CHAPTER 10

Efficient Binary Trees

LEARNING OBJECTIVE

In this chapter, we will discuss efficient binary trees such as binary search trees, AVL trees, threaded binary trees, red-black trees, and splay trees. This chapter is an extension of binary trees.

10.1 BINARY SEARCH TREES

We have already discussed binary trees in the previous chapter. A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. 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. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)

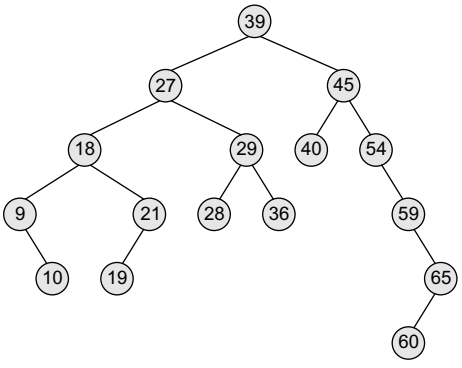


Figure 10.1 Binary search tree

Look at Fig. 10.1. 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. For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced. Whenever we search for an element, we do not

need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in. For example, in the given tree, if we have to search for 29, then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value). The left sub-tree has a root node with the value 27. Since 29 is greater than 27, we will move to the right sub-tree, where we will find the element. Thus, the average running time of a search operation is $O(\log_2 n)$, as at every step, we eliminate half of the sub-tree from the search process. Due to its efficiency in searching elements, binary search trees are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

Binary search trees also speed up the insertion and deletion operations. The tree has a speed advantage when the data in the structure changes rapidly.

Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists. In a sorted array, searching can be done in $O(\log_2 n)$ time, but insertions and deletions are quite expensive. In contrast, inserting and deleting elements in a linked list is easier, but searching for an element is done in $O(n)$ time.

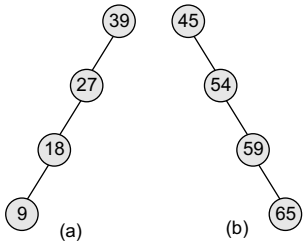


Figure 10.2 (a) Left skewed, and (b) right skewed binary search trees

However, in the worst case, a binary search tree will take $O(n)$ time to search for an element. The worst case would occur when the tree is a linear chain of nodes as given in Fig. 10.2.

To summarize, a binary search tree is a binary tree with the following properties:

- 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 N's value.
- Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

Example 10.1 State whether the binary trees in Fig. 10.3 are binary search trees or not.

Solution

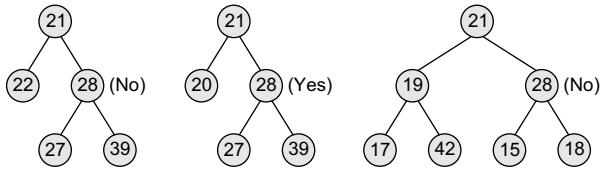
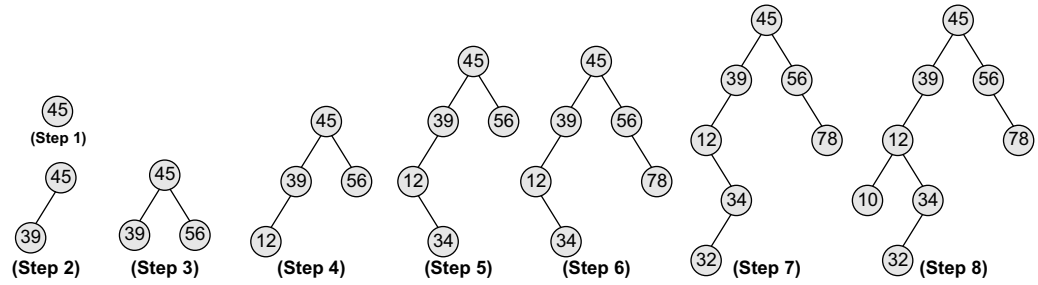


Figure 10.3 Binary trees

Example 10.2 Create a binary search tree using the following data elements:

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

Solution



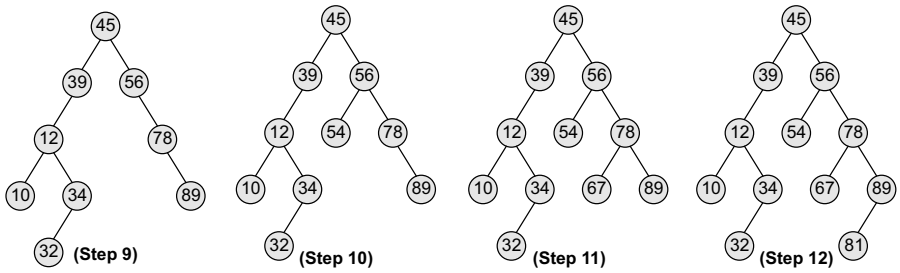


Figure 10.4 Binary search tree

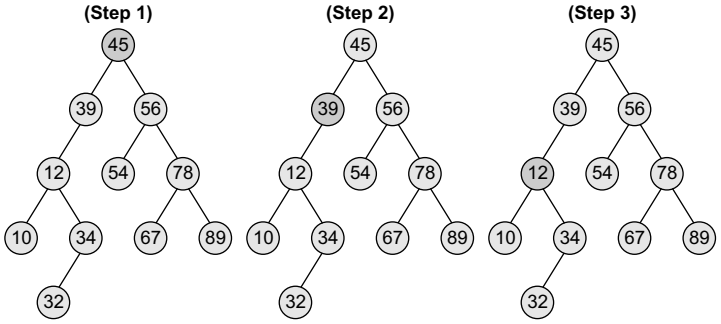


Figure 10.5 Searching a node with value 12 in the given binary search tree

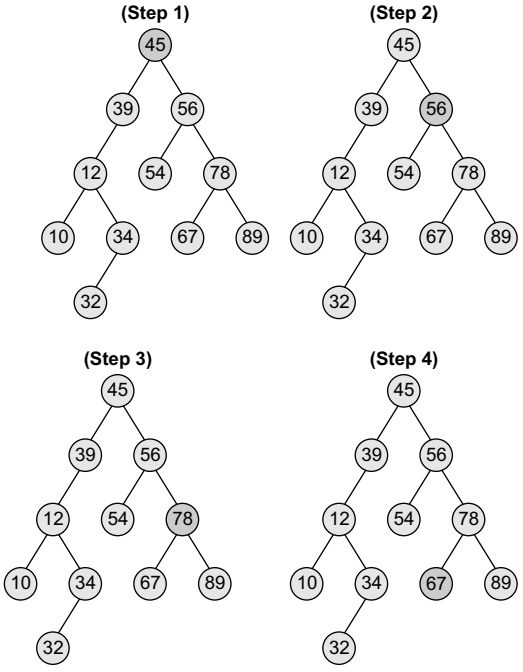


Figure 10.6 Searching a node with value 67 in the given binary search tree

10.2 OPERATIONS ON BINARY SEARCH TREES

In this section, we will discuss the different operations that are performed on a binary search tree. All these operations require comparisons to be made between the nodes.

10.2.1 Searching for a Node in a Binary Search Tree

The search function is used to find whether a given value is present in the tree or not. 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.

Look at Fig. 10.5. The figure shows how a binary tree is searched to find a specific element. First, see how the tree will be traversed to find the node with value 12. The procedure to find the node with value 67 is illustrated in Fig. 10.6.

The procedure to find the node with value 40 is shown in Fig. 10.7. The search would terminate after reaching node 39 as it does not have any right child.

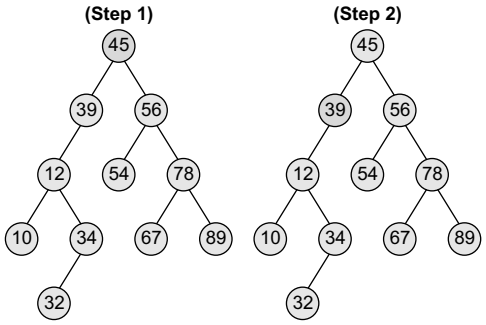


Figure 10.7 Searching a node with the value 40 in the given binary search tree

Now let us look at the algorithm to search for an element in the binary search tree as shown in Fig. 10.8. In Step 1, we check if the value stored at the current node of TREE is equal to VAL or if the current node is NULL, then we return the current node of TREE. Otherwise, if the value stored at the current node is less than VAL, then the algorithm is recursively called on its right sub-tree, else the algorithm is called on its left sub-tree.

10.2.2 Inserting a New Node in a Binary Search Tree

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. Figure 10.9 shows the algorithm to insert a given value in a binary search tree.

The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

```
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
```

Figure 10.8 Algorithm to search for a given value in a binary search tree

```
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
```

Figure 10.9 Algorithm to insert a given value in a 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. That is, if the new node's value is greater than that of the parent node, the new node is inserted in the right sub-tree, else it is inserted in the left sub-tree. The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

Look at Fig. 10.10 which shows insertion of values in a given tree. We will take up the case of inserting 12 and 55.

10.2.3 Deleting a Node from a Binary Search Tree

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not

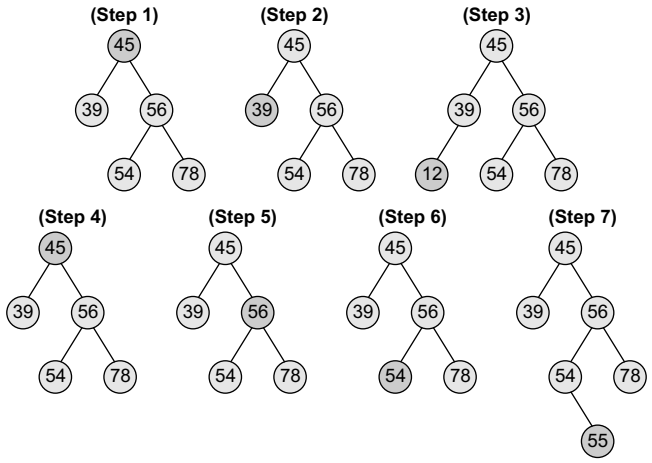


Figure 10.10 Inserting nodes with values 12 and 55 in the given binary search tree

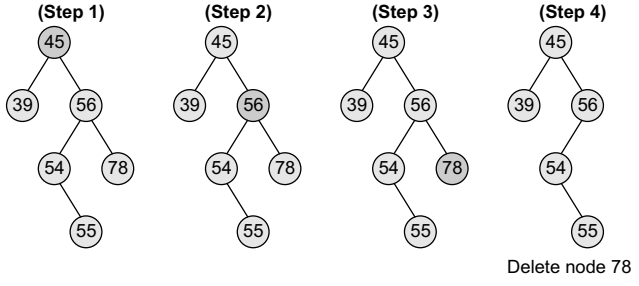


Figure 10.11 Deleting node 78 from the given binary search tree

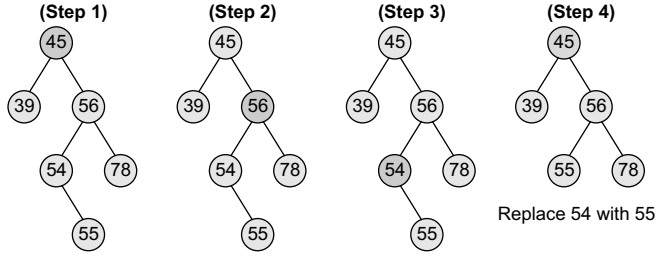


Figure 10.12 Deleting node 54 from the given binary search tree

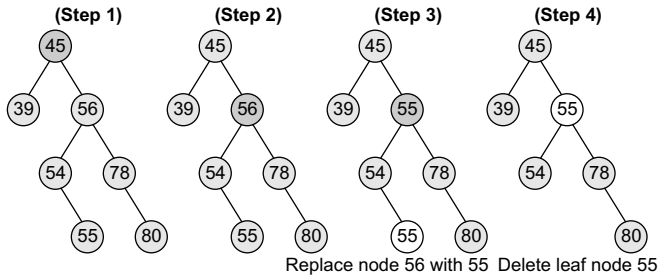


Figure 10.13 Deleting node 56 from the given binary search tree

lost in the process. We will take up three cases in this section and discuss how a node is deleted from a binary search tree.

Case 1: Deleting a Node that has No Children

Look at the binary search tree given in Fig. 10.11. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.

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. Look at the binary search tree shown in Fig. 10.12 and see how deletion of node 54 is handled.

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. Look at the binary search tree given in Fig. 10.13 and see how deletion of node with value 56 is handled.

This deletion could also be handled by replacing node 56 with its in-order successor, as shown in Fig. 10.14.

Now, let us look at Fig. 10.15 which shows the algorithm to delete a node from a binary search tree.

In Step 1 of the algorithm, we first check if `TREE=NULL`, because if it is true, then the node to be deleted is not present in the tree. However, if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree, otherwise the algorithm

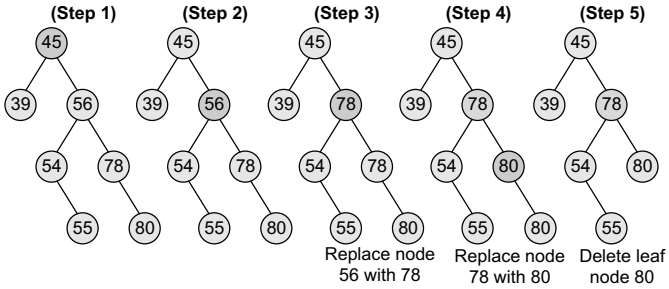


Figure 10.14 Deleting node 56 from the given binary search tree

```
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
```

Figure 10.15 Algorithm to delete a node from a binary search tree

is called recursively on the node's right sub-tree.

Note that if we have found the node whose value is equal to VAL, then we check which case of deletion it is. If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling `findLargestNode(TREE -> LEFT)` and replace the current node's value with that of its in-order predecessor. Then, we call `Delete(TREE -> LEFT, TEMP -> DATA)` to delete the initial node of the in-order predecessor. Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion.

If the node to be deleted does not have any child, then we simply set the node to NULL. Last but not the least, if the node to be deleted has either a left or a right child but not both, then the current node is replaced by its child node and the initial child node is deleted from the tree.

The delete function requires time proportional to the height of the tree in the worst case. It takes $O(\log n)$ time to execute in the average case and $\Omega(n)$ time in the worst case.

10.2.4 Determining the Height of a Binary Search Tree

In order to determine the height of a binary search tree, we calculate the height of the left sub-tree and the right sub-tree. Whichever height is greater, 1 is added to it. For example, if the height of the left sub-tree is greater than that of the right sub-tree, then 1 is added to the left sub-tree, else 1 is added to the right sub-tree.

Look at Fig. 10.16. Since the height of the right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1= 2 + 1 = 3.

Figure 10.17 shows a recursive algorithm that determines the height of a binary search tree.

In Step 1 of the algorithm, we first check if the current node of the `TREE = NULL`. If the condition is true, then 0 is returned to the calling code. Otherwise, for every node, we recursively call the algorithm to calculate the height of its left sub-tree as well as its right sub-tree. The height of the tree at that node is given by adding 1 to the height of the left sub-tree or the height of right sub-tree, whichever is greater.

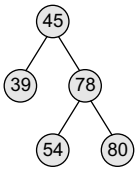


Figure 10.16 Binary search tree with height = 3

10.2.5 Determining the Number of Nodes

Determining the number of nodes in a binary search tree is similar to determining its height. To calculate the total number of elements/nodes

```
Height (TREE)

Step 1: IF TREE = NULL
            Return 0
        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
```

Figure 10.17 Algorithm to determine the height of a binary search tree

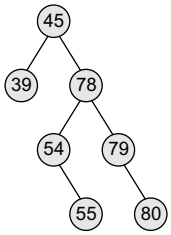


Figure 10.18 Binary search tree

in the tree, we count the number of nodes in the left sub-tree and the right sub-tree.

```
Number of nodes = totalNodes(left sub-tree)
+ totalNodes(right sub-tree) + 1
```

Consider the tree given in Fig. 10.18. The total number of nodes in the tree can be calculated as:

```
Total nodes of left sub-tree = 1
Total nodes of left sub-tree = 5
Total nodes of tree = (1 + 5) + 1
                    = 7
```

Figure 10.19 shows a recursive algorithm to calculate the number of nodes in a binary search tree. For every node, we recursively call the algorithm on its left sub-tree as well as the right sub-tree. The total number of nodes at a given node is then returned by adding 1 to the number of nodes in its left as well as right sub-tree. However if the tree is empty, that is `TREE = NULL`, then the number of nodes will be zero.

Determining the Number of Internal Nodes

To calculate the total number of internal nodes or non-leaf nodes, we count the number of internal nodes in the left sub-tree and the right sub-tree and add 1 to it (1 is added for the root node).

```
Number of internal nodes =
  totalInternalNodes(left sub-tree) +
totalInternalNodes(right sub-tree) + 1
```

Consider the tree given in Fig. 10.18. The total number of internal nodes in the tree can be calculated as:

```
Total internal nodes of left sub-tree = 0
Total internal nodes of right sub-tree = 3
Total internal nodes of tree = (0 + 3) + 1
                             = 4
```

Figure 10.20 shows a recursive algorithm to calculate the total number of internal nodes in a binary search tree. For every node, we recursively call the algorithm on its left sub-tree as well as the right sub-tree. The total number of internal nodes at a given node is then returned by adding internal nodes in its left as well as right sub-tree. However, if the tree is empty, that is `TREE = NULL`, then the number of internal nodes will be zero. Also if there is only one node in the tree, then the number of internal nodes will be zero.

```
totalNodes(TREE)

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

Figure 10.19 Algorithm to calculate the number of nodes in a binary search tree

```
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
```

Figure 10.20 Algorithm to calculate the total number of internal nodes in a binary search tree

Determining the Number of External Nodes

To calculate the total number of external nodes or leaf nodes, we add the number of

external nodes in the left sub-tree and the right sub-tree. However if the tree is empty, that is TREE = NULL, then the number of external nodes will be zero. But if there is only one node in the tree, then the number of external nodes will be one.

```
Number of external nodes = totalExternalNodes(left sub-tree) +
                           totalExternalNodes (right sub-tree)
```

Consider the tree given in Fig. 10.18. The total number of external nodes in the given tree can be calculated as:

```
Total external nodes of left sub-tree = 1
Total external nodes of left sub-tree = 2
Total external nodes of tree = 1 + 2
                             = 3
```

```
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
```

Figure 10.21 Algorithm to calculate the total number of external nodes in a binary search tree

Figure 10.21 shows a recursive algorithm to calculate the total number of external nodes in a binary search tree. For every node, we recursively call the algorithm on its left sub-tree as well as the right sub-tree. The total number of external nodes at a given node is then returned by adding the external nodes in its left as well as right sub-tree. However if the tree is empty, that is TREE = NULL, then the number of external nodes will be zero. Also if there is only one node in the tree, then there will be only one external node (that is the root node).

10.2.6 Finding the Mirror Image of a Binary Search Tree

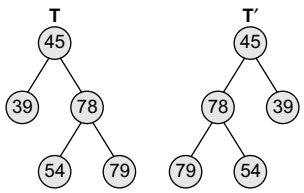


Figure 10.22 Binary search tree T and its

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. For example, given a tree T, the mirror image of T can be obtained as T′. Consider the tree T given in Fig. 10.22.

Figure 10.23 shows a recursive algorithm to obtain the mirror image of a binary search tree. In the algorithm, if TREE != NULL, that is if the current node in the tree has one or more nodes, then the algorithm is recursively called at every node in the tree to swap the nodes in its left and right sub-trees.

```
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

10.2.7 Deleting a Binary Search Tree

To delete/remove an entire binary search tree from the memory, we first delete the elements/nodes in the left sub-tree and then delete the nodes in the right sub-tree. The algorithm shown in Fig. 10.24 gives a recursive procedure to remove the binary search tree.

10.2.8 Finding the Smallest Node in a Binary Search Tree

The very basic property of the binary search tree states that the smaller value will occur in the left sub-tree. If

the left sub-tree is NULL, then the value of the root node will be smallest as compared to the nodes in the right sub-tree. So, to find the node with the smallest value, we find the value of the leftmost node of the left sub-tree. The recursive algorithm to find the smallest node in a binary search tree is shown in Fig. 10.25.

```
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

```
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

10.2.9 Finding the Largest Node in a Binary Search Tree

To find the node with the largest value, we find the value of the rightmost node of the right sub-tree. However, if the right sub-tree is empty, then the root node will be the largest value in the tree. The recursive algorithm to find the largest node in a binary search tree is shown in Fig. 10.26.

```
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

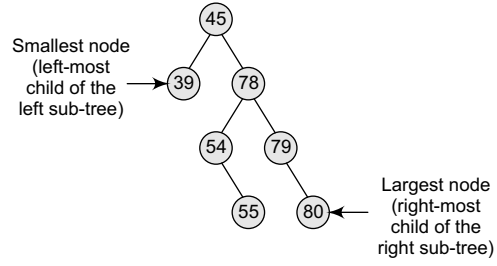


Figure 10.27 Binary search tree