

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.)

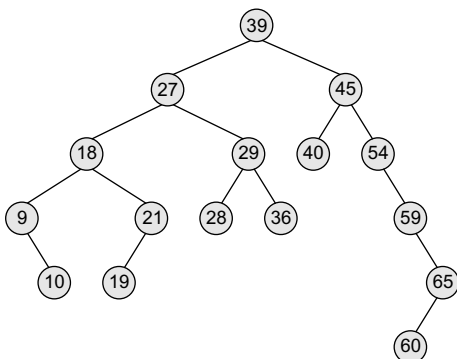


Figure 10.1 Binary search tree

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.

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.

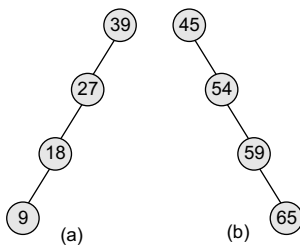


Figure 10.2 (a) Left skewed, and (b) right skewed binary search trees

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

Solution

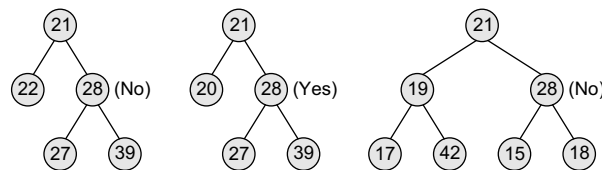
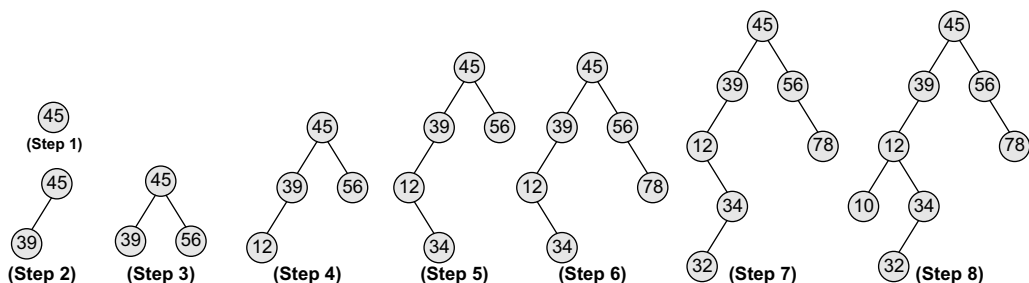


Figure 10.3 Binary trees

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



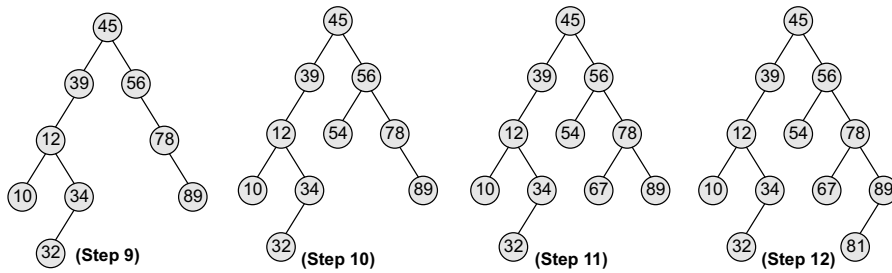


Figure 10.4 Binary search tree

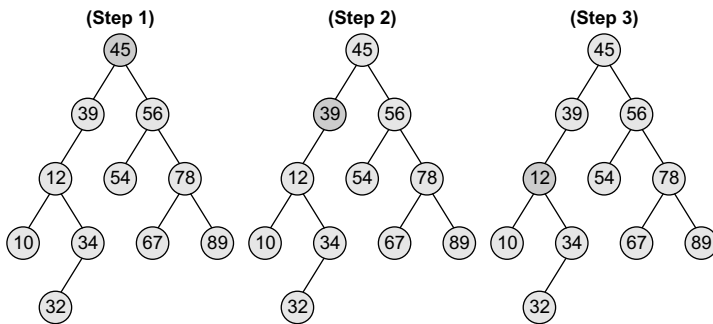


Figure 10.5 Searching a node with value 12 in the given binary search tree

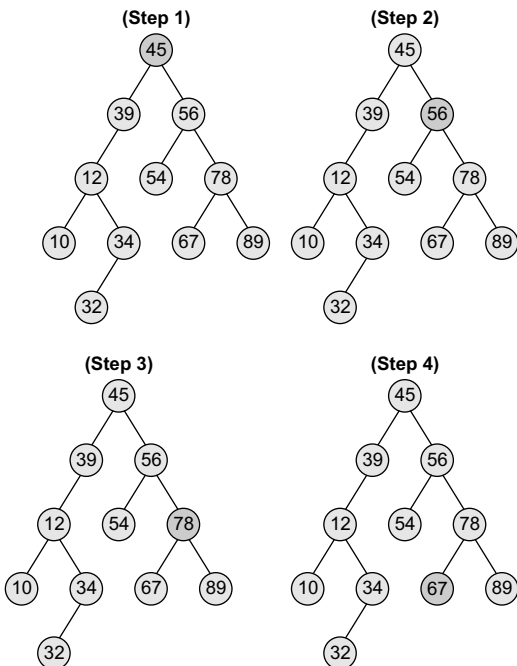


Figure 10.6 Searching a node with value 67 in the given binary search tree

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.

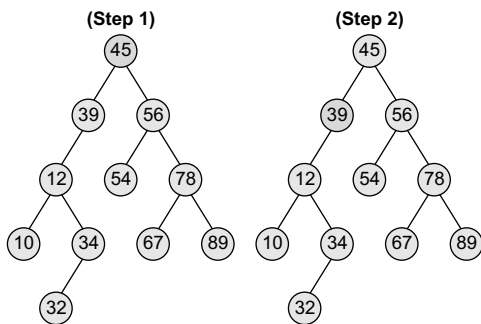


Figure 10.7 Searching a node with the value 40 in the given binary search tree

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

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

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

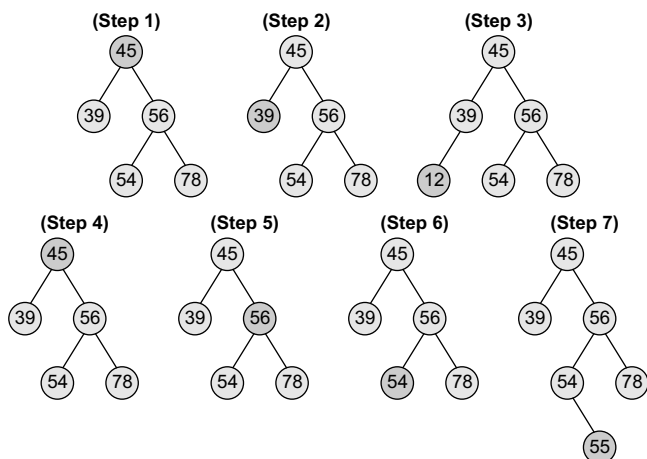


Figure 10.10 Inserting nodes with values 12 and 55 in the given binary search tree

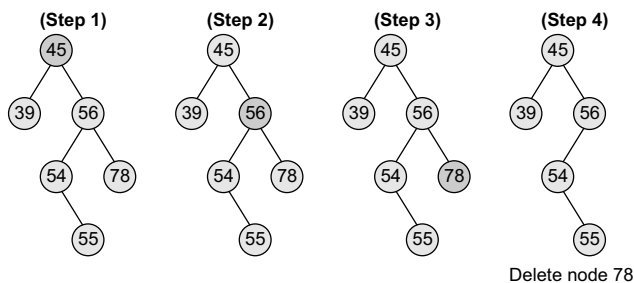


Figure 10.11 Deleting node 78 from the given binary search tree

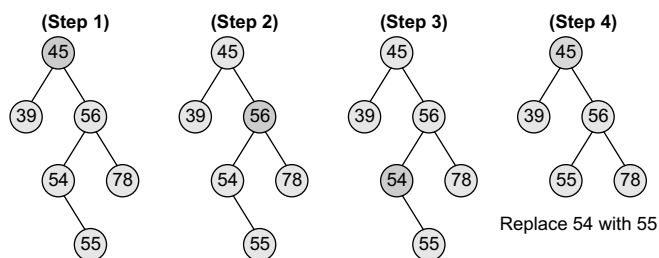


Figure 10.12 Deleting node 54 from the given binary search tree

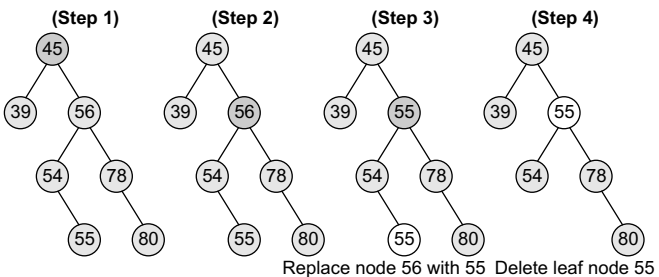


Figure 10.13 Deleting node 56 from the given binary search tree

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 $TREE = \text{NULL}$, 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

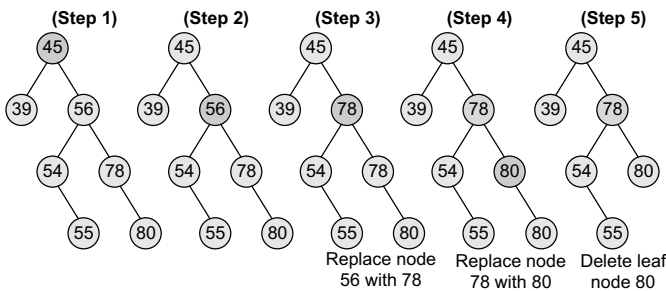


Figure 10.14 Deleting node 56 from the given binary search tree

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

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.

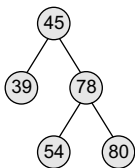


Figure 10.16 Binary search tree with height = 3

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.

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

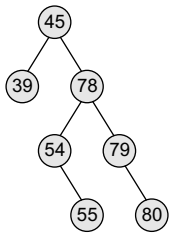


Figure 10.18 Binary search tree

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

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 right 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.

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

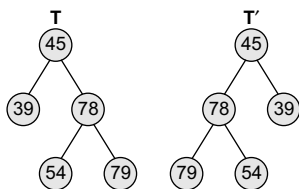


Figure 10.22 Binary search tree T and its mirror image T'

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 $TREE \neq NULL$, 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 Algorithm to delete a binary search tree

findSmallestElement(TREE)

```
Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
        Return 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

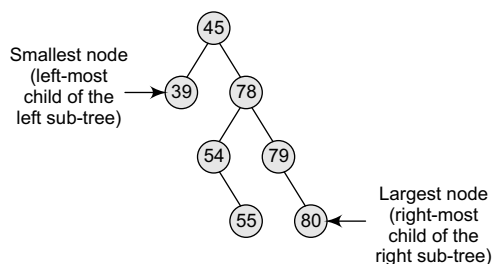


Figure 10.27 Binary search tree

Consider the tree given in Fig. 10.27. The smallest and the largest node can be given as:

PROGRAMMING EXAMPLE

1. Write a program to create a binary search tree and perform all the operations discussed in the preceding sections.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
struct node *tree;
void create_tree(struct node *);
struct node *insertElement(struct node *, int);
void preorderTraversal(struct node *);
void inorderTraversal(struct node *);
```

```

void postorderTraversal(struct node *);
struct node *findSmallestElement(struct node *);
struct node *findLargestElement(struct node *);
struct node *deleteElement(struct node *, int);
struct node *mirrorImage(struct node *);
int totalNodes(struct node *);
int totalExternalNodes(struct node *);
int totalInternalNodes(struct node *);
int Height(struct node *);
struct node *deleteTree(struct node *);
int main()
{
    int option, val;
    struct node *ptr;
    create_tree(tree);
    clrscr();
    do
    {
        printf("\n *****MAIN MENU***** \n");
        printf("\n 1. Insert Element");
        printf("\n 2. Preorder Traversal");
        printf("\n 3. Inorder Traversal");
        printf("\n 4. Postorder Traversal");
        printf("\n 5. Find the smallest element");
        printf("\n 6. Find the largest element");
        printf("\n 7. Delete an element");
        printf("\n 8. Count the total number of nodes");
        printf("\n 9. Count the total number of external nodes");
        printf("\n 10. Count the total number of internal nodes");
        printf("\n 11. Determine the height of the tree");
        printf("\n 12. Find the mirror image of the tree");
        printf("\n 13. Delete the tree");
        printf("\n 14. Exit");
        printf("\n\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
            case 1:
                printf("\n Enter the value of the new node : ");
                scanf("%d", &val);
                tree = insertElement(tree, val);
                break;
            case 2:
                printf("\n The elements of the tree are : \n");
                preorderTraversal(tree);
                break;
            case 3:
                printf("\n The elements of the tree are : \n");
                inorderTraversal(tree);
                break;
            case 4:
                printf("\n The elements of the tree are : \n");
                postorderTraversal(tree);
                break;
            case 5:
                ptr = findSmallestElement(tree);
                printf("\n Smallest element is : %d", ptr->data);
                break;
            case 6:
                ptr = findLargestElement(tree);
                printf("\n Largest element is : %d", ptr->data);
                break;
            case 7:
                printf("\n Enter the element to be deleted : ");
                scanf("%d", &val);

```

```

        tree = deleteElement(tree, val);
        break;
    case 8:
        printf("\n Total no. of nodes = %d", totalNodes(tree));
        break;
    case 9:
        printf("\n Total no. of external nodes = %d",
            totalExternalNodes(tree));
        break;
    case 10:
        printf("\n Total no. of internal nodes = %d",
            totalInternalNodes(tree));
        break;
    case 11:
        printf("\n The height of the tree = %d", Height(tree));
        break;
    case 12:
        tree = mirrorImage(tree);
        break;
    case 13:
        tree = deleteTree(tree);
        break;
    }
    }while(option!=14);
    getch();
    return 0;
}

void create_tree(struct node *tree)
{
    tree = NULL;
}

struct node *insertElement(struct node *tree, int val)
{
    struct node *ptr, *nodeptr, *parentptr;
    ptr = (struct node*)malloc(sizeof(struct node));
    ptr->data = val;
    ptr->left = NULL;
    ptr->right = NULL;
    if(tree==NULL)
    {
        tree=ptr;
        tree->left=NULL;
        tree->right=NULL;
    }
    else
    {
        parentptr=NULL;
        nodeptr=tree;
        while(nodeptr!=NULL)
        {
            parentptr=nodeptr;
            if(val<nodeptr->data)
                nodeptr=nodeptr->left;
            else
                nodeptr = nodeptr->right;
        }
        if(val<parentptr->data)
            parentptr->left = ptr;
        else
            parentptr->right = ptr;
    }
    return tree;
}

void preorderTraversal(struct node *tree)
{

```

```

        if(tree != NULL)
        {
            printf("%d\t", tree->data);
            preorderTraversal(tree->left);
            preorderTraversal(tree->right);
        }
    }
    void inorderTraversal(struct node *tree)
    {
        if(tree != NULL)
        {
            inorderTraversal(tree->left);
            printf("%d\t", tree->data);
            inorderTraversal(tree->right);
        }
    }
    void postorderTraversal(struct node *tree)
    {
        if(tree != NULL)
        {
            postorderTraversal(tree->left);
            postorderTraversal(tree->right);
            printf("%d\t", tree->data);
        }
    }
    struct node *findSmallestElement(struct node *tree)
    {
        if( (tree == NULL) || (tree->left == NULL))
            return tree;
        else
            return findSmallestElement(tree->left);
    }
    struct node *findLargestElement(struct node *tree)
    {
        if( (tree == NULL) || (tree->right == NULL))
            return tree;
        else
            return findLargestElement(tree->right);
    }
    struct node *deleteElement(struct node *tree, int val)
    {
        struct node *cur, *parent, *suc, *psuc, *ptr;
        if(tree->left==NULL)
        {
            printf("\n The tree is empty ");
            return(tree);
        }
        parent = tree;
        cur = tree->left;
        while(cur!=NULL && val!= cur->data)
        {
            parent = cur;
            cur = (val<cur->data)? cur->left:cur->right;
        }
        if(cur == NULL)
        {
            printf("\n The value to be deleted is not present in the tree");
            return(tree);
        }
        if(cur->left == NULL)
            ptr = cur->right;
        else if(cur->right == NULL)

```

```

        ptr = cur->left;
    else
    {
        // Find the in-order successor and its parent
        psuc = cur;
        cur = cur->left;
        while(suc->left!=NULL)
        {
            psuc = suc;
            suc = suc->left;
        }
        if(cur==psuc)
        {
            // Situation 1
            suc->left = cur->right;
        }
        else
        {
            // Situation 2
            suc->left = cur->left;
            psuc->left = suc->right;
            suc->right = cur->right;
        }
        ptr = suc;
    }
    // Attach ptr to the parent node
    if(parent->left == cur)
        parent->left=ptr;
    else
        parent->right=ptr;
    free(cur);
    return tree;
}

int totalNodes(struct node *tree)
{
    if(tree==NULL)
        return 0;
    else
        return(totalNodes(tree->left) + totalNodes(tree->right) + 1);
}

int totalExternalNodes(struct node *tree)
{
    if(tree==NULL)
        return 0;
    else if((tree->left==NULL) && (tree->right==NULL))
        return 1;
    else
        return (totalExternalNodes(tree->left) +
            totalExternalNodes(tree->right));
}

int totalInternalNodes(struct node *tree)
{
    if( (tree==NULL) || ((tree->left==NULL) && (tree->right==NULL)))
        return 0;
    else
        return (totalInternalNodes(tree->left)
            + totalInternalNodes(tree->right) + 1);
}

int Height(struct node *tree)
{
    int leftheight, rightheight;

```

```

        if(tree==NULL)
            return 0;
        else
        {
            leftheight = Height(tree->left);
            rightheight = Height(tree->right);
            if(leftheight > rightheight)
                return (leftheight + 1);
            else
                return (rightheight + 1);
        }
    }
}
struct node *mirrorImage(struct node *tree)
{
    struct node *ptr;
    if(tree!=NULL)
    {
        mirrorImage(tree->left);
        mirrorImage(tree->right);
        ptr=tree->left;
        ptr->left = ptr->right;
        tree->right = ptr;
    }
}
struct node *deleteTree(struct node *tree)
{
    if(tree!=NULL)
    {
        deleteTree(tree->left);
        deleteTree(tree->right);
        free(tree);
    }
}

```

Output

```

*****MAIN  MENU*****
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Find the smallest element
6. Find the largest element
7. Delete an element
8. Count the total number of nodes
9. Count the total number of external nodes
10. Count the total number of internal nodes
11. Determine the height of the tree
12. Find the mirror image of the tree
13. Delete the tree
14. Exit
Enter your option : 1
Enter the value of the new node : 1
Enter the value of the new node : 2
Enter the value of the new node : 4
Enter your option : 3
2 1 4
Enter your option : 14

```

10.3 THREADED BINARY TREES

A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers. Consider the linked representation of a binary tree as given in Fig. 10.28.

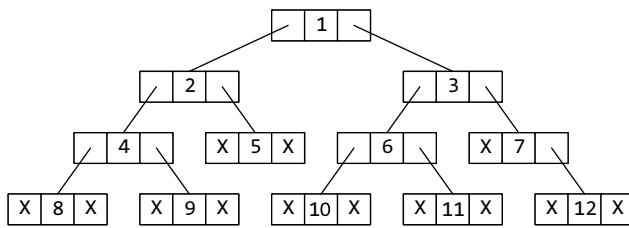


Figure 10.28 Linked representation of a binary tree

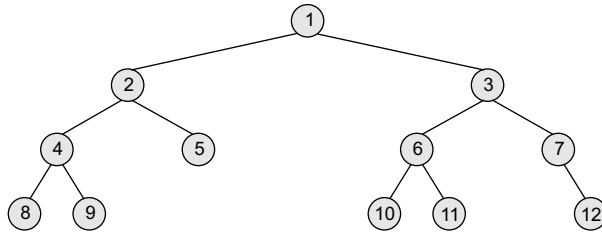


Figure 10.29 (a) Binary tree without threading

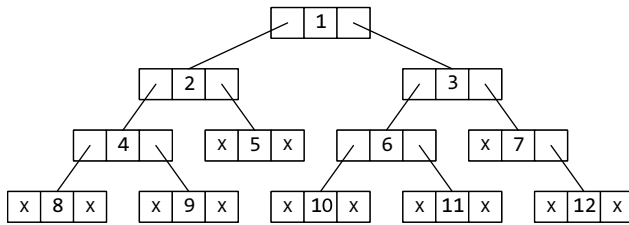


Figure 10.29 (b) Linked representation of the binary tree (without threading)

will point to the in-order successor of the node. Such a one-way threaded tree is called a right-threaded binary tree.

In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node. While the left field will point to the in-order predecessor of the node, the right field will point to its successor. A two-way threaded binary tree is also called a fully threaded binary tree. One-way threading and two-way threading of binary trees are explained below. Figure 10.29 shows a binary tree without threading and its corresponding linked representation. The in-order traversal of the tree is given as 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12

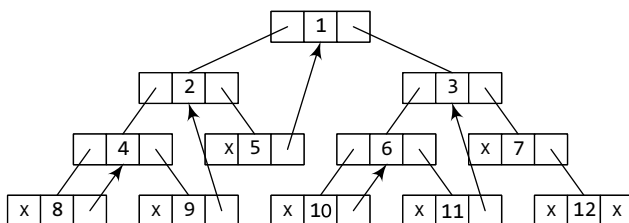


Figure 10.30 (a) Linked representation of the binary tree with one-way threading

In the linked representation, 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 arrows.

There are many ways of threading a binary tree and each type may vary according to the way the tree is traversed. In this book, we will discuss in-order traversal of the tree. Apart from this, a threaded binary tree may correspond to one-way threading or a two-way threading.

In one-way threading, a thread will appear either in the right field or the left field of the node. A one-way threaded tree is also called a single-threaded tree. If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node. Such a one-way threaded tree is called a left-threaded binary tree. On the contrary, if the thread appears in the right field, then it

One-way Threading

Figure 10.30 shows a binary tree with one-way threading and its corresponding linked representation.

Node 5 contains a NULL pointer in its RIGHT field, so it will be replaced to point to node 1, which is its in-order successor. Similarly, the RIGHT field of node 8 will point to node 4, the RIGHT field of node 9 will point to node 2, the RIGHT field of node 10 will point to node

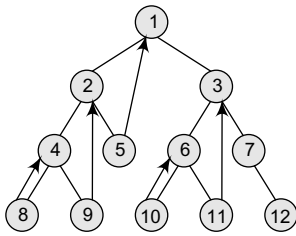


Figure 10.30 (b) Binary tree with one-way threading

6, the RIGHT field of node 11 will point to node 3, and the RIGHT field of node 12 will contain NULL because it has no in-order successor.

Two-way Threading

Figure 10.31 shows a binary tree with two-way threading and its corresponding linked representation.

Node 5 contains a NULL pointer in its LEFT field, so it will be replaced to point to node 2, which is its in-order predecessor. Similarly, the LEFT field of node 8 will contain NULL because it has no in-order predecessor, the LEFT field of node 7 will point to node 3, the LEFT field of node 9 will point to node 4, the LEFT field of node 10 will point to node 1, the LEFT field of node 11 will contain 6, and the LEFT field of node 12 will point to node 7.

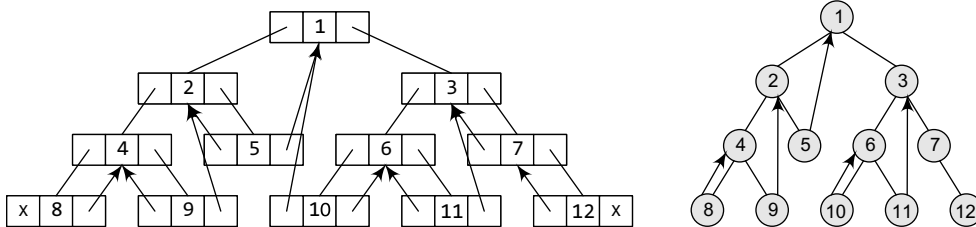


Figure 10.31 (a) Linked representation of the binary tree with threading, (b) binary tree with two-way threading

Now, let us look at the memory representation of a binary tree without threading, with one-way threading, and with two-way threading. This is illustrated in Fig. 10.32.

	LEFT	DATA	RIGHT
ROOT 1	-1	8	-1
2	-1	10	-1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	-1	7	18
12	-1	9	-1
13			
14	-1	5	-1
15			
16	-1	11	-1
17			
18	-1	12	-1
19			
20	2	6	16

(a)

	LEFT	DATA	RIGHT
ROOT 1	-1	8	9
2	-1	10	20
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	-1	7	18
12	-1	9	5
13			
14	-1	5	3
15			
16	-1	11	8
17			
18	-1	12	-1
19			
20	2	6	16

(b)

	LEFT	DATA	RIGHT
ROOT 1	-1	8	9
2	3	10	20
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	8	7	18
12	9	9	5
13			
14	5	5	3
15			
16	20	11	8
17			
18	11	12	-1
19			
20	2	6	16

(c)

Figure 10.32 Memory representation of binary trees: (a) without threading, (b) with one-way, and (c) two-way threading

10.3.1 Traversing a Threaded Binary Tree

For every node, visit the left sub-tree first, provided if one exists and has not been visited earlier. Then the node (root) itself is followed by visiting its right sub-tree (if one exists). In case there is no right sub-tree, check for the threaded link and make the threaded node the current node in consideration. The algorithm for in-order traversal of a threaded binary tree is given in Fig. 10.33.

- Step 1:** Check if the current node has a left child that has not been visited. If a left child exists that has not been visited, go to Step 2, else go to Step 3.
- Step 2:** Add the left child in the list of visited nodes. Make it as the current node and then go to Step 6.
- Step 3:** If the current node has a right child, go to Step 4 else go to Step 5.
- Step 4:** Make that right child as current node and go to Step 6.
- Step 5:** Print the node and if there is a threaded node make it the current node.
- Step 6:** If all the nodes have visited then END else go to Step 1.

Figure 10.33 Algorithm for in-order traversal of a threaded binary tree

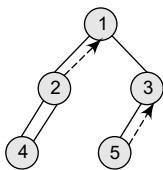


Figure 10.34 Threaded binary tree

Let's consider the threaded binary tree given in Fig. 10.34 and traverse it using the algorithm.

1. Node 1 has a left child i.e., 2 which has not been visited. So, add 2 in the list of visited nodes, make it as the current node.
2. Node 2 has a left child i.e., 4 which has not been visited. So, add 4 in the list of visited nodes, make it as the current node.
3. Node 4 does not have any left or right child, so print 4 and check for its threaded link. It has a threaded link to node 2, so make node 2 the current node.
4. Node 2 has a left child which has already been visited. However, it does not have a right child. Now, print 2 and follow its threaded link to node 1. Make node 1 the current node.
5. Node 1 has a left child that has been already visited. So print 1. Node 1 has a right child 3 which has not yet been visited, so make it the current node.
6. Node 3 has a left child (node 5) which has not been visited, so make it the current node.
7. Node 5 does not have any left or right child. So print 5. However, it does have a threaded link which points to node 3. Make node 3 the current node.
8. Node 3 has a left child which has already been visited. So print 3.
9. Now there are no nodes left, so we end here. The sequence of nodes printed is—4 2 1 5 3.

Advantages of Threaded Binary Tree

- It enables linear traversal of elements in the tree.
- Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
- It enables to find the parent of a given element without explicit use of parent pointers.
- Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.

Thus, we see the basic difference between a binary tree and a threaded binary tree is that in binary trees a node stores a NULL pointer if it has no child and so there is no way to traverse back.

PROGRAMMING EXAMPLE

2. Write a program to implement simple right in-threaded binary trees.

```
#include <stdio.h>
#include <conio.h>
struct tree
{
    int val;
    struct tree *right;
    struct tree *left;
    int thread;
};
struct tree *root = NULL;
struct tree* insert_node(struct tree *root, struct tree *ptr, struct tree *rt)
{
    if(root == NULL)
    {
        root = ptr;
        if(rt != NULL)
        {
            root->right = rt;
            root->thread = 1;
        }
    }
    else if(ptr->val < root->val)
        root->left = insert_node(root->left, ptr, root);
    else
    {
        if(root->thread == 1)
        {
            root->right = insert_node(NULL, ptr, rt);
            root->thread=0;
        }
        else
            root->right = insert_node(root->right, ptr, rt);
    }
    return root;
}
struct tree* create_threaded_tree()
{
    struct tree *ptr;
    int num;
    printf("\n Enter the elements, press -1 to terminate ");
    scanf("%d", &num);
    while(num != -1)
    {
        ptr = (struct tree*)malloc(sizeof(struct tree));
        ptr->val = num;
        ptr->left = ptr->right = NULL;
        ptr->thread = 0;
        root = insert_node(root, ptr, NULL);
        printf(" \n Enter the next element ");
        fflush(stdin);
        scanf("%d", &num);
    }
    return root;
}
void inorder(struct tree *root)
{
    struct tree *ptr = root, *prev;
    do
    {
        while(ptr != NULL)
        {
            prev = ptr;
            ptr = ptr->left;
        }
    }
}
```

```

    }
    if(prev != NULL)
    {
        printf(" %d", prev->val);
        ptr = prev->right;
        while(prev != NULL && prev->thread)
        {
            printf(" %d", ptr->val);
            prev = ptr;
            ptr = ptr->right;
        }
    }
    }while(ptr != NULL);
}
void main()
{
    // struct tree *root=NULL;
    clrscr();
    create_threaded_tree();
    printf(" \n The in-order traversal of the tree can be given as : ");
    inorder(root);
    getch();
}

```

Output

```

Enter the elements, press -1 to terminate 5
Enter the next element 8
Enter the next element 2
Enter the next element 3
Enter the next element 7
Enter the next element -1
The in-order traversal of the tree can be given as:
2      3      5      7      8

```

10.4 AVL TREES

AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. The tree is named AVL in honour of its inventors. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree. The key advantage of using an AVL tree is that it takes $O(\log n)$ time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to $O(\log n)$.

The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the **BalanceFactor**. Thus, every node has a balance factor associated with it. 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. 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 of the tree.

$$\text{Balance factor} = \text{Height (left sub-tree)} - \text{Height (right sub-tree)}$$

- If the balance factor of a node is 1 , then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a *left-heavy tree*.
- If the balance factor of a node is 0 , then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the 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 therefore called as a *right-heavy tree*.

Look at Fig. 10.35. Note that the nodes 18, 39, 54, and 72 have no children, so their balance factor = 0 . Node 27 has one left child and zero right child. So, the height of left sub-tree = 1 , whereas the height of right sub-tree = 0 . Thus, its balance factor = 1 . Look at node 36, it has a left

sub-tree with height = 2, whereas the height of right sub-tree = 1. Thus, its balance factor = $2 - 1 = 1$. Similarly, the balance factor of node 45 = $3 - 2 = 1$; and node 63 has a balance factor of 0 ($1 - 1$).

Now, look at Figs 10.35 (a) and (b) which show a right-heavy AVL tree and a balanced AVL tree.

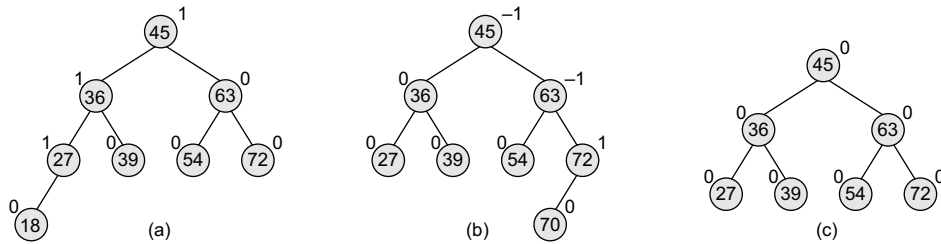


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

The trees given in Fig. 10.35 are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or -1. However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done. The tree is rebalanced by performing rotation at the critical node. There are four types of rotations: LL rotation, RR rotation, LR rotation, and RL rotation. The type of rotation that has to be done will vary depending on the particular situation. In the following section, we will discuss insertion, deletion, searching, and rotations in AVL trees.

10.4.1 Operations on AVL Trees

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. Due to the height-balancing of the tree, the search operation takes $O(\log n)$ time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

Inserting a New Node in an AVL Tree

Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still -1, 0, or 1, then rotations are not required.

During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path between the root of the tree and the newly inserted node. The possible changes which may take place in any node on the path are as follows:

- Initially, the node was either left- or right-heavy and after insertion, it becomes balanced.
- Initially, the node was balanced and after insertion, it becomes either left- or right-heavy.
- Initially, the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree, thereby creating an unbalanced sub-tree. Such a node is said to be a *critical node*.

Consider the AVL tree given in Fig. 10.36.

If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case. Look at the tree given in Fig. 10.37 which shows the tree after inserting node 30.

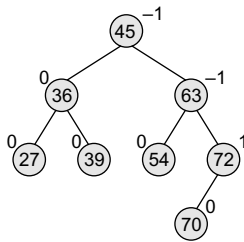


Figure 10.36 AVL tree

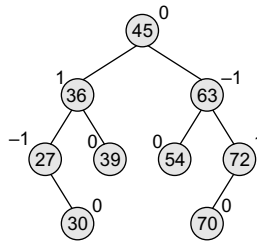


Figure 10.37 AVL tree after inserting a node with the value 30

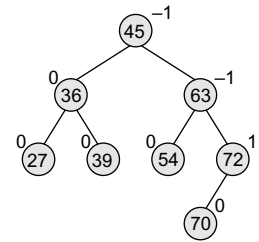


Figure 10.38 AVL tree

Let us take another example to see how insertion can disturb the balance factors of the nodes and how rotations are done to restore the AVL property of a tree. Look at the tree given in Fig. 10.38.

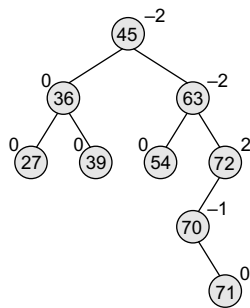


Figure 10.39 AVL tree after inserting a node with the value 71

After inserting a new node with the value 71, the new tree will be as shown in Fig. 10.39. Note that there are three nodes in the tree that have their balance factors 2, -2, and -2, thereby disturbing the *AVLness* of the tree. So, here comes the need to perform rotation. To perform rotation, our first task is to find the critical node. Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither -1, 0, nor 1. In the tree given above, the critical node is 72. The second task in rebalancing the tree is to determine which type of rotation has to be done. There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node. The four categories of rotations are:

- **LL rotation** The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
- **RR rotation** The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
- **LR rotation** The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
- **RL rotation** The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

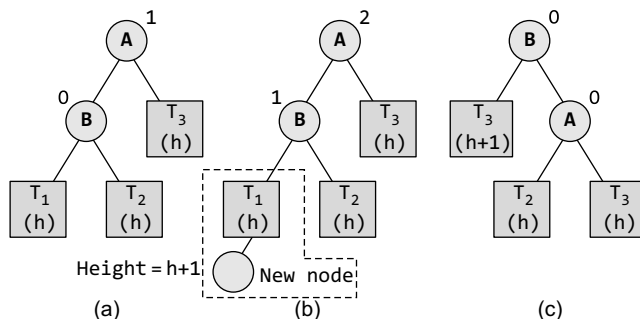


Figure 10.40 LL rotation in an AVL tree

LL Rotation

Let us study each of these rotations in detail. First, we will see where and how LL rotation is applied. Consider the tree given in Fig. 10.40 which shows an AVL tree.

Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not -1, 0, or 1), so we apply LL rotation as shown in tree (c).

Note The new node has now become a part of tree T_1 .

While rotation, node \mathbf{B} becomes the root, with τ_1 and \mathbf{A} as its left and right child. τ_2 and τ_3 become the left and right sub-trees of \mathbf{A} .

Example 10.3 Consider the AVL tree given in Fig. 10.41 and insert 18 into it.

Solution

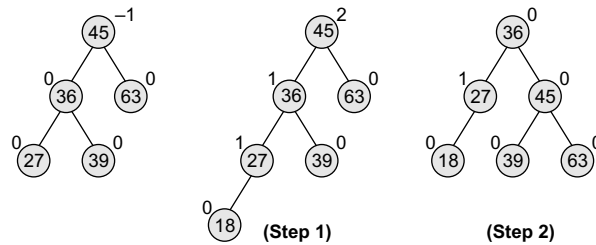


Figure 10.41 AVL tree

RR Rotation

Let us now discuss where and how RR rotation is applied. Consider the tree given in Fig. 10.42 which shows an AVL tree.

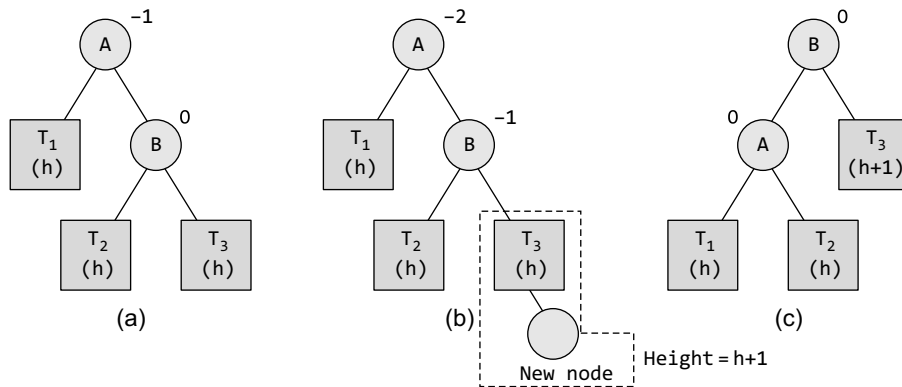


Figure 10.42 RR rotation in an AVL tree

Example 10.4 Consider the AVL tree given in Fig. 10.43 and insert 89 into it.

Solution

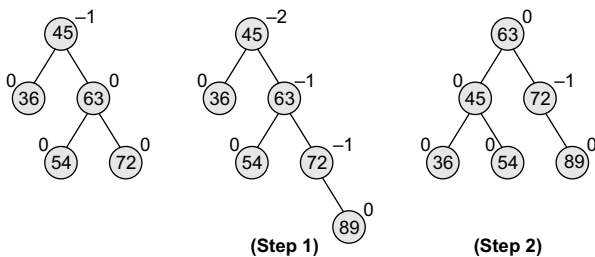


Figure 10.43 AVL tree

Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the right sub-tree of the critical node \mathbf{A} (node \mathbf{A} is the critical node because it is the closest ancestor whose balance factor is not -1 , 0 , or 1), so we apply RR rotation as shown in tree (c). Note that the new node has now become a part of tree τ_3 .

While rotation, node \mathbf{B} becomes the root, with \mathbf{A} and τ_3 as its left and right child. τ_1 and τ_2 become the left and right sub-trees of \mathbf{A} .

LR and RL Rotations

Consider the AVL tree given in Fig. 10.44 and see how LR rotation is done to rebalance the tree.

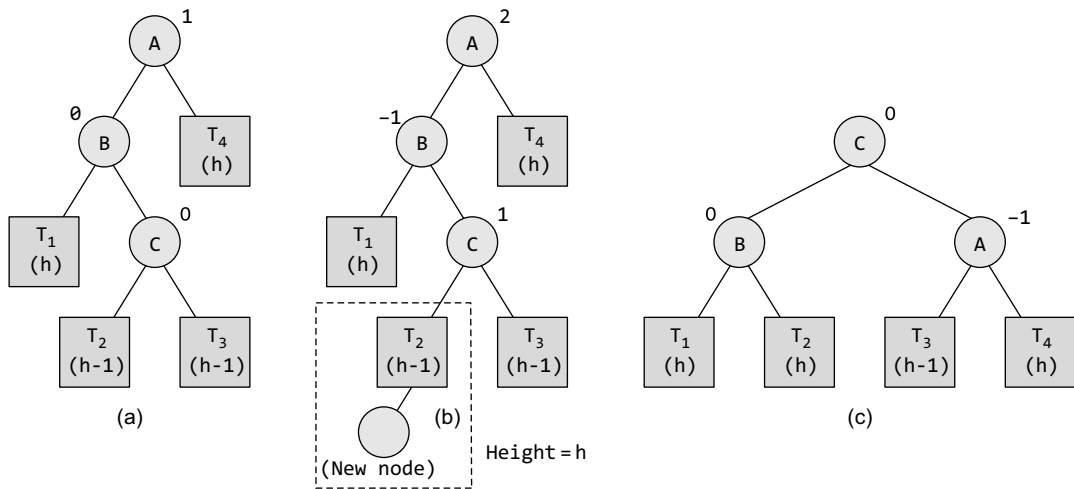


Figure 10.44 LR rotation in an AVL tree

Example 10.5 Consider the AVL tree given in Fig. 10.45 and insert 37 into it.

Solution

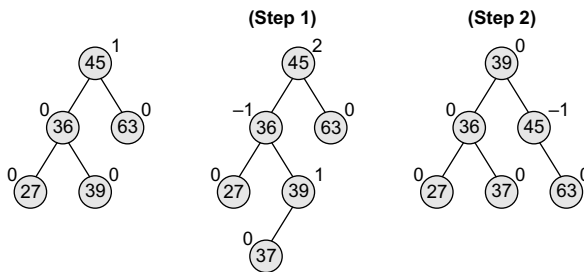


Figure 10.45 AVL tree

Tree (a) is an AVL tree. In tree (b), a new node is inserted in the right sub-tree of the left sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not -1 , 0 or 1), so we apply LR rotation as shown in tree (c). Note that the new node has now become a part of tree T_2 .

While rotation, node c becomes the root, with B and A as its left and right children. Node B has T_1 and T_2 as its left and right sub-trees and T_3 and T_4 become the left and right sub-trees of node A.

Now, consider the AVL tree given in Fig.

10.46 and see how RL rotation is done to rebalance the tree.

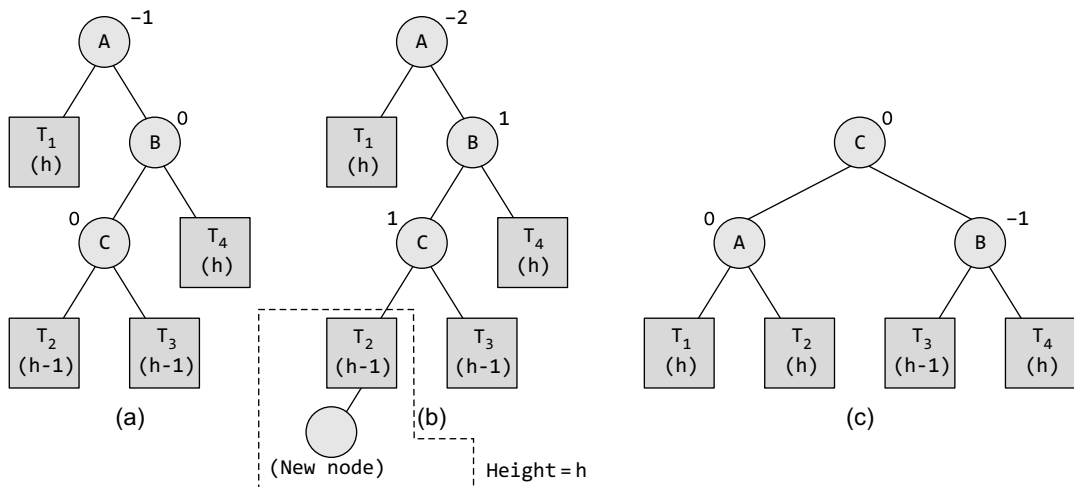


Figure 10.46 RL rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not -1 , 0 , or 1), so we apply RL rotation as shown in tree (c). Note that the new node has now become a part of tree τ_2 .

While rotation, node c becomes the root, with A and B as its left and right children. Node A has τ_1 and τ_2 as its left and right sub-trees and τ_3 and τ_4 become the left and right sub-trees of node B.

Example 10.6 Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

Solution

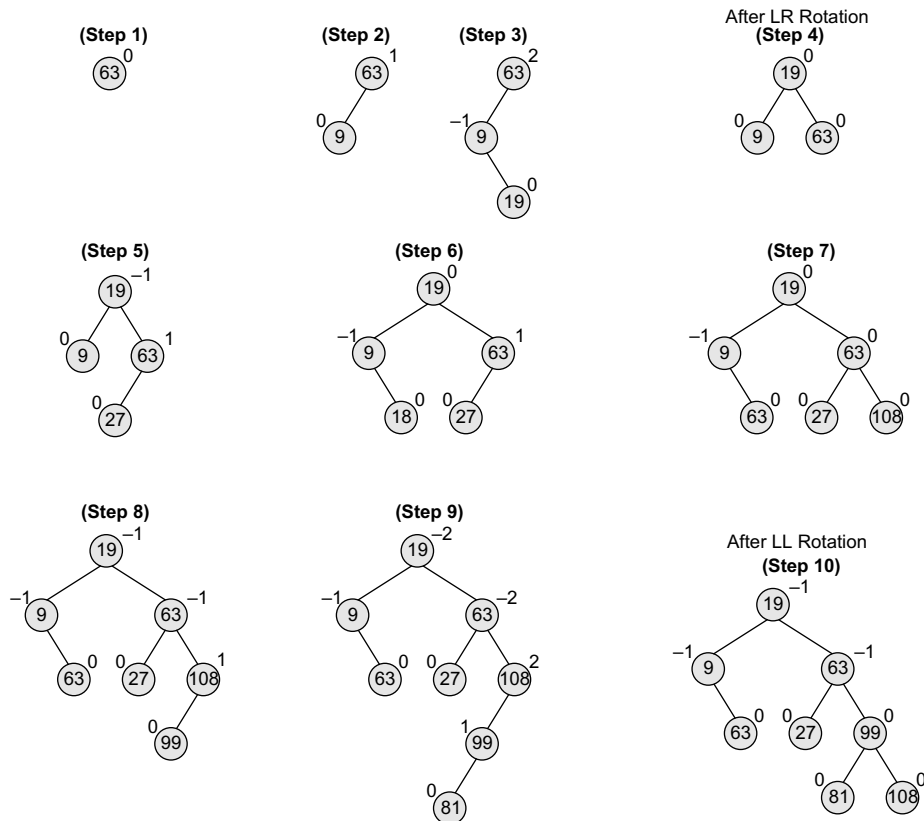


Figure 10.47 AVL tree

Deleting a Node from an AVL Tree

Deletion of a node in an AVL tree is similar to that of binary search trees. But it goes one step ahead. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation.

On deletion of node x from the AVL tree, if node A becomes the critical node (closest ancestor node on the path from x to the root node that does not have its balance factor as 1, 0, or -1), then the type of rotation depends on whether x is in the left sub-tree of A or in its right sub-tree. If the

node to be deleted is present in the left sub-tree of A, then L rotation is applied, else if x is in the right sub-tree, R rotation is performed.

Further, there are three categories of L and R rotations. The variations of L rotation are L-1, L0, and L1 rotation. Correspondingly for R rotation, there are R0, R-1, and R1 rotations. In this section, we will discuss only R rotation. L rotations are the mirror images of R rotations.

R0 Rotation

Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0. This is illustrated in Fig. 10.48.

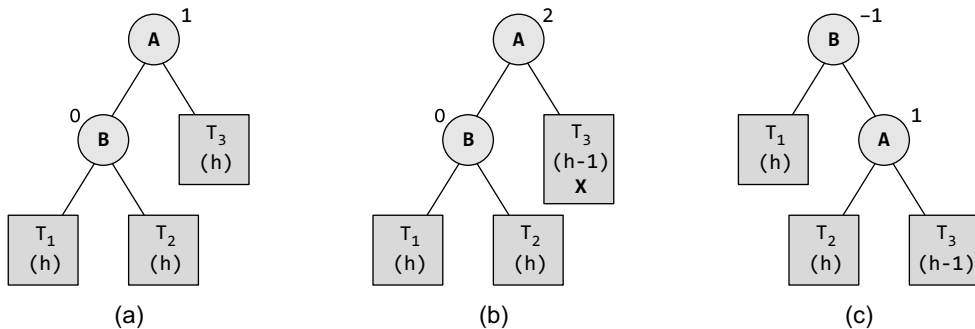


Figure 10.48 R0 rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node x is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not -1, 0, or 1). Since the balance factor of node B is 0, we apply R0 rotation as shown in tree (c).

During the process of rotation, node B becomes the root, with τ_1 and A as its left and right child. τ_2 and τ_3 become the left and right sub-trees of A.

R1 Rotation

Let B be the root of the left or right sub-tree of A (critical node). R1 rotation is applied if the balance factor of B is 1. Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors. This is illustrated in Fig. 10.50.

Example 10.7 Consider the AVL tree given in Fig. 10.49 and delete 72 from it.

Solution

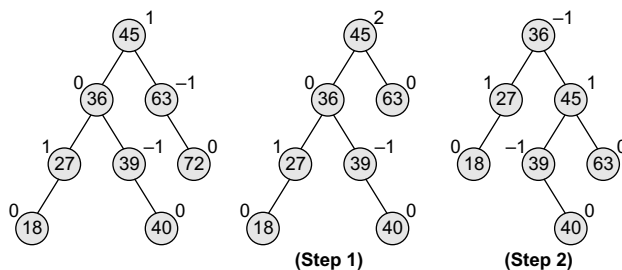


Figure 10.49 AVL tree

Tree (a) is an AVL tree. In tree (b), the node x is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not -1, 0, or 1). Since the balance factor of node B is 1, we apply R1 rotation as shown in tree (c).

During the process of rotation, node B becomes the root, with τ_1 and A as its left and right children. τ_2 and τ_3 become the left and right sub-trees of A.

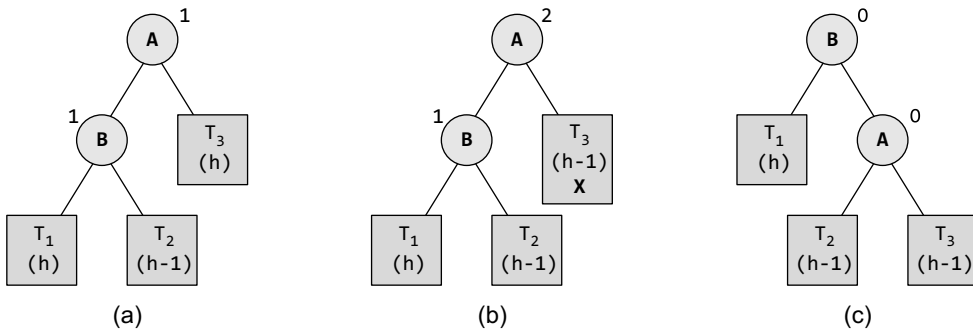


Figure 10.50 R1 rotation in an AVL tree

Example 10.8 Consider the AVL tree given in Fig. 10.51 and delete 72 from it.

Solution

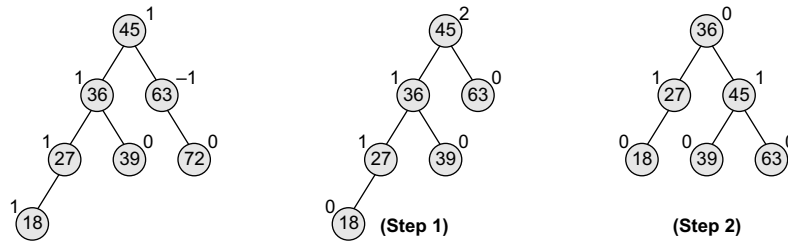


Figure 10.51 AVL tree

R-1 Rotation

Let B be the root of the left or right sub-tree of A (critical node). R-1 rotation is applied if the balance factor of B is -1 . Observe that R-1 rotation is similar to LR rotation. This is illustrated in Fig. 10.52.

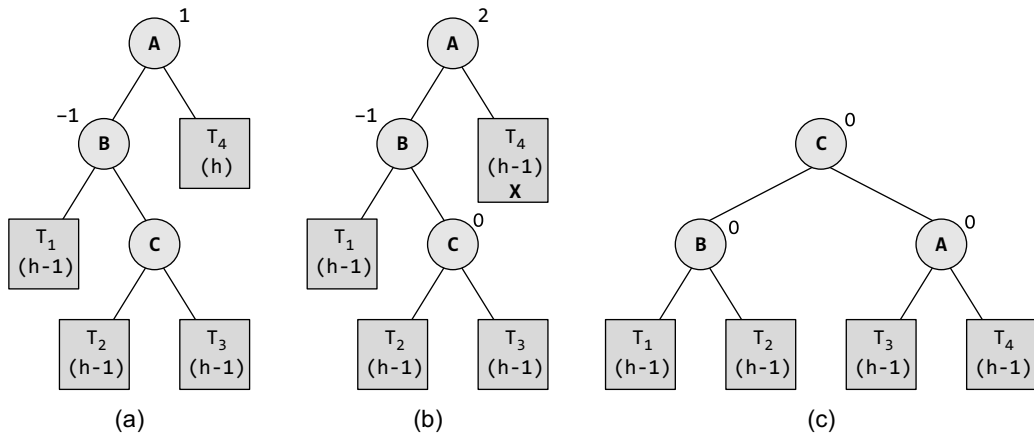


Figure 10.52 R1 Rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), the node x is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not -1 , 0 or 1). Since the balance factor of node B is -1 , we apply R-1 rotation as shown in tree (c).

While rotation, node c becomes the root, with τ_1 and A as its left and right child. τ_2 and τ_3 become the left and right sub-trees of A .

Example 10.9 Consider the AVL tree given in Fig. 10.53 and delete 72 from it.

Solution

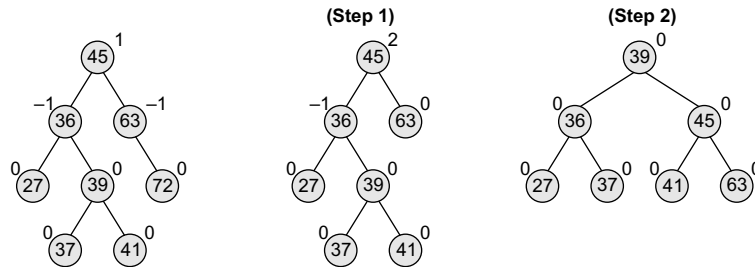


Figure 10.53 AVL tree

Example 10.10 Delete nodes 52, 36, and 61 from the AVL tree given in Fig. 10.54.

Solution

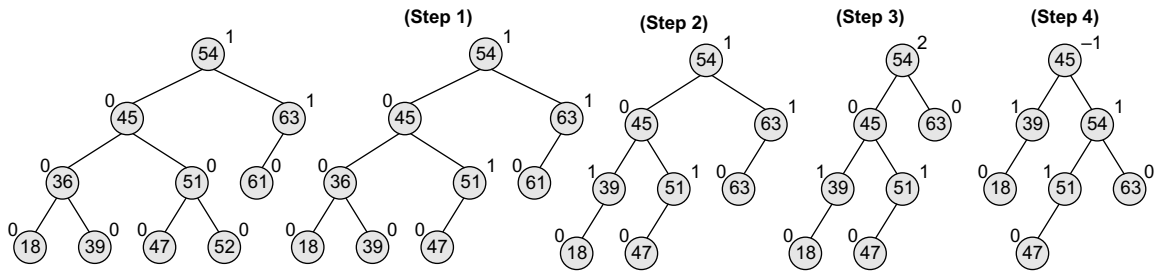


Figure 10.54 AVL tree

PROGRAMMING EXAMPLE

3. Write a program that shows insertion operation in an AVL tree.

```
#include <stdio.h>
typedef enum { FALSE ,TRUE } bool;
struct node
{
    int val;
    int balance;
    struct node *left_child;
    struct node *right_child;
};
struct node* search(struct node *ptr, int data)
{
    if(ptr!=NULL)
        if(data < ptr->val)
            ptr = search(ptr->left_child,data);
        else if( data > ptr->val)
            ptr = search(ptr->right_child, data);
    return(ptr);
}
struct node *insert (int data, struct node *ptr, int *ht_inc)
{
    struct node *aptr;
```

```

struct node *bptr;
if(ptr==NULL)
{
    ptr = (struct node *) malloc(sizeof(struct node));
    ptr->val = data;
    ptr->left_child = NULL;
    ptr->right_child = NULL;
    ptr->balance = 0;
    *ht_inc = TRUE;
    return (ptr);
}
if(data < ptr->val)
{
    ptr->left_child = insert(data, ptr->left_child, ht_inc);
    if(*ht_inc==TRUE)
    {
        switch(ptr->balance)
        {
            case -1: /* Right heavy */
                ptr->balance = 0;
                *ht_inc = FALSE;
                break;
            case 0: /* Balanced */
                ptr->balance = 1;
                break;
            case 1: /* Left heavy */
                aptr = ptr->left_child;
                if(aptr->balance == 1)
                {
                    printf("Left to Left Rotation\n");
                    ptr->left_child= aptr->right_child;
                    aptr->right_child = ptr;
                    ptr->balance = 0;
                    aptr->balance=0;
                    ptr = aptr;
                }
                else
                {
                    printf("Left to right rotation\n");
                    bptr = aptr->right_child;
                    aptr->right_child = bptr->left_child;
                    bptr->left_child = aptr;
                    ptr->left_child = bptr->right_child;
                    bptr->right_child = ptr;
                    if(bptr->balance == 1 )
                        ptr->balance = -1;
                    else
                        ptr->balance = 0;
                    if(bptr->balance == -1)
                        aptr->balance = 1;
                    else
                        aptr->balance = 0;
                    bptr->balance=0;
                    ptr = bptr;
                }
                *ht_inc = FALSE;
            }
        }
    }
}
if(data > ptr->val)

```

```

    {
        ptr->right_child = insert(info, ptr->right_child, ht_inc);
        if(*ht_inc==TRUE)
        {
            switch(ptr->balance)
            {
                case 1: /* Left heavy */
                    ptr->balance = 0;
                    *ht_inc = FALSE;
                    break;
                case 0: /* Balanced */
                    ptr->balance = -1;
                    break;
                case -1: /* Right heavy */
                    aptr = ptr->right_child;
                    if(aptr->balance == -1)
                    {
                        printf("Right to Right Rotation\n");
                        ptr->right_child= aptr->left_child;
                        aptr->left_child = ptr;
                        ptr->balance = 0;
                        aptr->balance=0;
                        ptr = aptr;
                    }
                    else
                    {
                        printf("Right to Left Rotation\n");
                        bptr = aptr->left_child;
                        aptr->left_child = bptr->right_child;
                        bptr->right_child = aptr;
                        ptr->right_child = bptr->left_child;
                        bptr->left_child = pptr;
                        if(bptr->balance == -1)
                            ptr->balance = 1;
                        else
                            ptr->balance = 0;
                        if(bptr->balance == 1)
                            aptr->balance = -1;
                        else
                            aptr->balance = 0;
                        bptr->balance=0;
                        ptr = bptr;
                    }/*End of else*/
                    *ht_inc = FALSE;
            }
        }
    }
    return(ptr);
}

void display(struct node *ptr, int level)
{
    int i;
    if ( ptr!=NULL )
    {
        display(ptr->right_child, level+1);
        printf("\n");
        for (i = 0; i < level; i++)
            printf(" ");
        printf("%d", ptr->val);
        display(ptr->left_child, level+1);
    }
}

```

```

    }
}
void inorder(struct node *ptr)
{
    if(ptr!=NULL)
    {
        inorder(ptr->left_child);
        printf("%d ",ptr->val);
        inorder(ptr->right_child);
    }
}
main()
{
    bool ht_inc;
    int data ;
    int option;
    struct node *root = (struct node *)malloc(sizeof(struct node));
    root = NULL;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Display\n");
        printf("3.Quit\n");
        printf("Enter your option : ");
        scanf("%d",&option);
        switch(choice)
        {
            case 1:
                printf("Enter the value to be inserted : ");
                scanf("%d", &data);
                if( search(root,data) == NULL )
                    root = insert(data, root, &ht_inc);
                else
                    printf("Duplicate value ignored\n");
                break;
            case 2:
                if(root==NULL)
                {
                    printf("Tree is empty\n");
                    continue;
                }
                printf("Tree is :\n");
                display(root, 1);
                printf("\n\n");
                printf("Inorder Traversal is: ");
                inorder(root);
                printf("\n");
                break;
            case 3:
                exit(1);
            default:
                printf("Wrong option\n");
        }
    }
}

```

10.5 RED-BLACK TREES

A red-black tree is a self-balancing binary search tree that was invented in 1972 by Rudolf Bayer who called it the 'symmetric binary B-tree'. Although a red-black tree is complex, it has good worst-