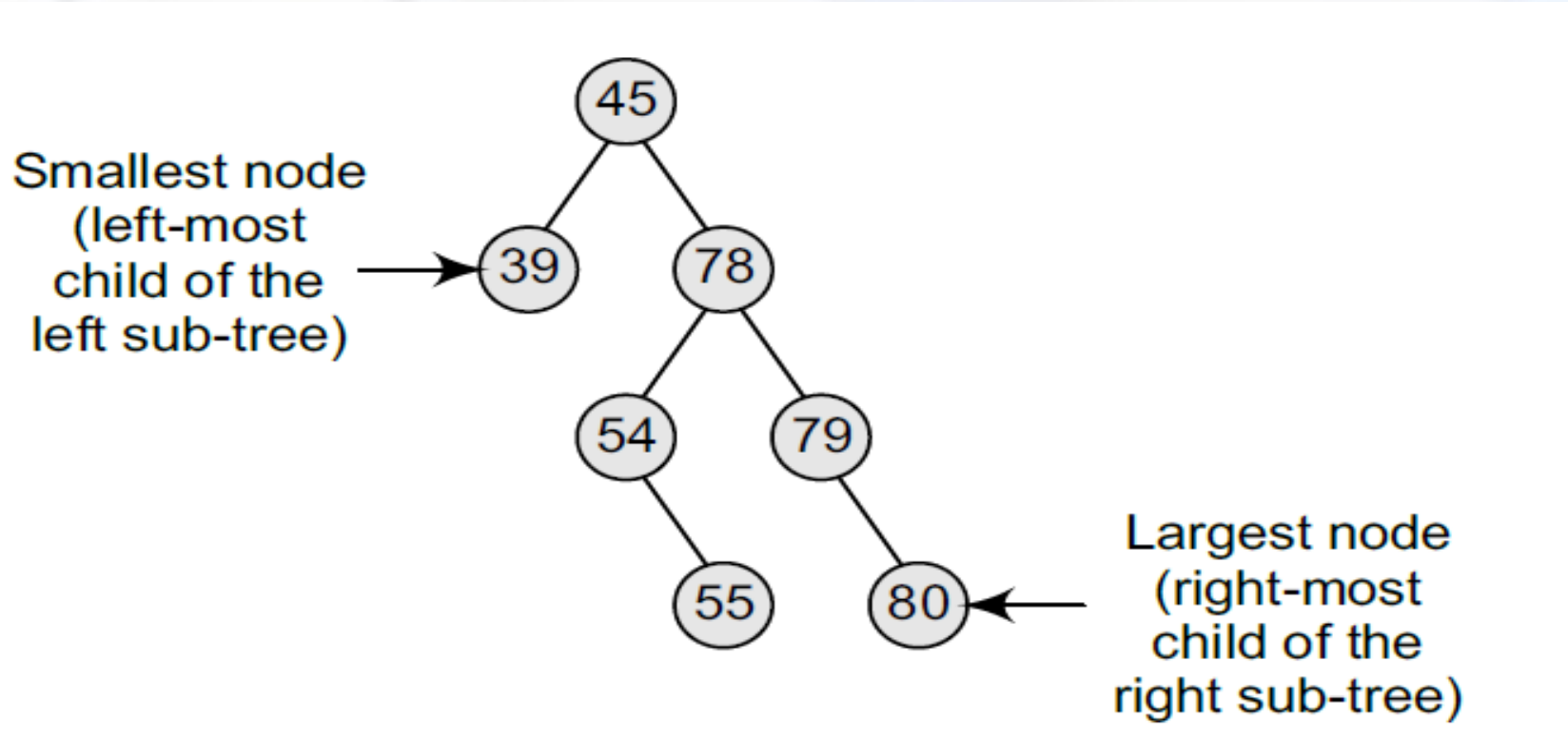
```
searchElement (TREE, VAL)
    Step 1:
    IF TREE->DATA = VAL OR TREE = NULL, then
       Return TREE
    ELSE IF VAL < TREE->DATA
        Return searchElement(TREE->LEFT, VAL)
    ELSE
        Return searchElement(TREE->RIGHT, VAL)
      END OF IF]
    END OF IF]
    Step 2: End
```

Algorithm to Insert a Value in a BST

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

Smallest element in BST

- By construction principle of BST, smallest element is the left most element in the tree. Largest element is the right-most element.



Smallest element in BST

- Algorithm:

```
findSmallestElement(TREE)

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

Figure 10.25 Algorithm to find the smallest node in a binary search tree

Largest element in BST

- Algorithm

```
findLargestElement(TREE)

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

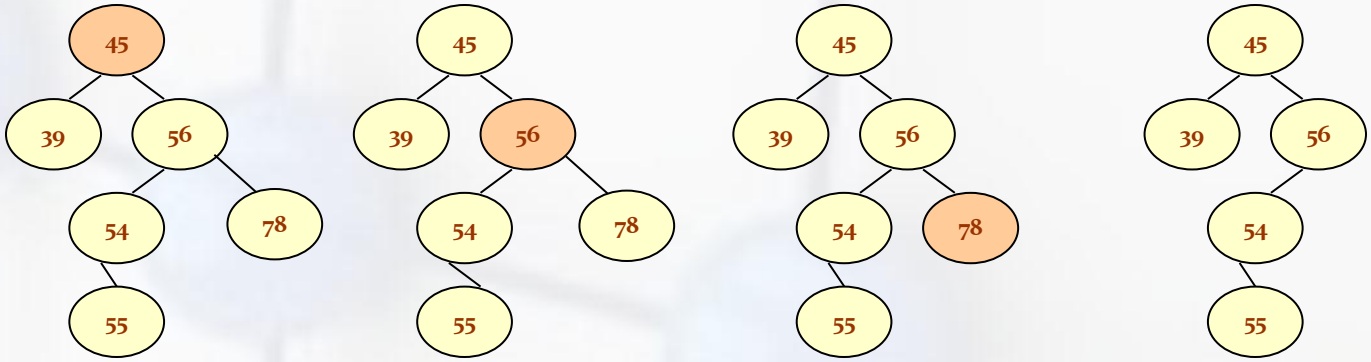
Figure 10.26 Algorithm to find the largest node in a binary search tree

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.

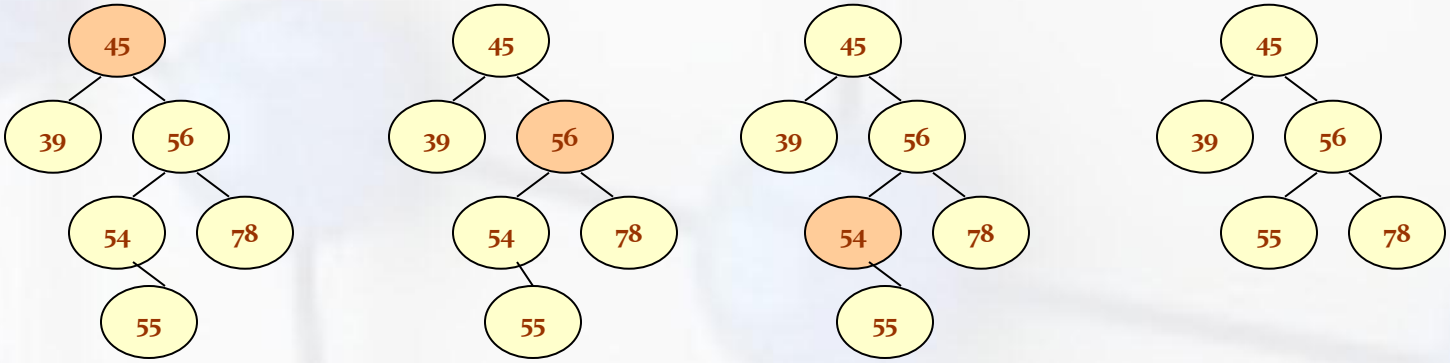
Case 1: **Deleting a node that has no children.**

For example, deleting node 78 in the tree below.



Deleting a Value from a BST

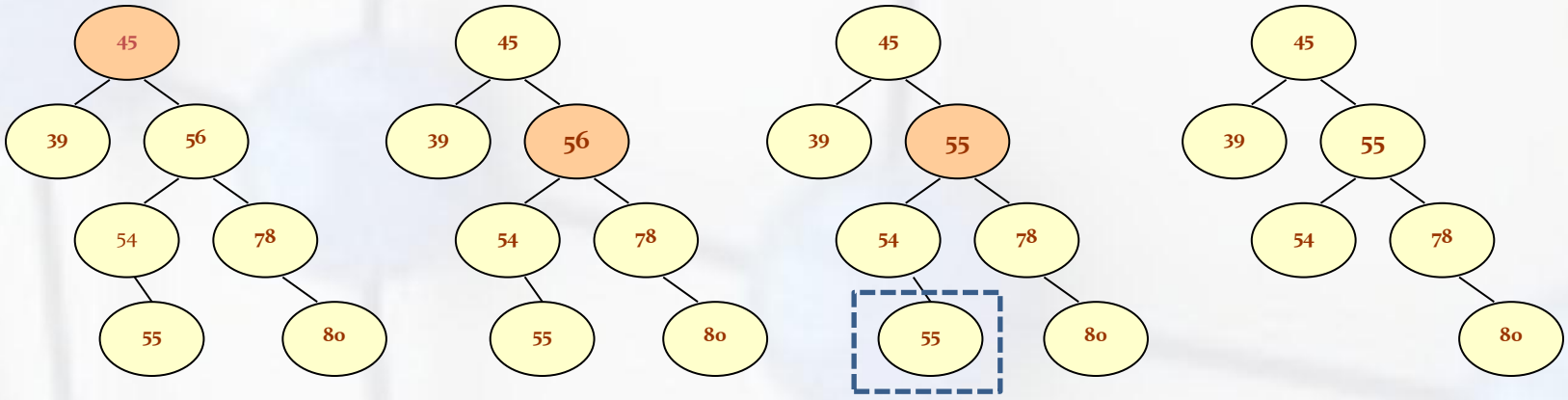
- ***Case 2:*** **Deleting a node with one child (either left or right).**
- To handle the deletion, the node's child is set to be the child of the node's parent.
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.
- For example, deleting node 54 in the tree below.



Deleting a Value from a BST

***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 cases 1 or 2.
- For example, deleting node 56 in the tree below.

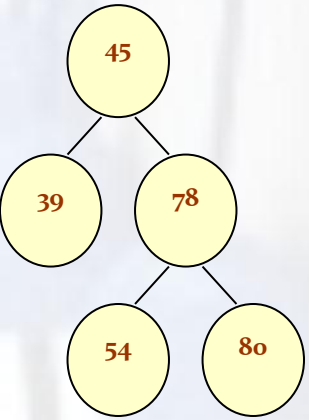


Algorithm to Delete from a BST

```
Delete(TREE, VAL):
    IF TREE = NULL, then
          Write "VAL not found in the tree"
    ELSE IF VAL < TREE->DATA
          TREE->LEFT = Delete(TREE->LEFT, VAL)
    ELSE IF VAL > TREE->DATA
          TREE->RIGHT = Delete(TREE->RIGHT, VAL)
    ELSE IF TREE->LEFT!= NULL AND TREE->RIGHT != NULL //Victim node has 2 children
          SET TEMP = findLargestNode(TREE->LEFT)
          SET TREE->DATA = TEMP->DATA
          TREE->LEFT = Delete(TREE->LEFT, TEMP->DATA)
    ELSE       // Victim node has one child(left or right) or no child.
          SET TEMP = TREE
          IF TREE->LEFT == NULL AND TREE ->RIGHT == NULL  // Leaf node to delete
                SET TREE = NULL
          ELSE IF TREE->LEFT != NULL       // Only Left child
                SET TREE = TREE->LEFT
          ELSE                             // Only right child
                SET TREE = TREE->RIGHT
          [END OF IF]
        FREE TEMP
      [END OF IF]
      RETURN TREE
Step 2: End
```

Determining the Height of a BST

- In order to determine the height of a BST, we will calculate the height of the left and right sub-trees. Whichever height is greater, 1 is added to it.
- Since height of right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1= 1 + 1 = 2




```
Height (TREE):
   Step 1: IF TREE = NULL, then
              Return -1    // Height of null tree is assumed as -1
           ELSE
              SET LeftHeight = Height(TREE->LEFT)
              SET RightHeight = Height(TREE->RIGHT)
              IF LeftHeight > RightHeight
                     Return LeftHeight + 1
              ELSE
                     Return RightHeight + 1
              [END OF IF]
           [END OF IF]
   Step 2: End
```

Determining the Number of Nodes

- To calculate the total number of elements/nodes in a BST, we will count the number of nodes in the left sub-tree and the right sub-tree.
- *Number of nodes = totalNodes(left sub-tree) + total Nodes(right sub-tree) + 1*

```
totalNodes (TREE)

Step 1: IF TREE = NULL, then
            Return 0
        ELSE
            Return totalNodes(TREE->LEFT)  +
                        totalNodes(TREE->RIGHT) + 1
        [END OF IF]
Step 2: End
```

Determining the Number of Internal Nodes

- To calculate the total number of elements/nodes in a BST, we will count the number of nodes having at-least one child.

```
totalInternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        [END OF IF]
        IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 0
        ELSE
            Return totalInternalNodes(TREE -> LEFT) +
            totalInternalNodes(TREE -> RIGHT) + 1
       [END OF IF]
Step 2: END
```

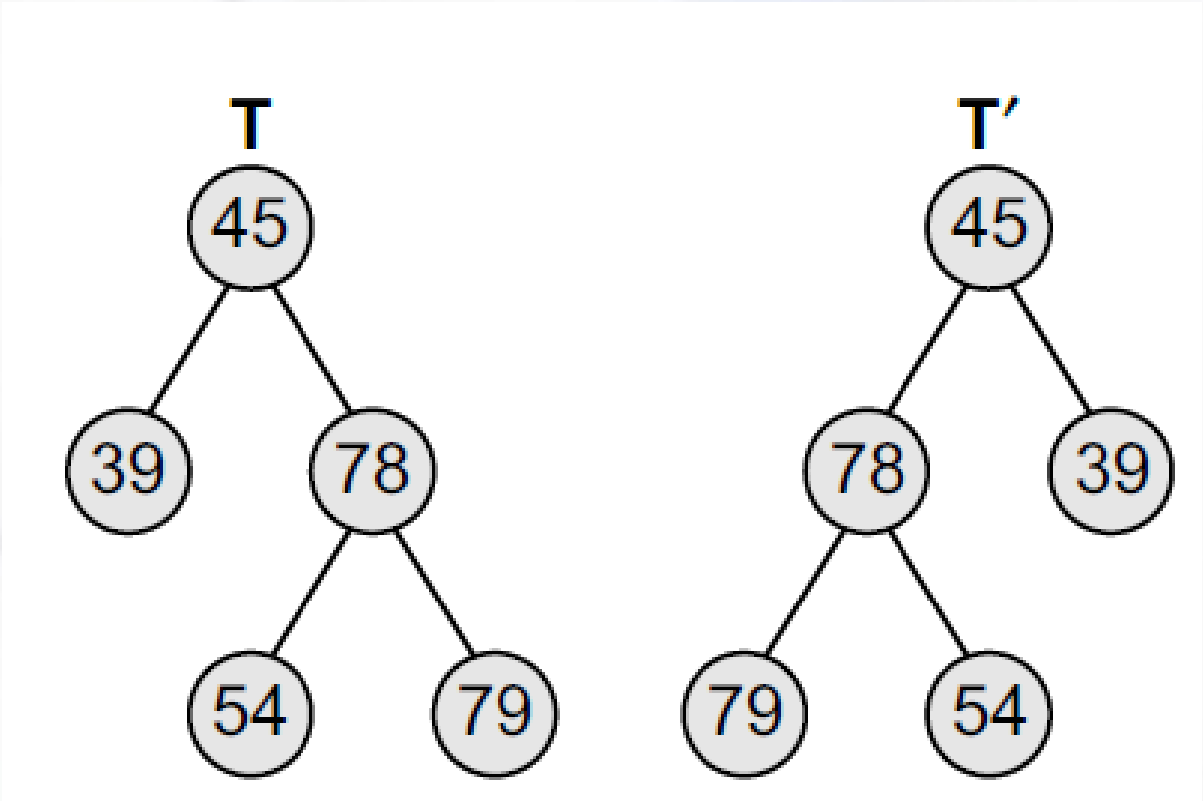
Determining the Number of External/Leaf Nodes

- To calculate the total number of elements/nodes in a BST, we will count the number of nodes having no children.

```
totalExternalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 1
         ELSE
            Return totalExternalNodes(TREE -> LEFT) +
            totalExternalNodes(TREE -> RIGHT)
         [END OF IF]
Step 2: END
```

Mirror Tree

- Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree.



Mirror Tree - Algorithm

- Mirror image of a binary search tree is 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
```

Figure 10.23 Algorithm to obtain the mirror image mirror image T′ of a binary search tree

Delete a BST

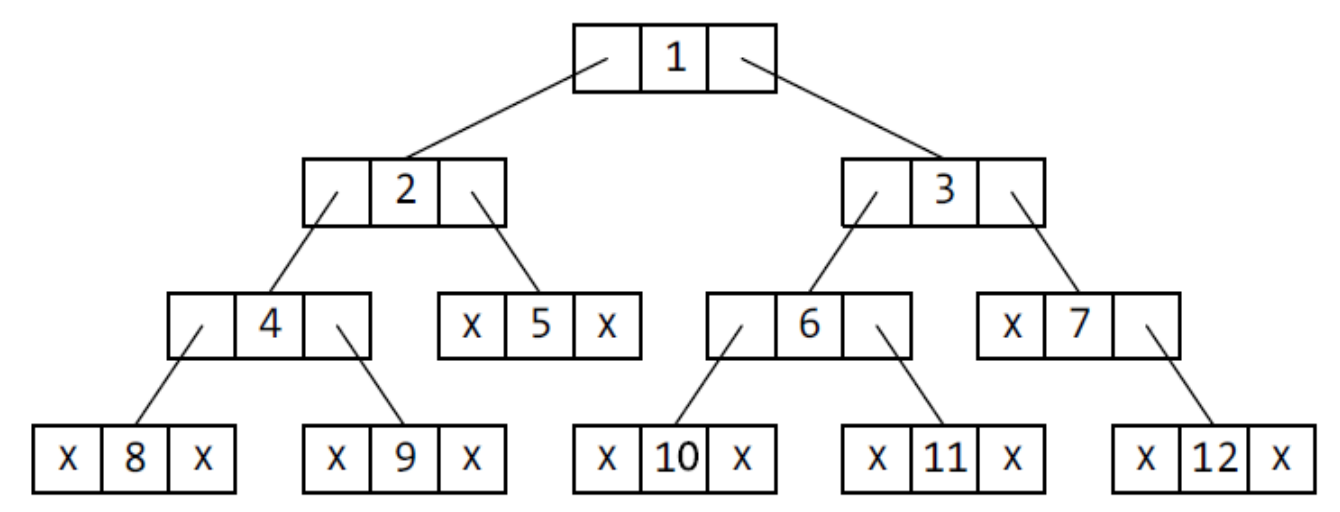
```
deleteTree(TREE)

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

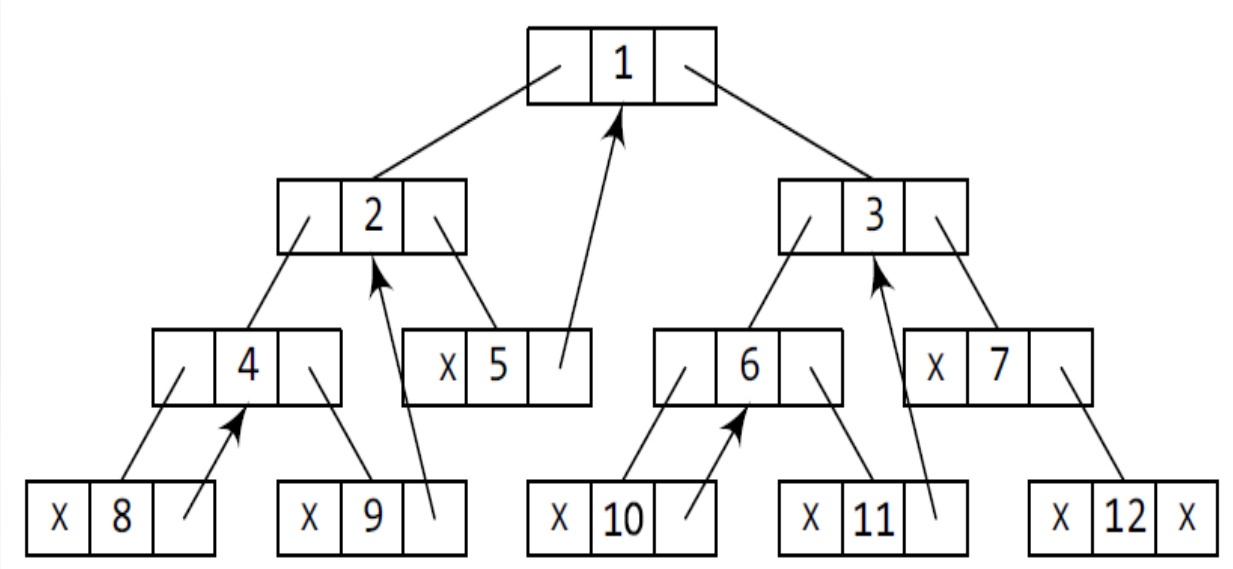
Figure 10.24 Alogrithm to delete a binary search tree

Threaded Binary Trees

- A threaded binary tree is same as that of a binary tree but with a difference in storing NULL pointers.
- In the linked representation of a BST, a number of nodes contain a NULL pointer either in their left or right fields or in both. This space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information.
- For example, the NULL entries can be replaced to store a pointer to the **in-order predecessor**, or the **in-order successor** of the node. These special pointers are called ***threads*** and binary trees containing threads are called ***threaded trees***. In **the** linked representation of a threaded binary tree, threads will be denoted using dotted lines.

Threaded Binary Trees

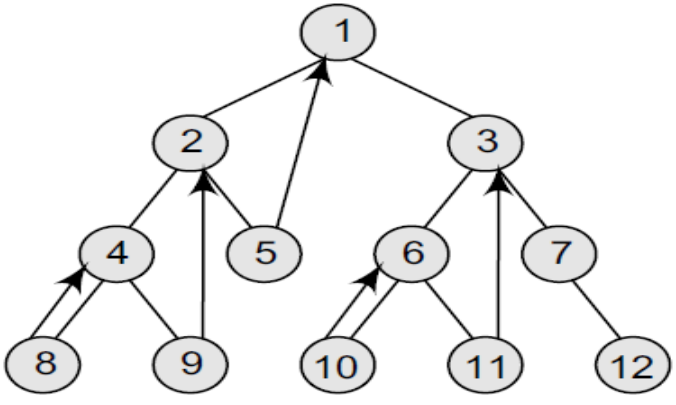


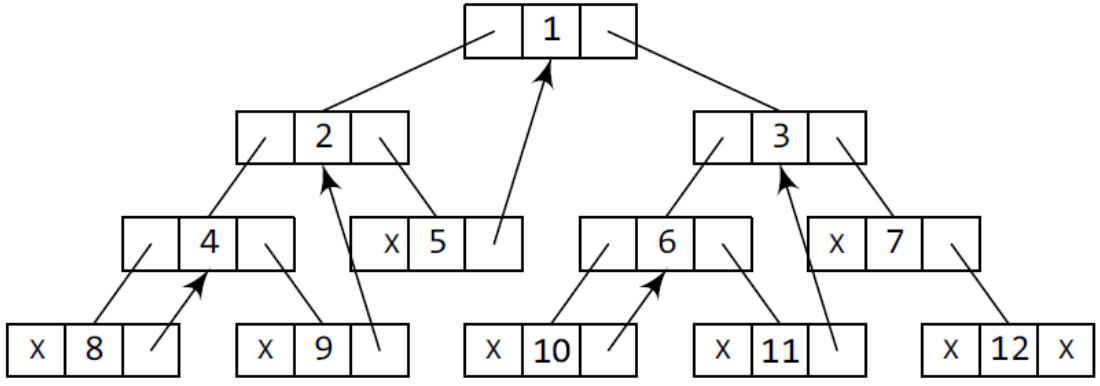


Threaded Binary Trees

- In one way threading, a thread will appear either in the right field or the left field of the node.
- If the thread appears in the **left field**, then it points to the **in-order predecessor** of the node. Such a one way threaded tree is called a **left threaded binary tree**.
- If the thread appears in the right field, then it will point to the **in-order successor** of the node. Such a one way threaded tree is called a **right-threaded binary tree.**

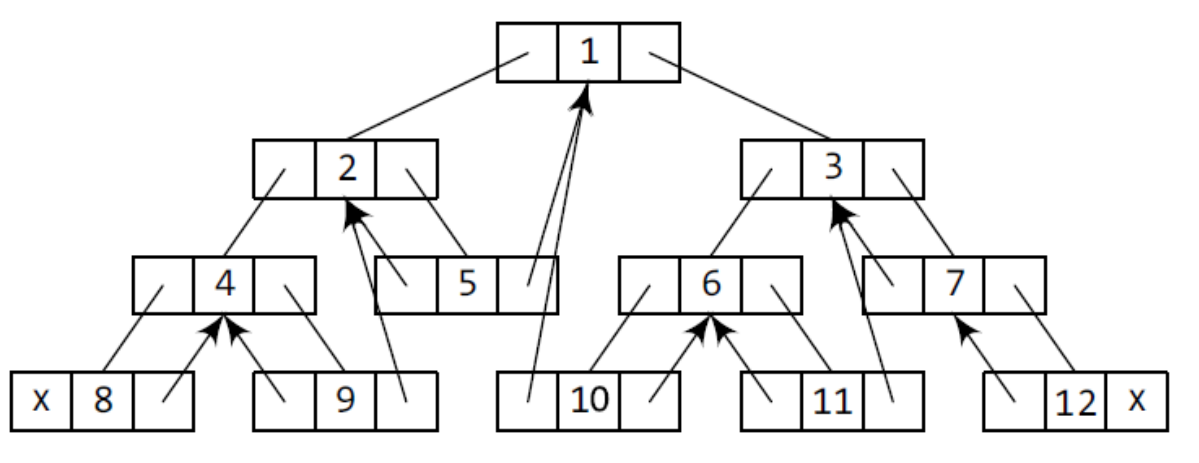
Binary tree with one way threading





Threaded Binary Trees

- In a **two-way threaded tree**, also called a double threaded tree, threads will appear in both the left and right fields of the node.
- While the left field will point to the **in-order predecessor** of the node, the right field will point to its **in-order successor**.
- A two-way threaded binary tree is also called a fully threaded binary tree.



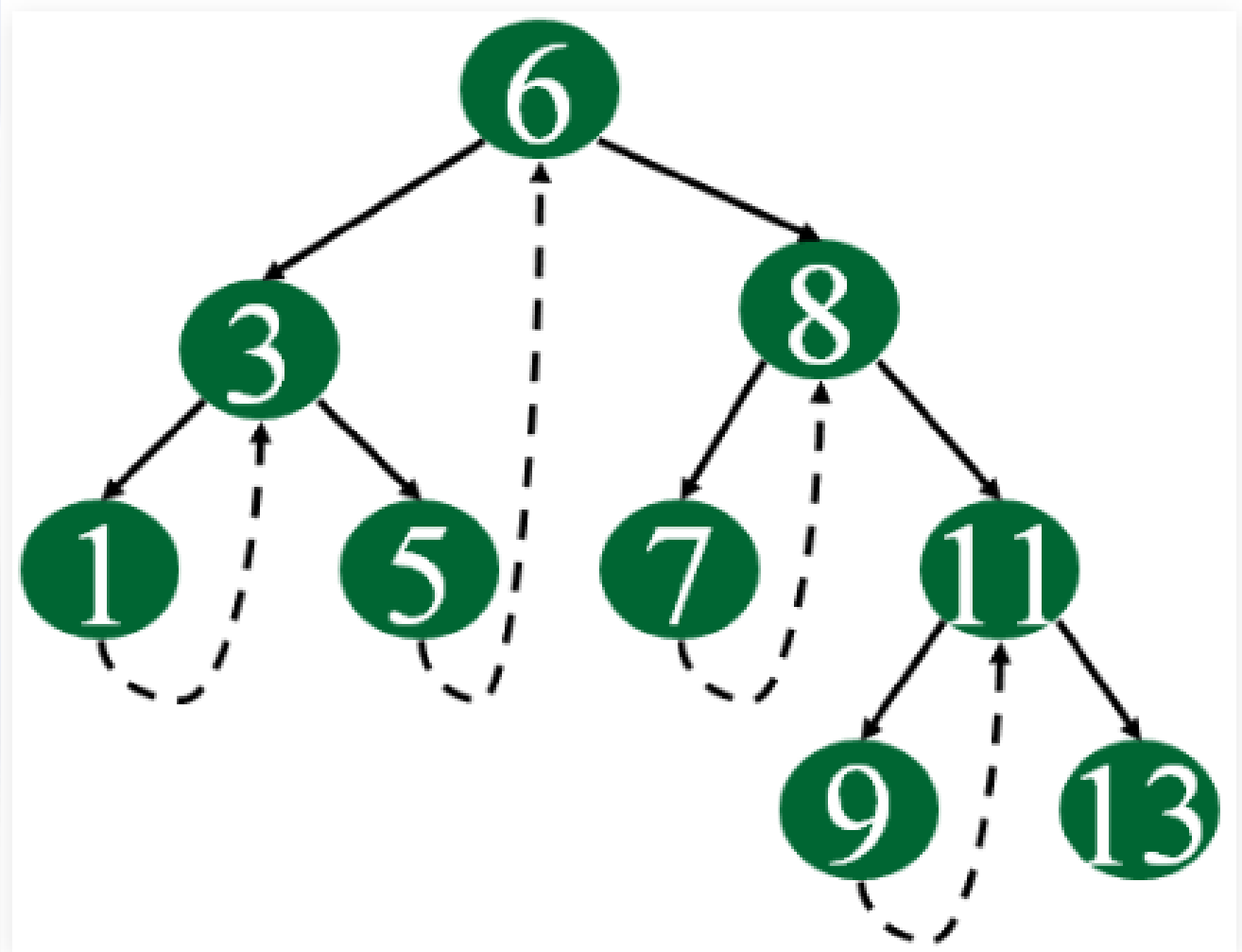
Binary tree with two-way threading

Threaded Binary Trees

Node structure of a Right-threaded Binary Tree

```
struct node
{
    int val;
    struct node *right;
    struct node *left;
    int rthread;
};
```

Threaded Binary Tree



Source: GeeksForGeeks.org

AVL Trees

- AVL tree is a **self-balancing binary search tree** in which the heights of the two sub-trees of a node may differ by at most one. Because of this property, AVL tree is also known as a **height-balanced tree**.
- Named after its inventors **A**delson-**V**elsky and **L**andis.

- The key advantage of using an AVL tree is that it **takes O(log n)** time to perform **search**, **insertion** and **deletion** operations in average case as well as worst case (because the height of the tree is limited to O(logn)).

- The structure of an AVL tree is same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the ***BalanceFactor***.

AVL Trees

- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

  Balance factor = Height (left sub-tree) – Height (right sub-tree)

- A binary search tree in which every node has a balance factor of -1, 0 or 1 is said to be height balanced.

- A node with any other balance factor is considered to be unbalanced and requires rebalancing.

AVL Trees

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called ***Left-heavy tree***.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called **Right-*heavy tree***.

AVL Trees

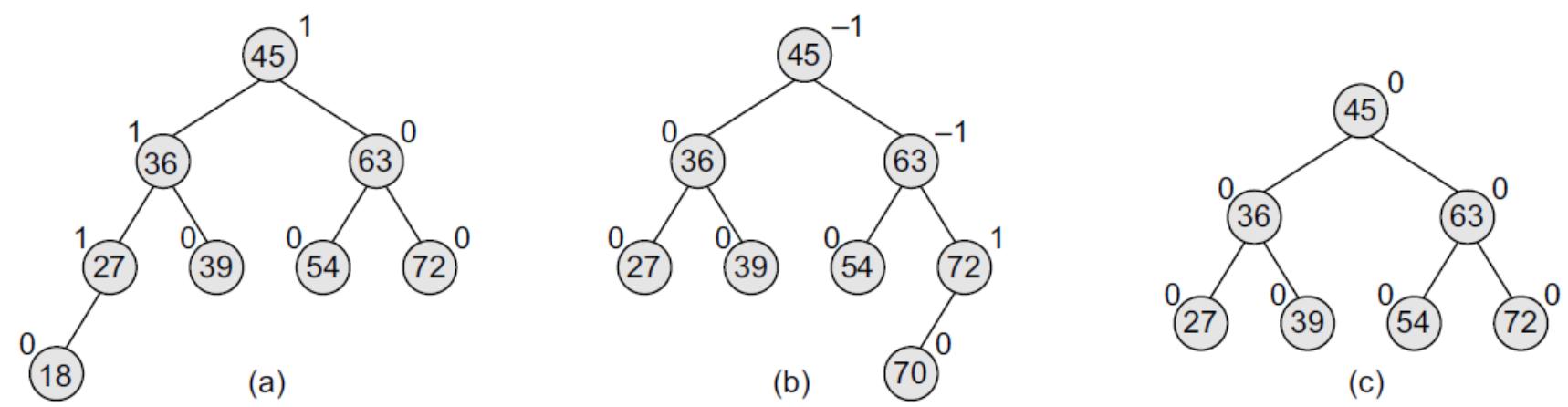


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Searching for a Node in an AVL Tree

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes O(log *n*) time to complete.