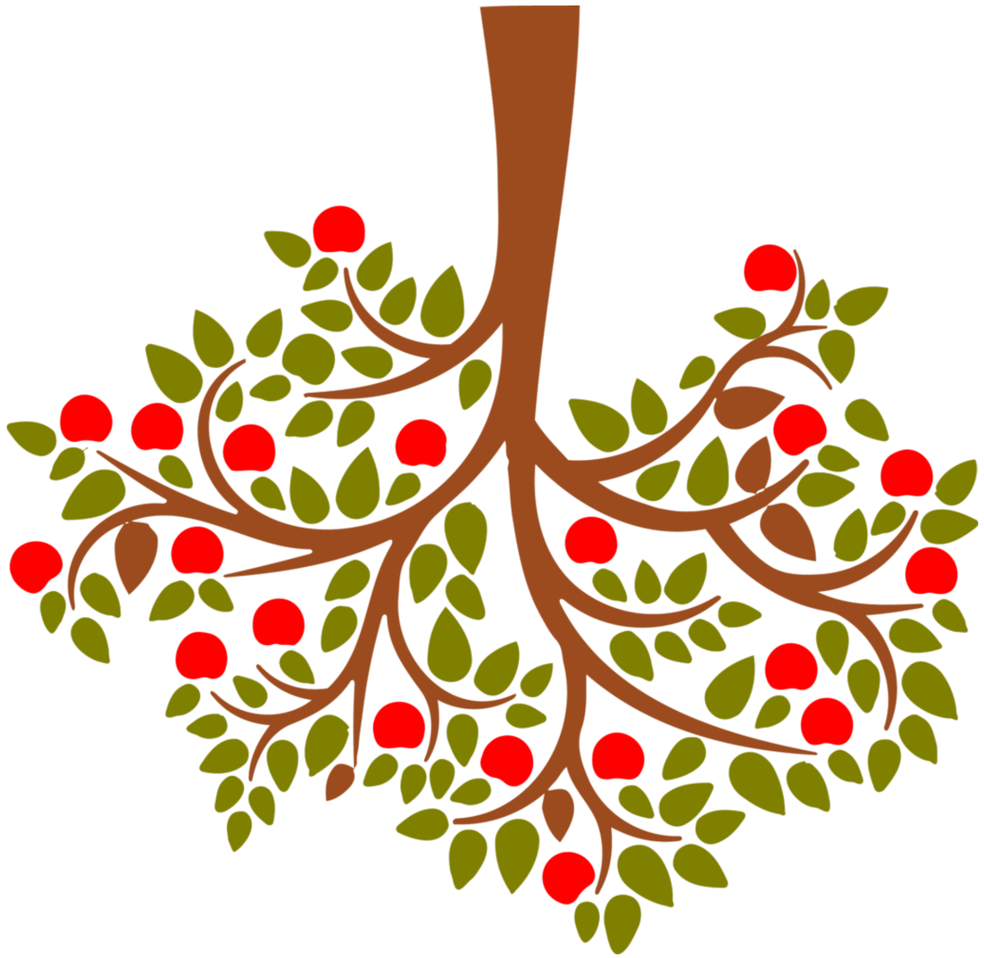
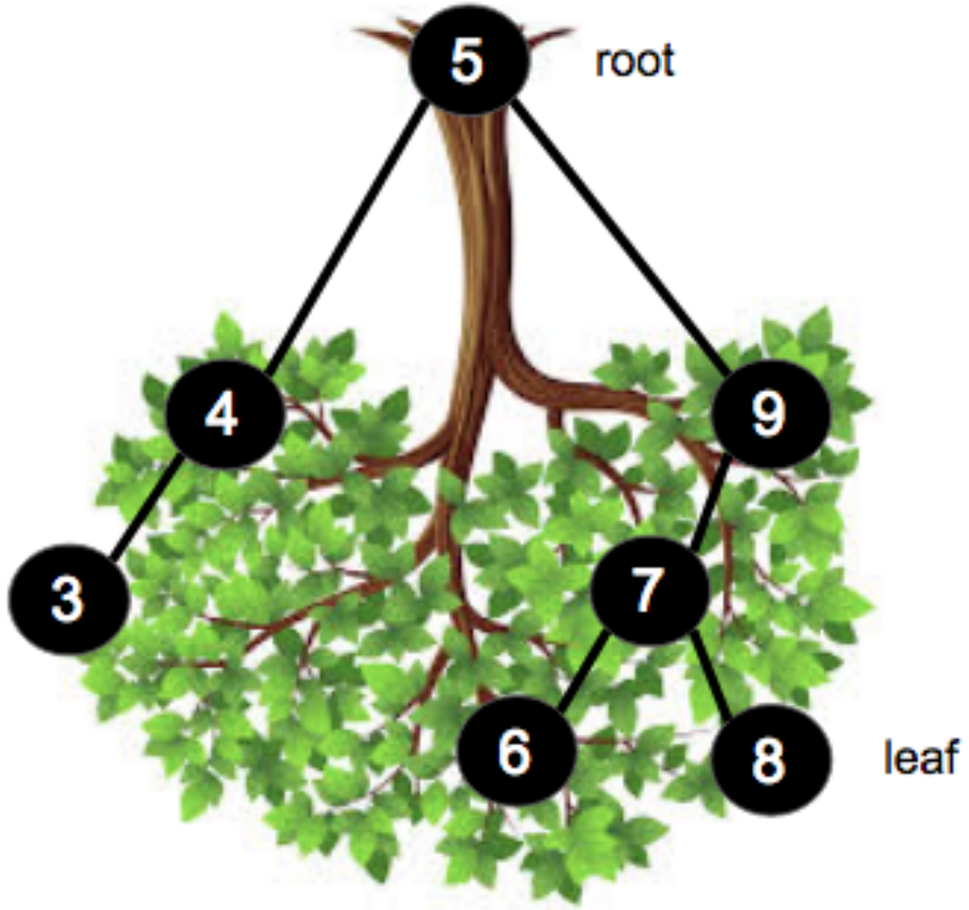
Module 2-2 Efficient Binary Tree





Efficient Binary Tree topic' s Outlines

- ★ Binary Search Tree
- ★ Threaded Binary Search Tree
- ★ AVL Tree
- ★ Applications of Trees

Binary Search Tree

- 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.

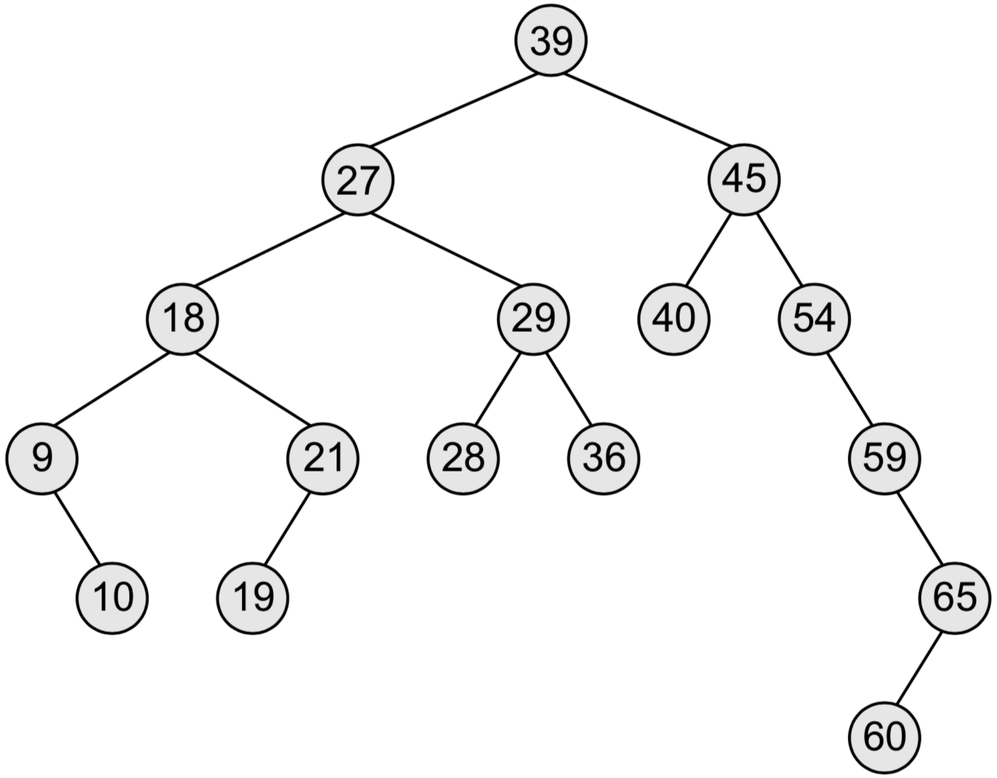
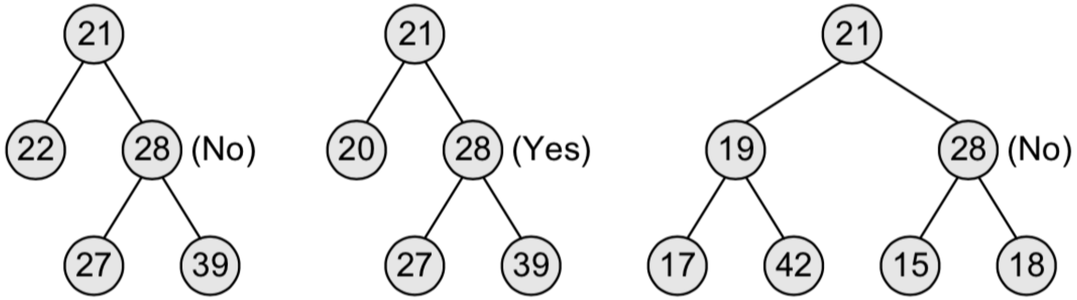


Figure 10.1 Binary search tree

Binary Search Tree

- 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.

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



Create Binary Search Tree

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

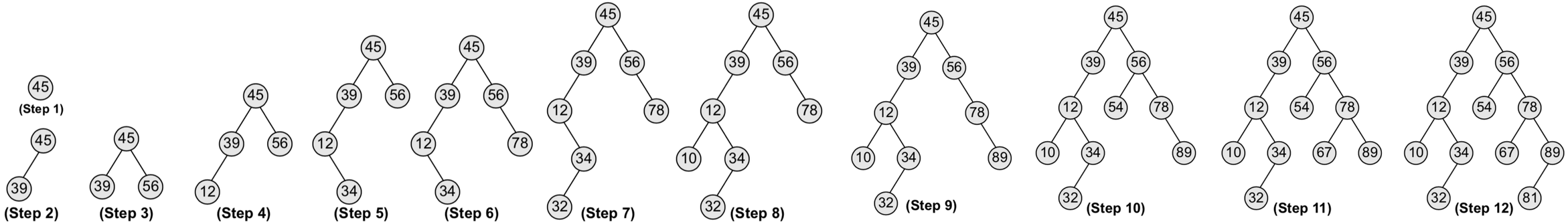
Solution

Create Binary Search Tree

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

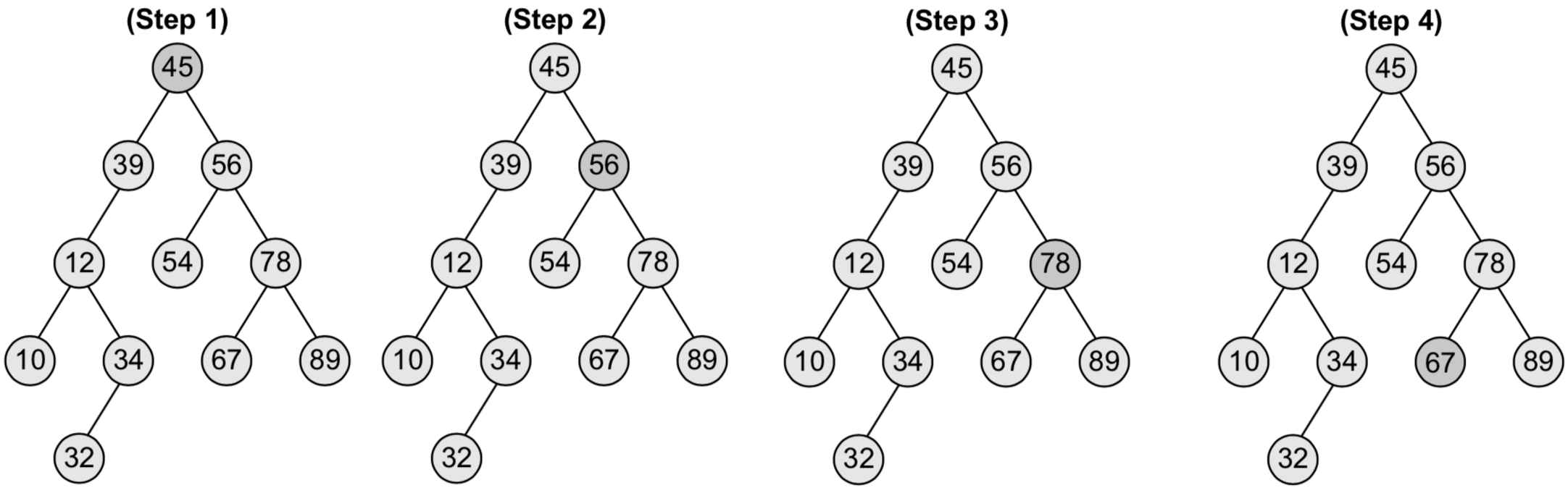
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution



Search a node in Binary Search Tree

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



Search a node in Binary Search Tree

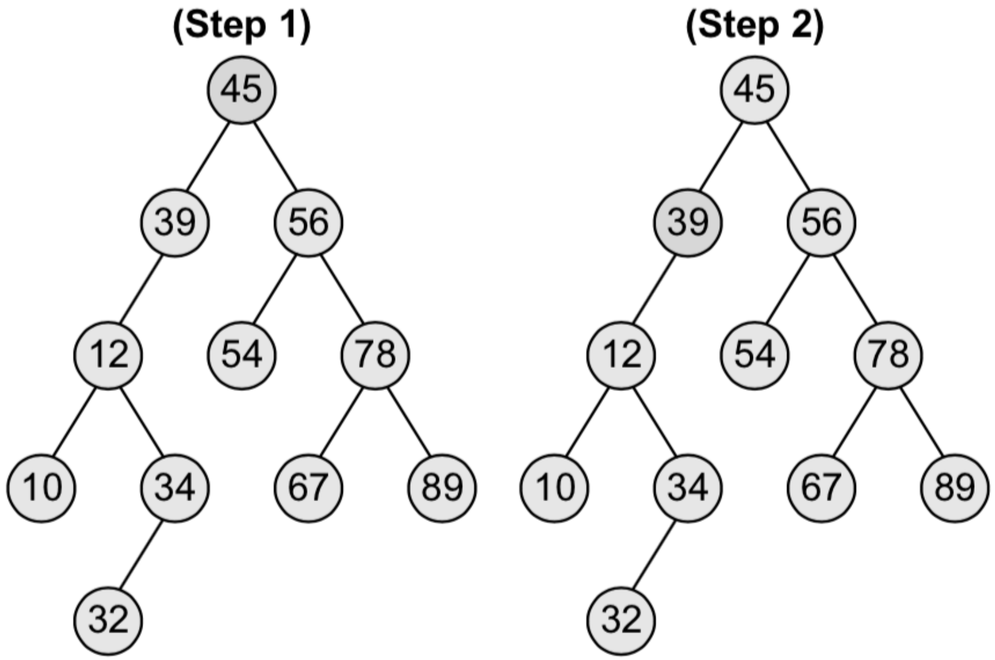


Figure 10.7 Searching a node with the value 40 in the given binary search tree

```
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 a new node into 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

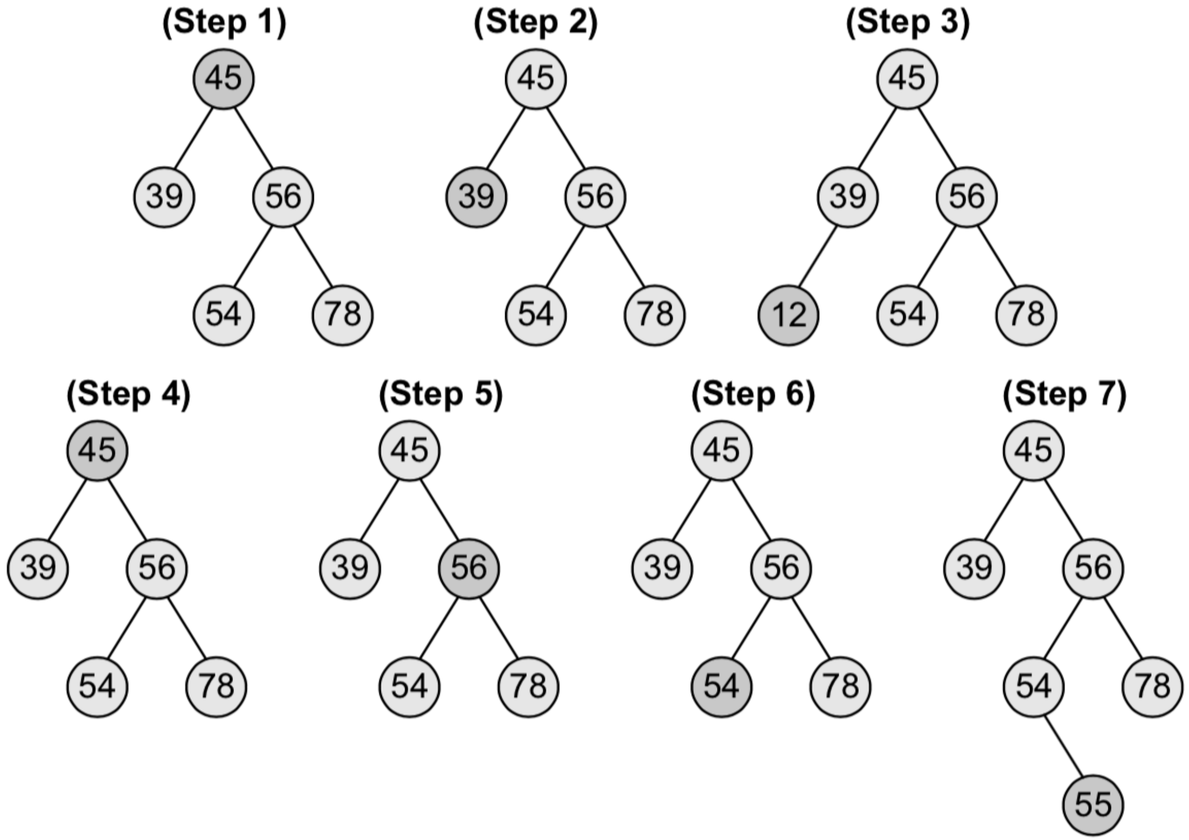


Figure 10.10 Inserting nodes with values 12 and 55 in the given binary search tree

Insert a new node into 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

```
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;
}
```

Delete a node from 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.

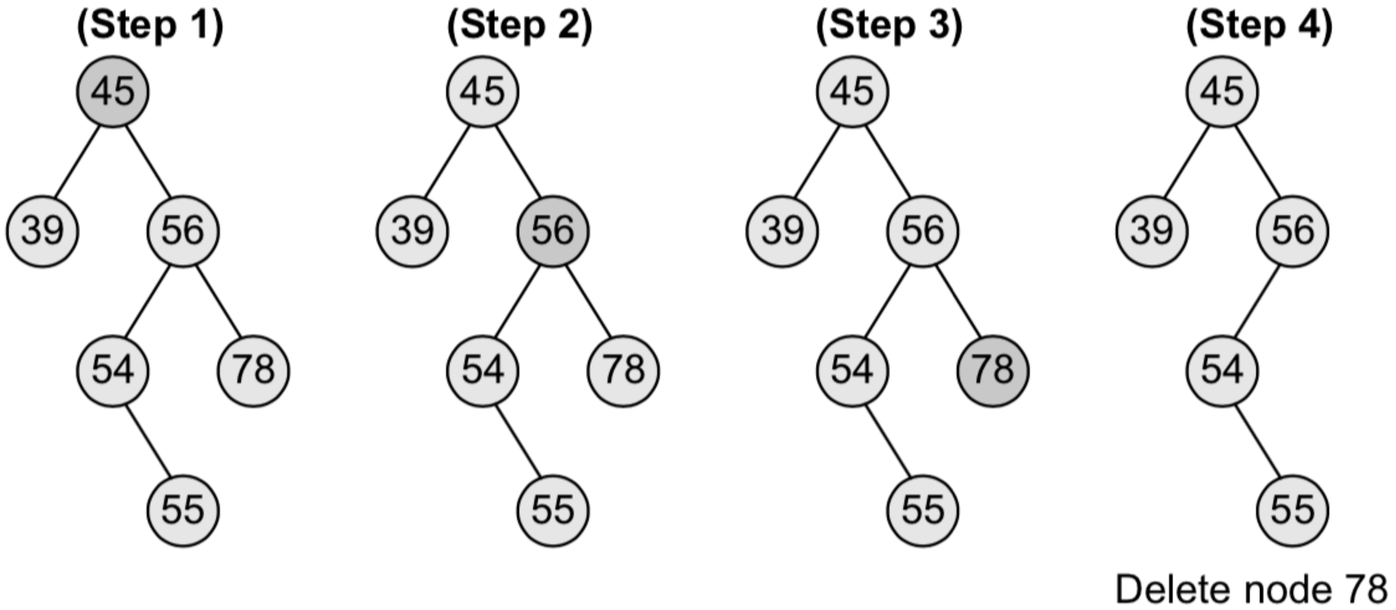


Figure 10.11 Deleting node 78 from the given binary search tree

Delete a node from Binary Search Tree

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.

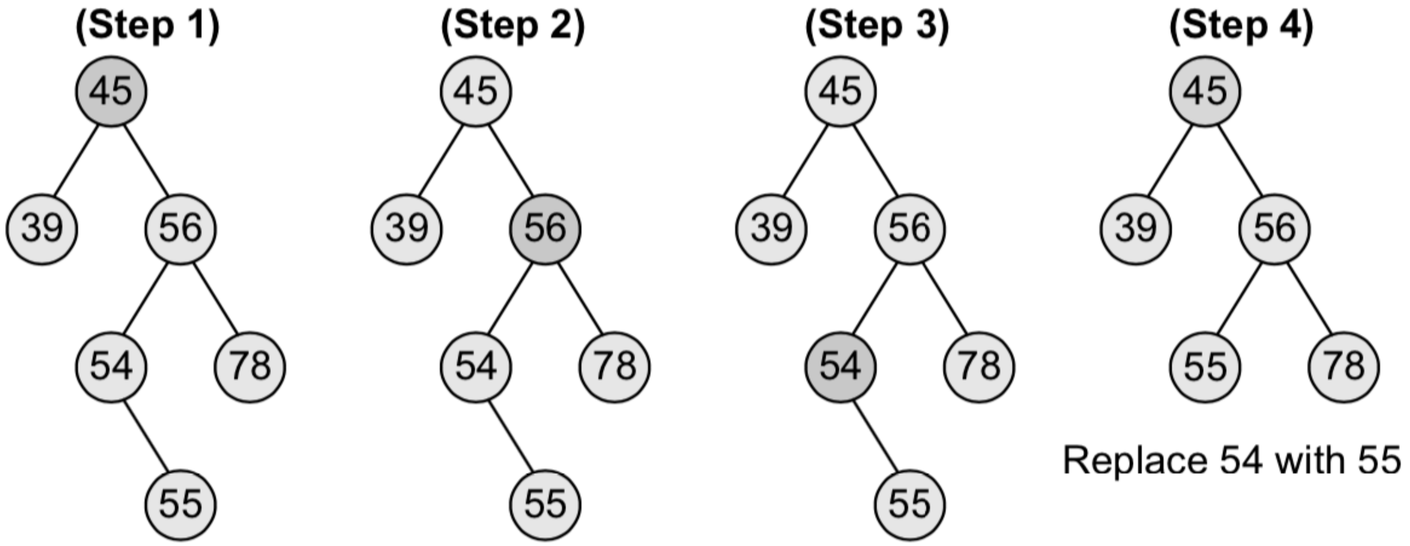


Figure 10.12 Deleting node 54 from the given binary search tree

Delete a node from Binary Search Tree

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.

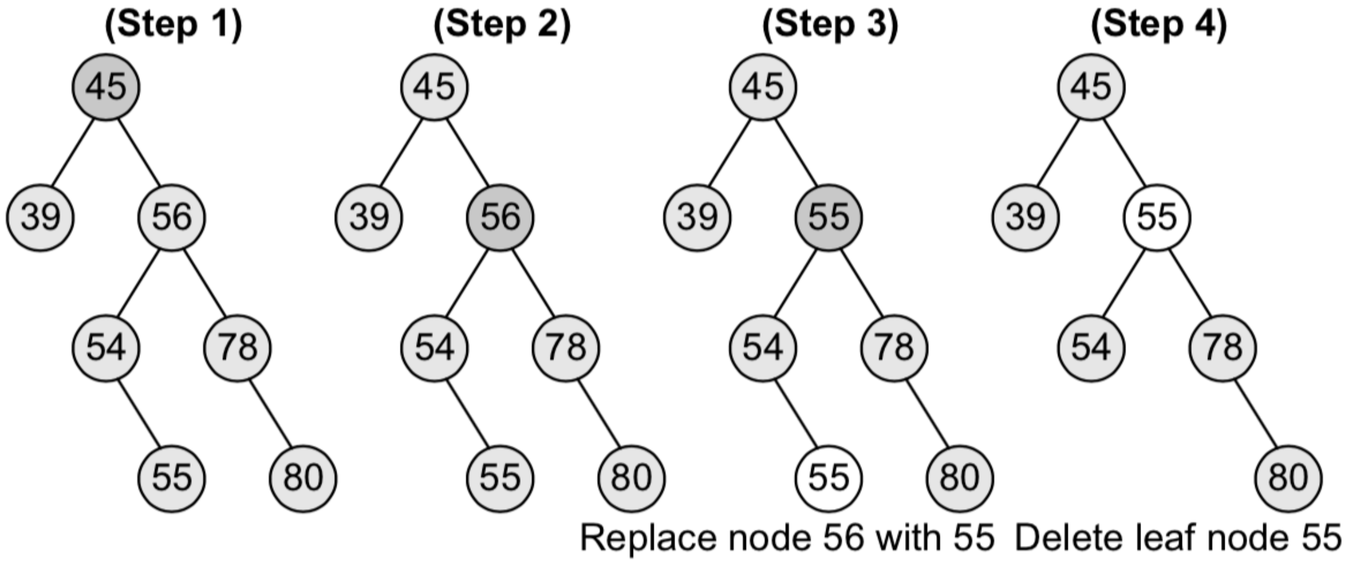


Figure 10.13 Deleting node 56 from the given binary search tree

Delete a node from Binary Search Tree

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

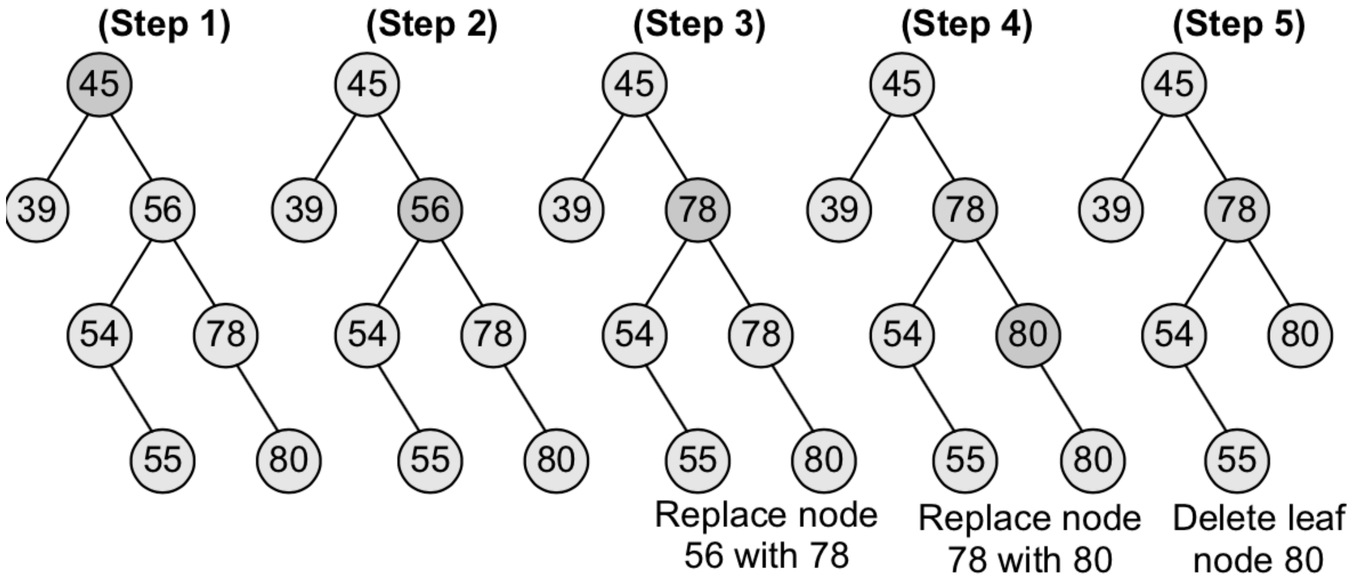


Figure 10.14 Deleting node 56 from the given binary search tree

Delete a node from Binary Search Tree

```
Delete (TREE, VAL)

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

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

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.

Height of Binary Search Tree

```
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

```
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);
        }
}
```

Total nodes of 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

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

Total external nodes of Binary Search Tree

```
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

```
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));
}
```

Total internal nodes of 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

```
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);
}
```

Threaded Binary Tree

- A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.
- 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.

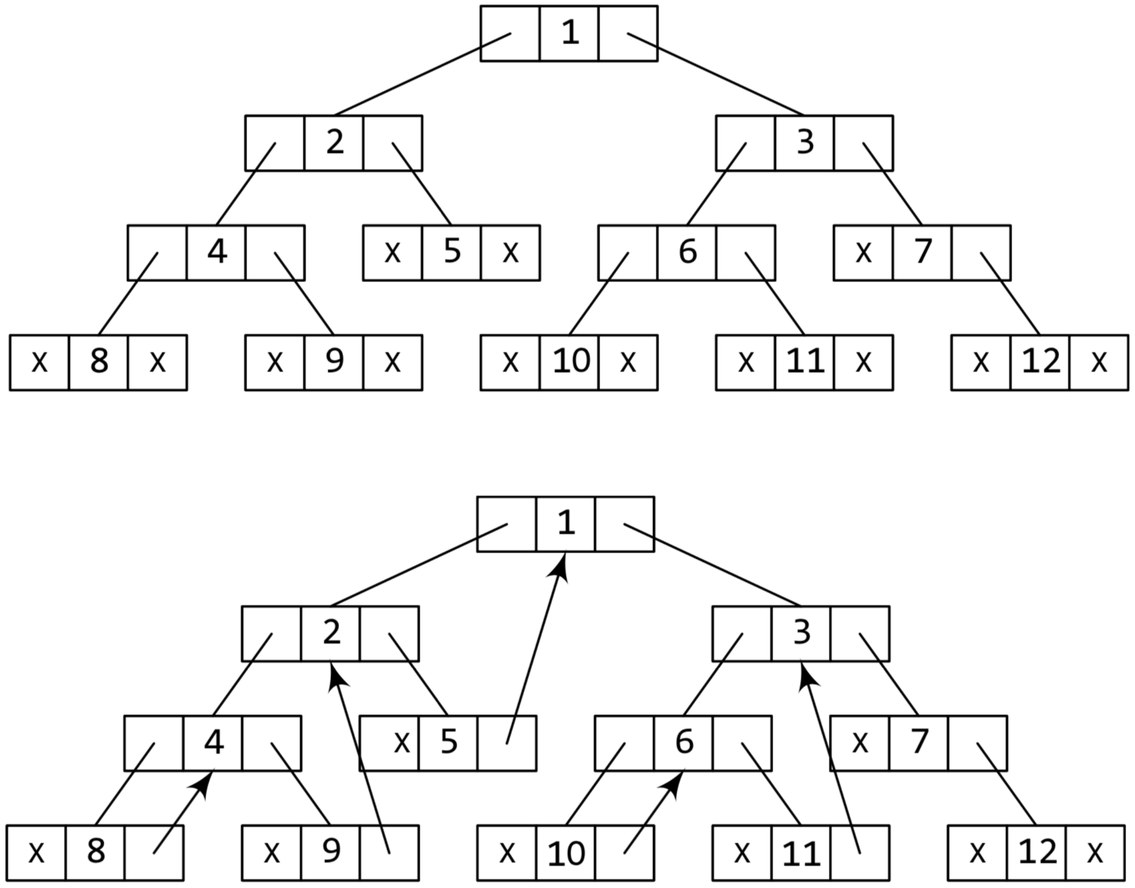


Figure 10.30 (a) Linked representation of the binary tree with one-way threading

Threaded Binary Tree

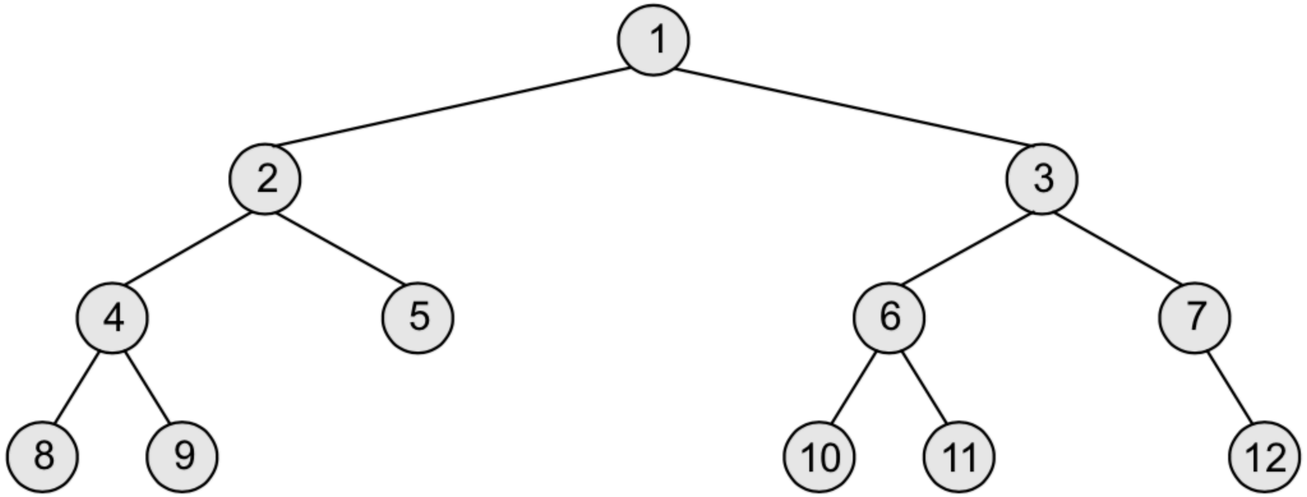


Figure 10.29 (a) Binary tree without threading

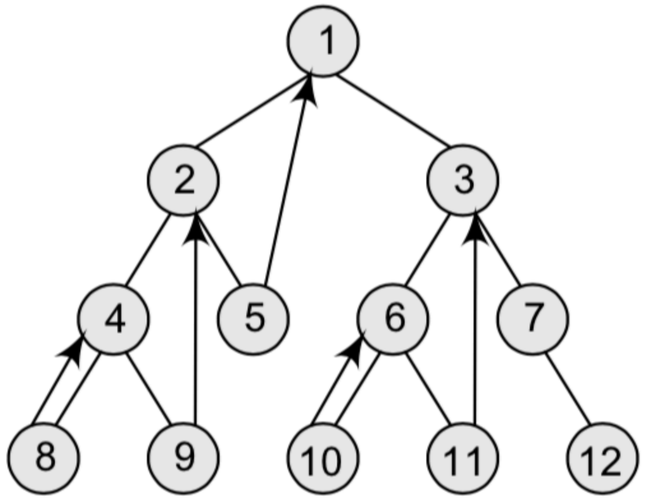


Figure 10.30 (b) Binary tree with one-way threading

Two-Way Threaded Binary Tree

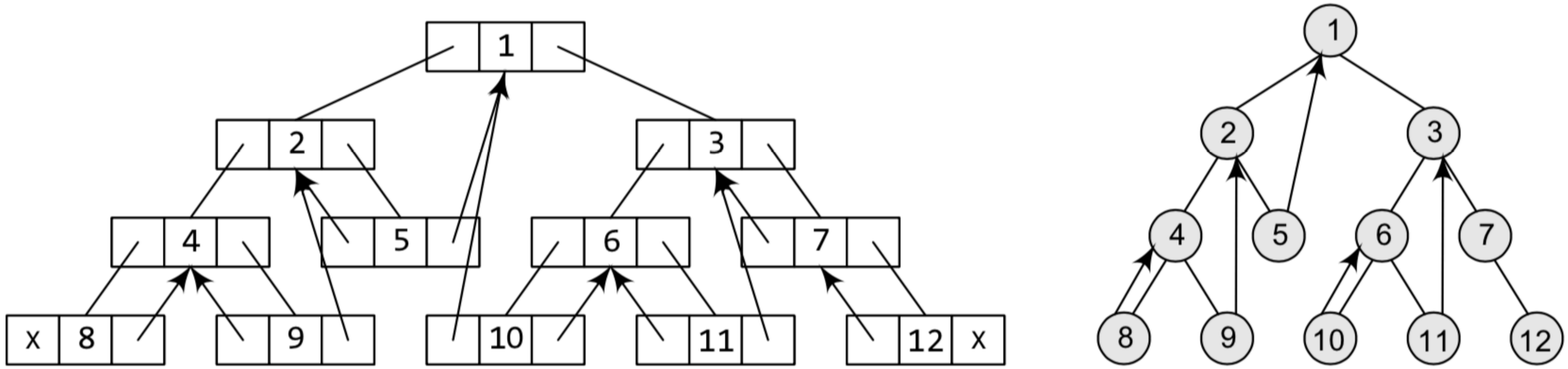


Figure 10.31 (a) Linked representation of the binary tree with threading, (b) binary tree with two-way threading

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.

Threaded Binary Tree

```
#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;
}
```

Threaded Binary Tree

```
#include <stdio.h>
#include <conio.h>
struct tree
{
        int val;
        struct tree *right;
        struct tree *left;
        int thread;
};
struct tree *root = NULL;
```

```
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;
}
```

Threaded Binary Tree

```
#include <stdio.h>
#include <conio.h>
struct tree
{
        int val;
        struct tree *right;
        struct tree *left;
        int thread;
};
struct tree *root = NULL;
```

```
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);
}
```

Threaded Binary Tree

```
#include <stdio.h>
#include <conio.h>
struct tree
{
        int val;
        struct tree *right;
        struct tree *left;
        int thread;
};
struct tree *root = 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
```

AVL Tree

✤ AVL tree is a self-balancing binary search tree invented by two soviet inventors, 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).

Remarked: It is often compared with Red-Black tree.

AVL Trees are faster than red-black trees because they are more strictly balanced.

Balance Factor in AVL Tree

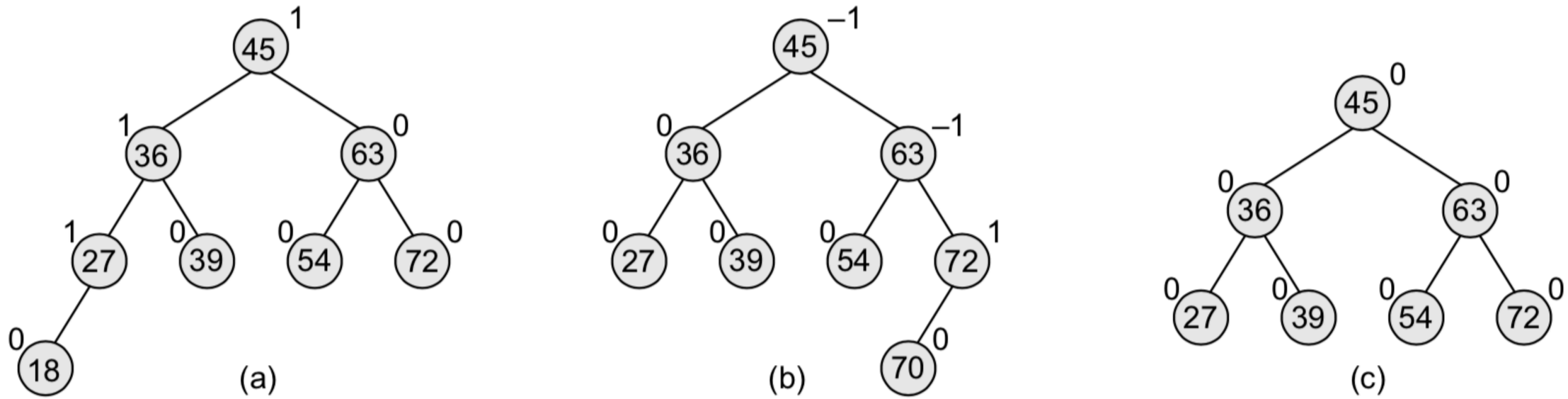


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Insert a new node into 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.

Insert a new node into AVL Tree

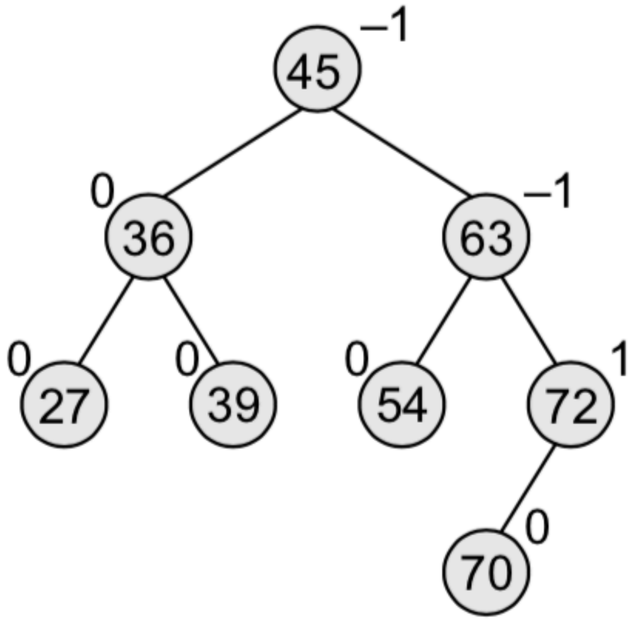


Figure 10.36 AVL tree

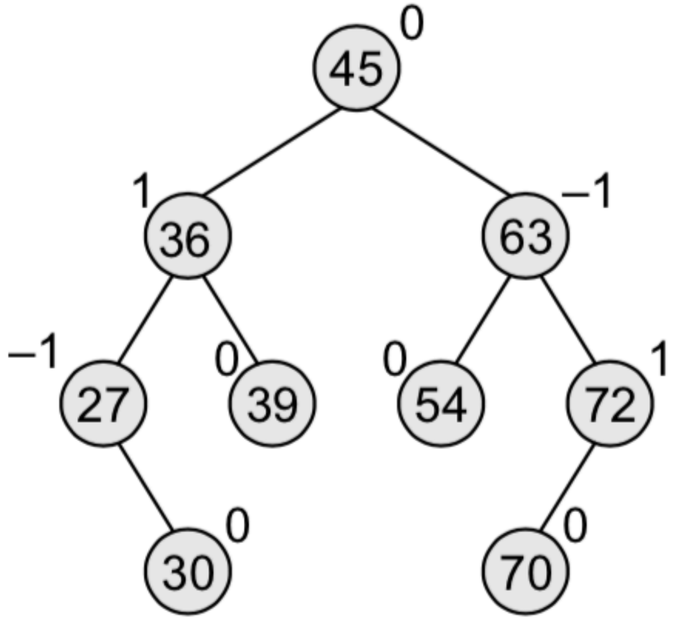


Figure 10.37 AVL tree after inserting a node with the value 30

Delete a node from AVL Tree

- **Balance factor** of all nodes is either 0, or +1 or -1.

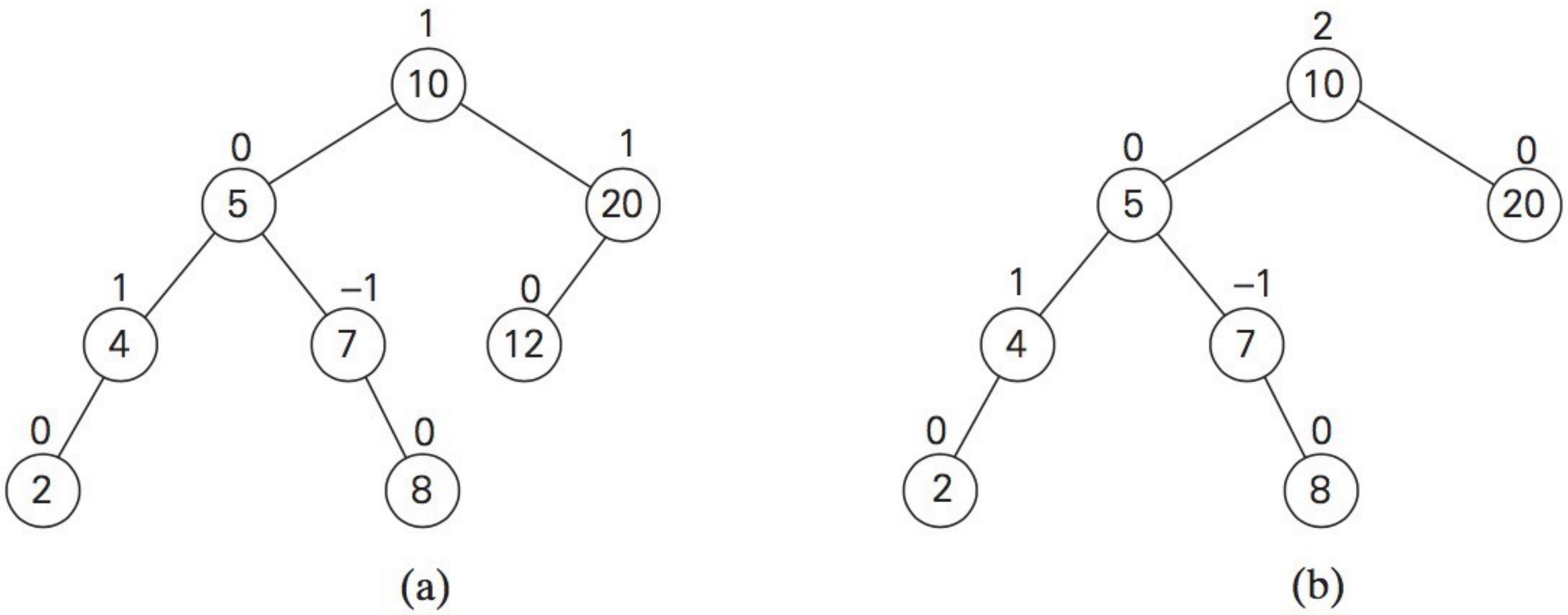


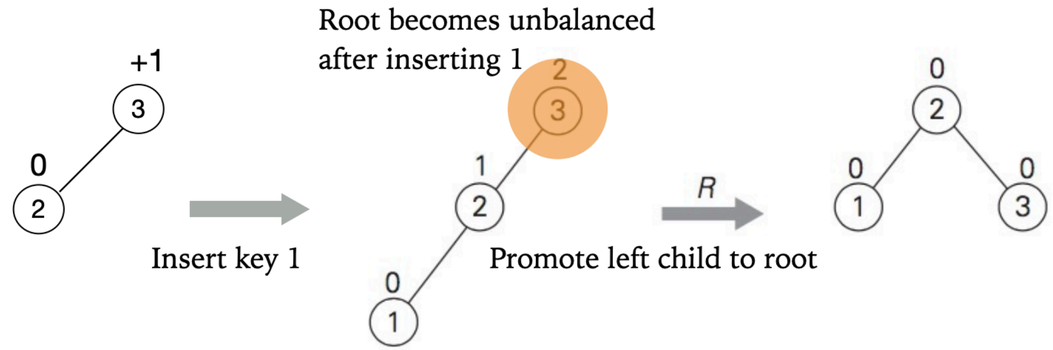
FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

Rotation in AVL Tree

✤ Perform rotation at the unbalanced node closest the newly inserted node.

✤ Four types of rotations depending on where the new key has been inserted

- ✤ Single R rotation
- ✤ Single L rotation
- ✤ Double LR rotation
- ✤ Double RL rotation



✤ LL rotation or Single-R Rotation: The new node is inserted in the left sub-tree of the left sub-tree of the critical node.

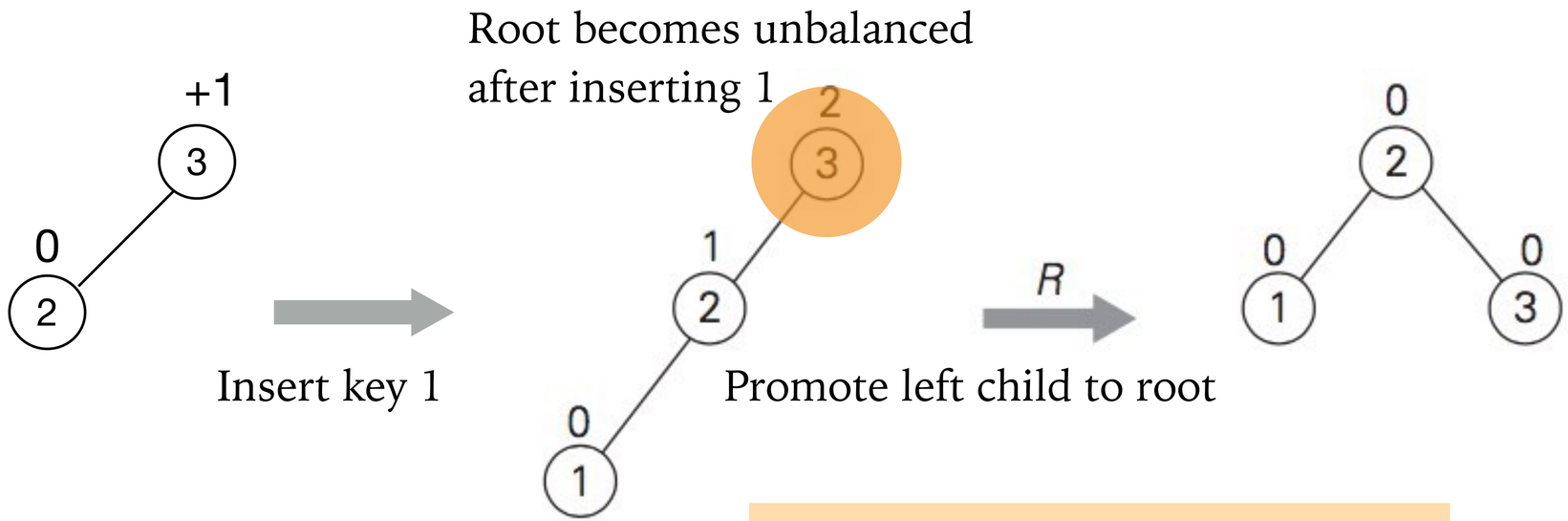
✤ RR rotation or Single-L Rotation: The new node is inserted in the right sub-tree of the right sub-tree of the critical node.

✤ 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.

LL or Single R-Rotation in AVL Tree

New key inserted into the ***left subtree of the left child*** of a tree whose root had the balance of +1 before the insertion.



Single R-rotation
(Promote left child to root)

LL or Single R-Rotation in AVL Tree

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

Solution

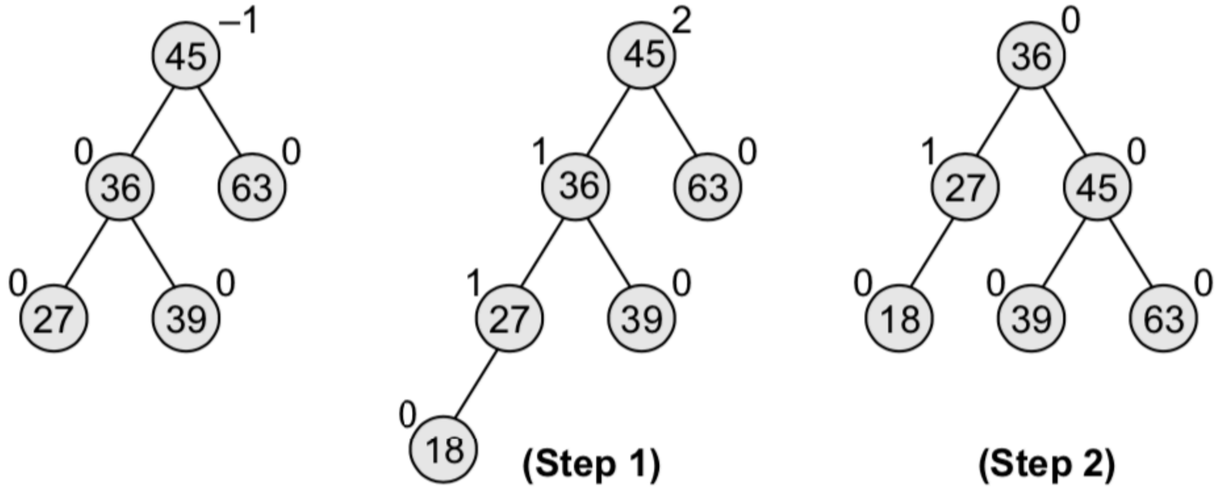


Figure 10.41 AVL tree

LL or Single R-Rotation in AVL Tree

New key inserted into the ***left subtree of the left child*** of a tree whose root had the balance of +1 before the insertion.

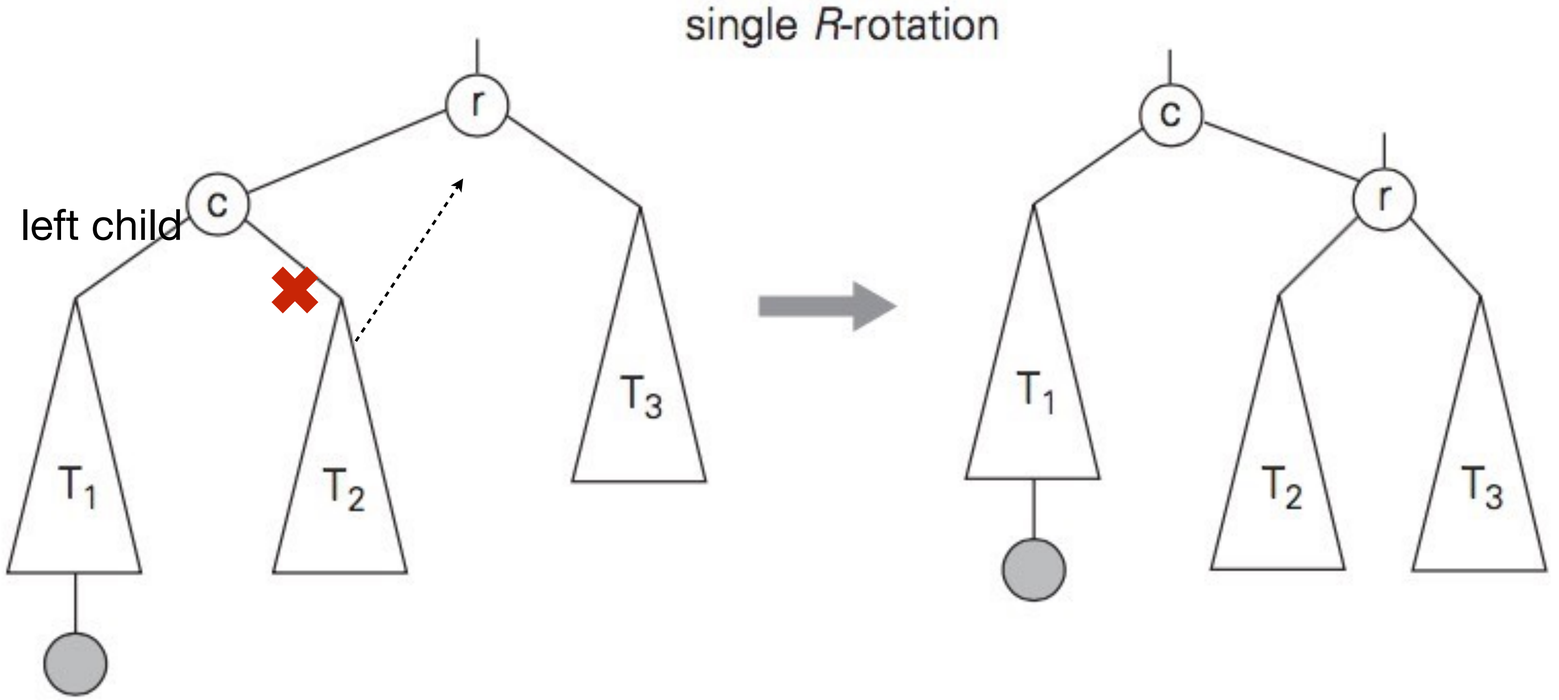
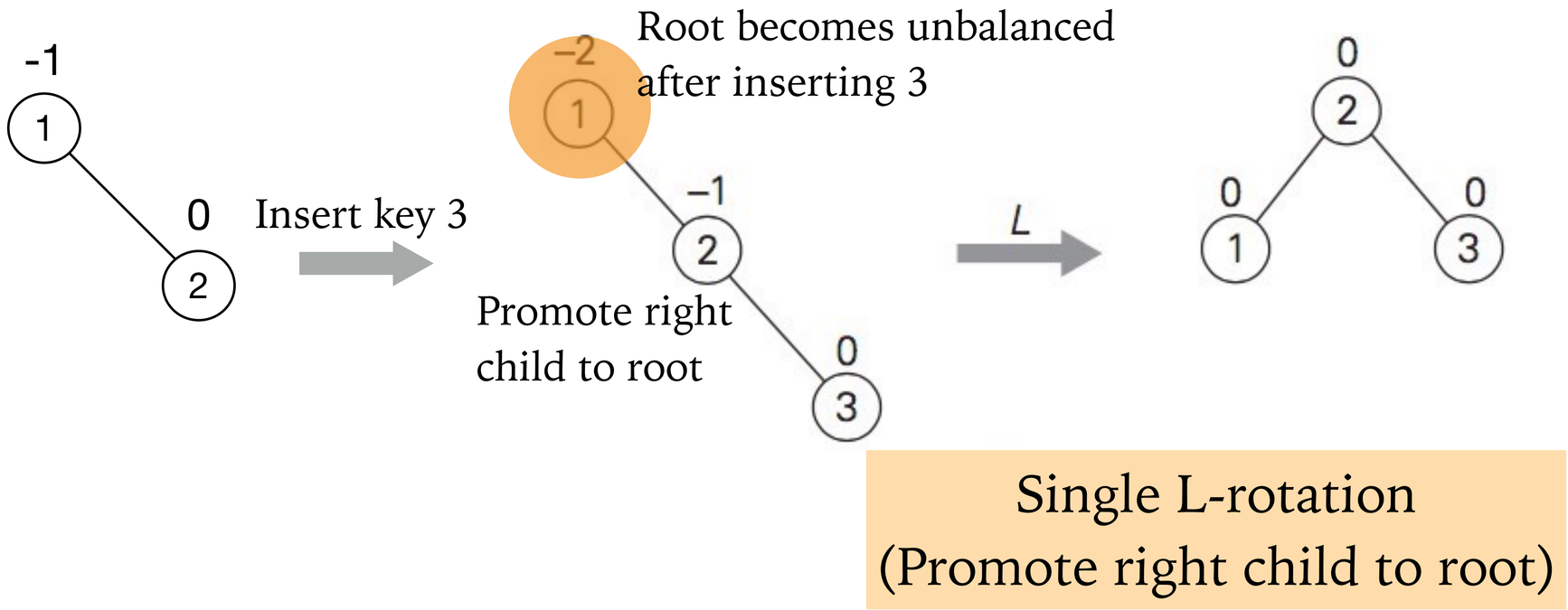


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

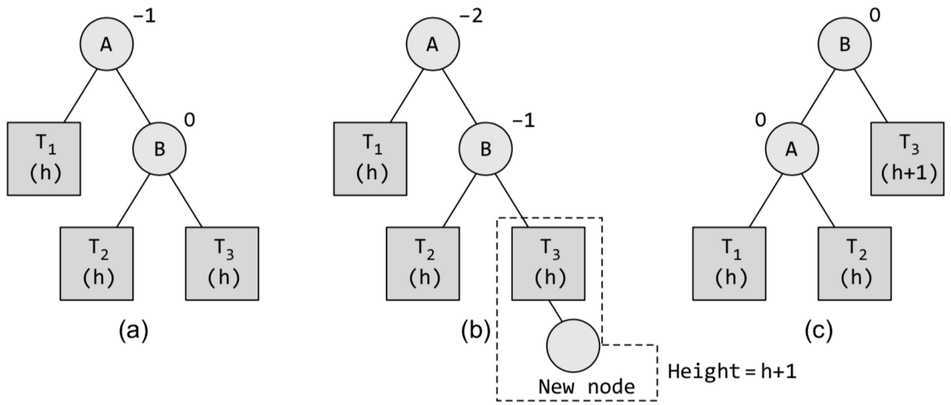
RR or Single L-Rotation in AVL Tree

New key inserted into the ***right subtree of the right child*** of a tree whose root had the balance of -1 before the insertion.



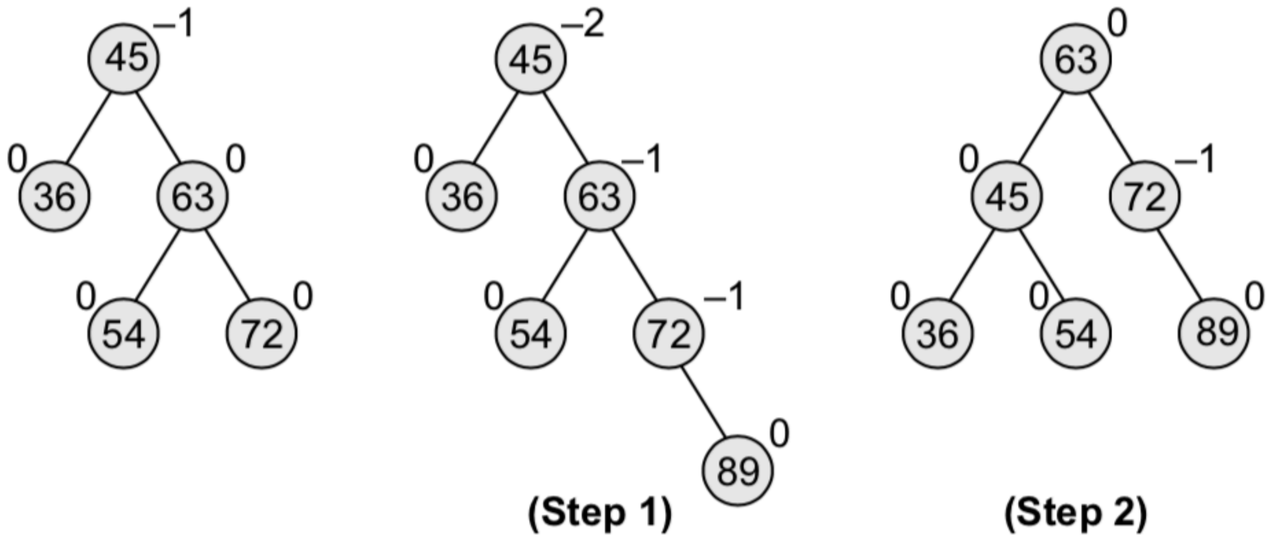
Single L-rotation
(Promote right child to root)

RR or Single L-Rotation in AVL Tree

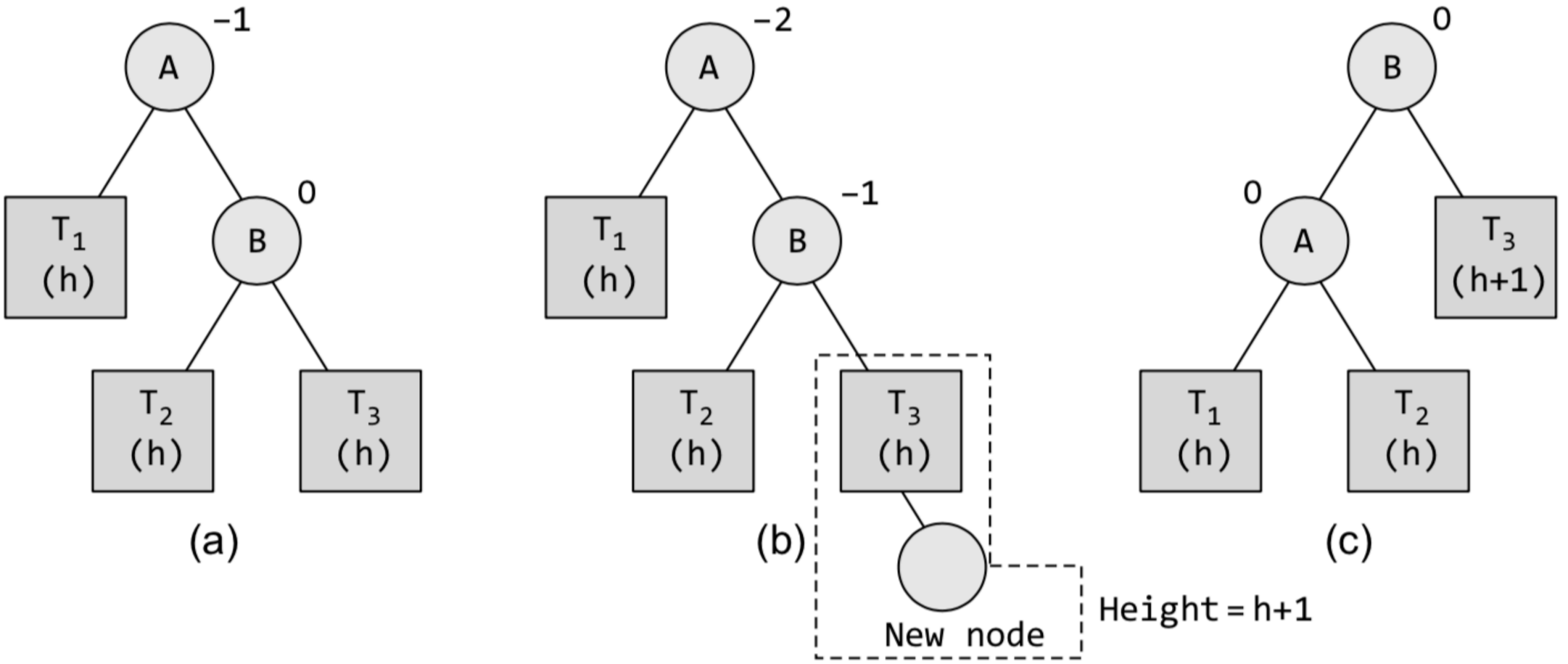


Example 10.4 Consider the AVL tree given in Fig. 10.43 and insert 89 into it.

Solution

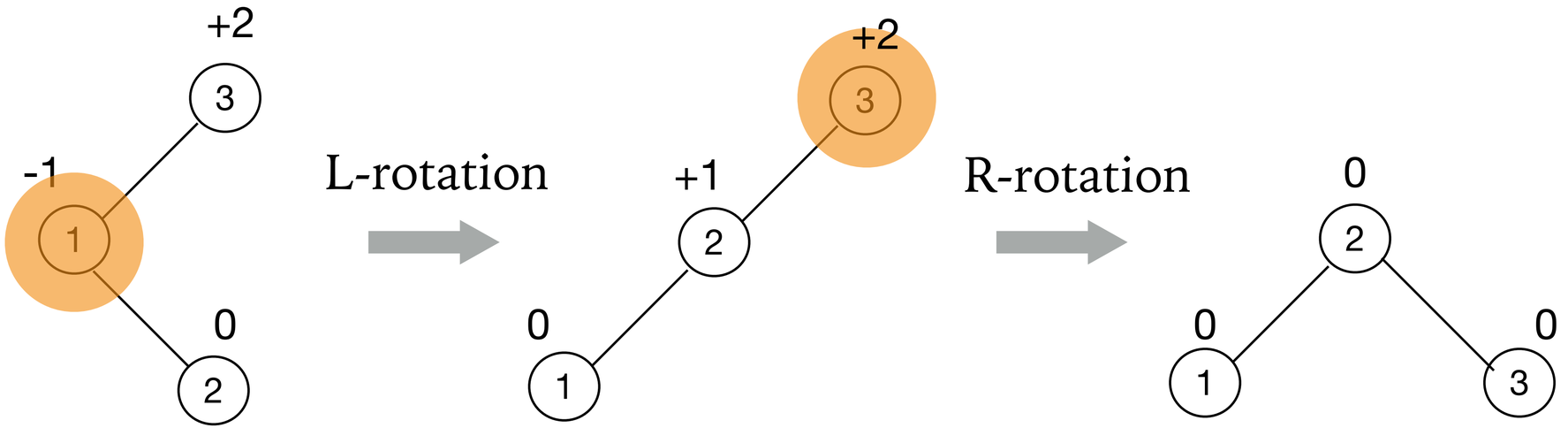


RR or Single L-Rotation in AVL Tree



LR-Rotation in AVL Tree

New key inserted into the *right subtree of the left child* of a tree whose root had the balance of +1 before the insertion.



Double LR-rotation
(L-rotation followed by R-rotation)

LR-Rotation in AVL Tree

New key inserted into the *right subtree of the left child* of a tree whose root had the balance of +1 before the insertion.

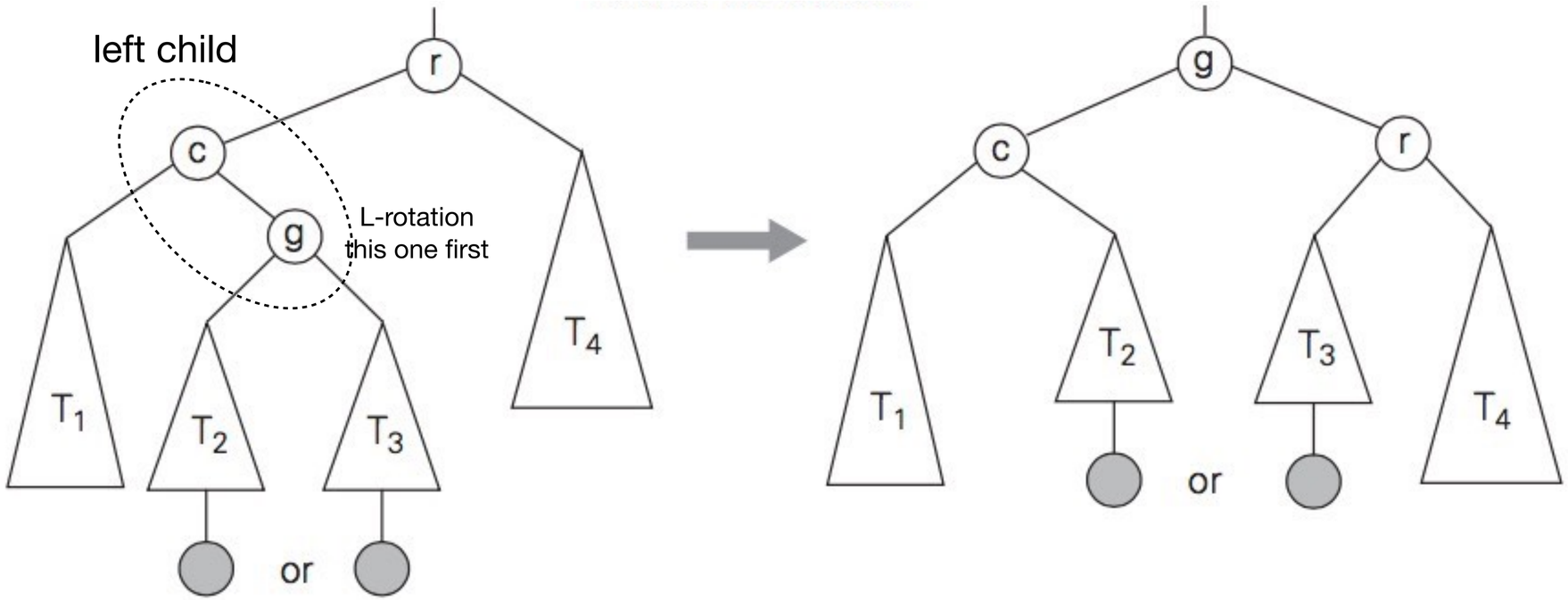


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

LR-Rotation in AVL Tree

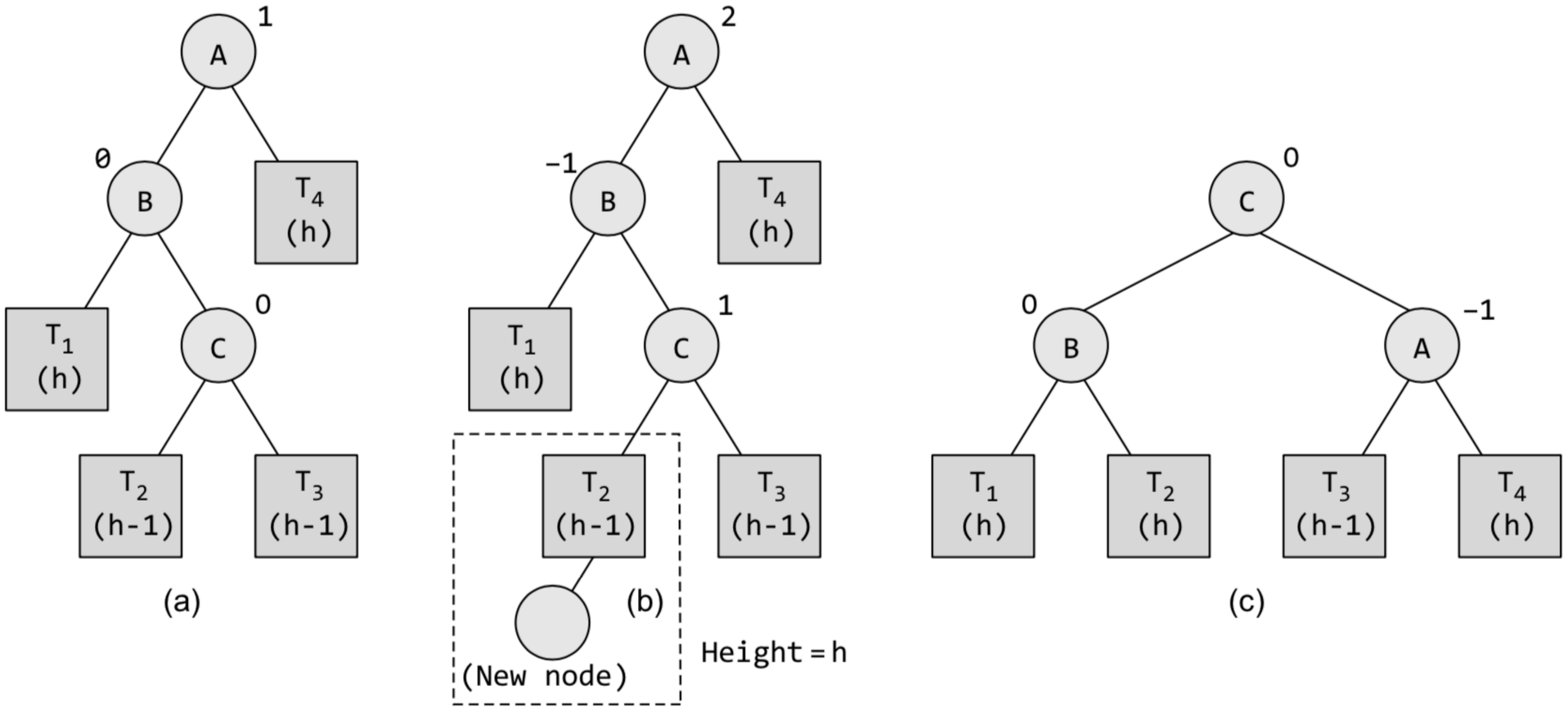
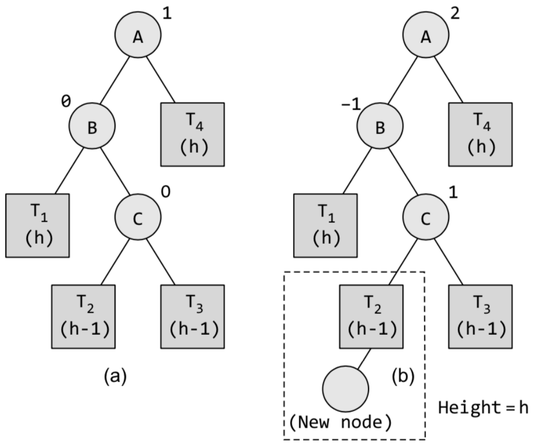
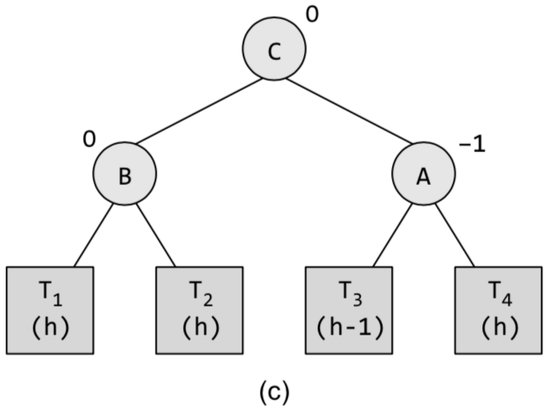


Figure 10.44 LR rotation in an AVL tree

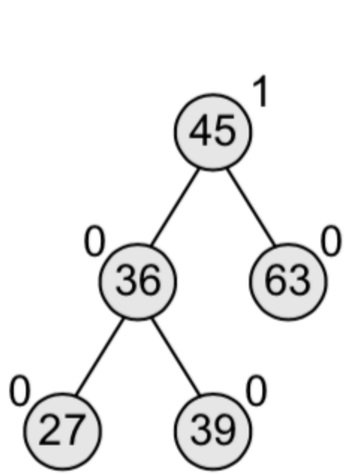
LR-Rotation in AVL Tree



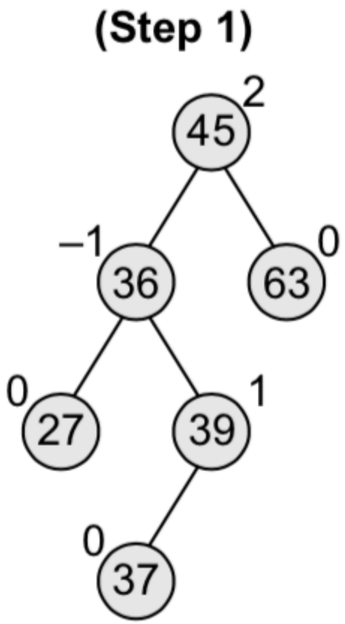


Example 10.5 Consider the AVL tree given in Fig. 10.45 and insert 37 into it.

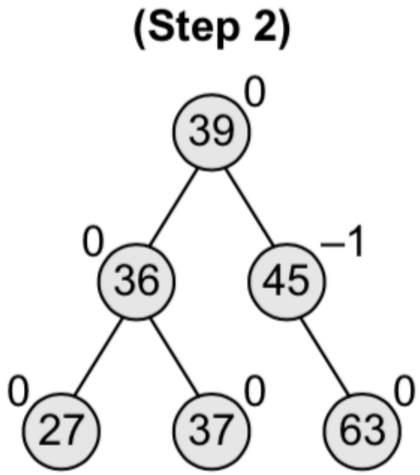
Solution



(Step 1)

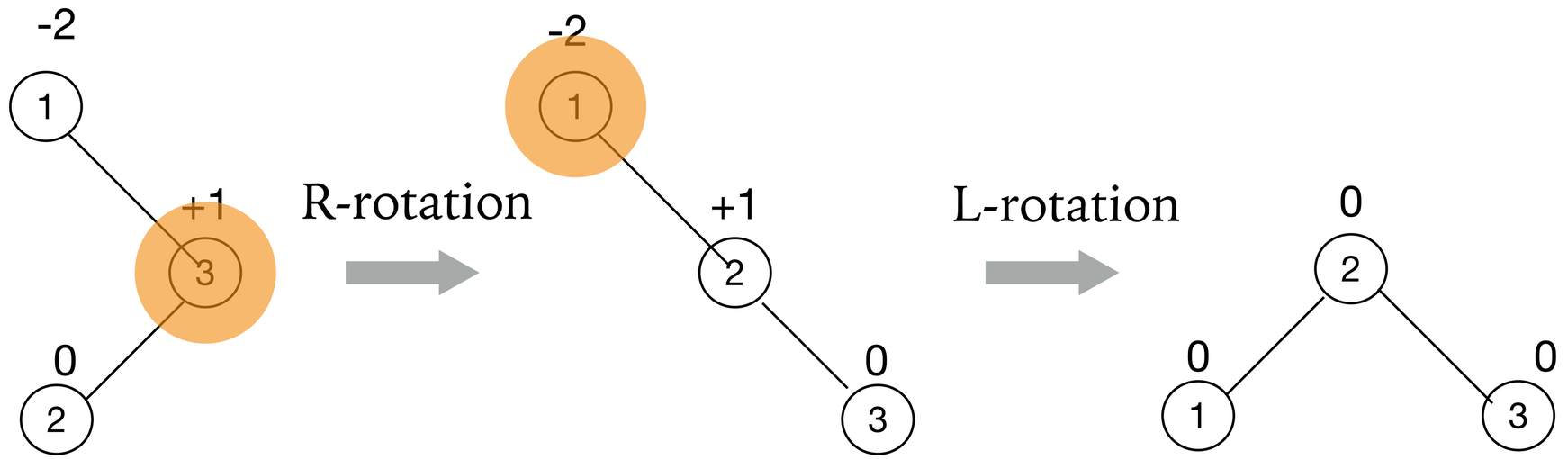


(Step 2)



RL-Rotation in AVL Tree

New key inserted into the *left subtree of the right child* of a tree whose root had the balance of -1 before the insertion.



Double RL-rotation
(R-rotation followed by L-rotation

RL-Rotation in AVL Tree

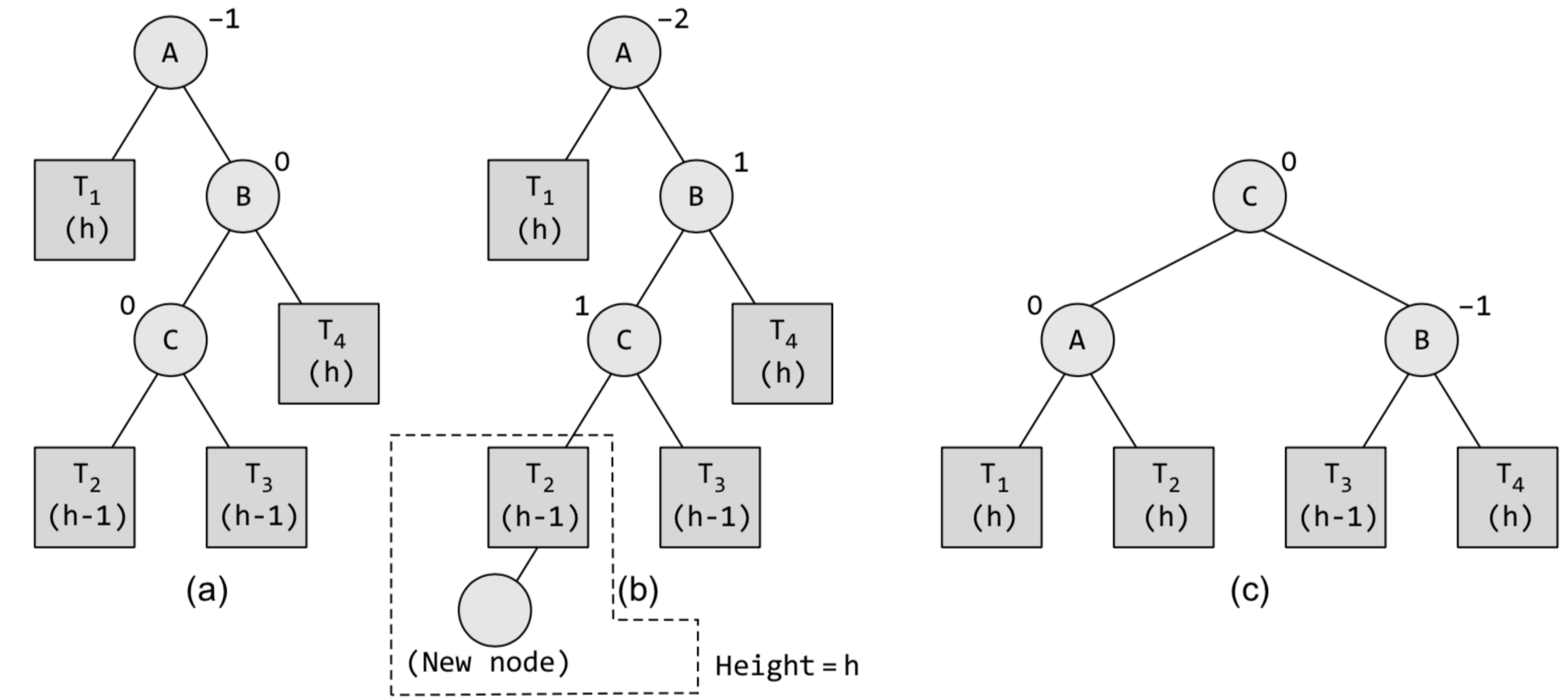
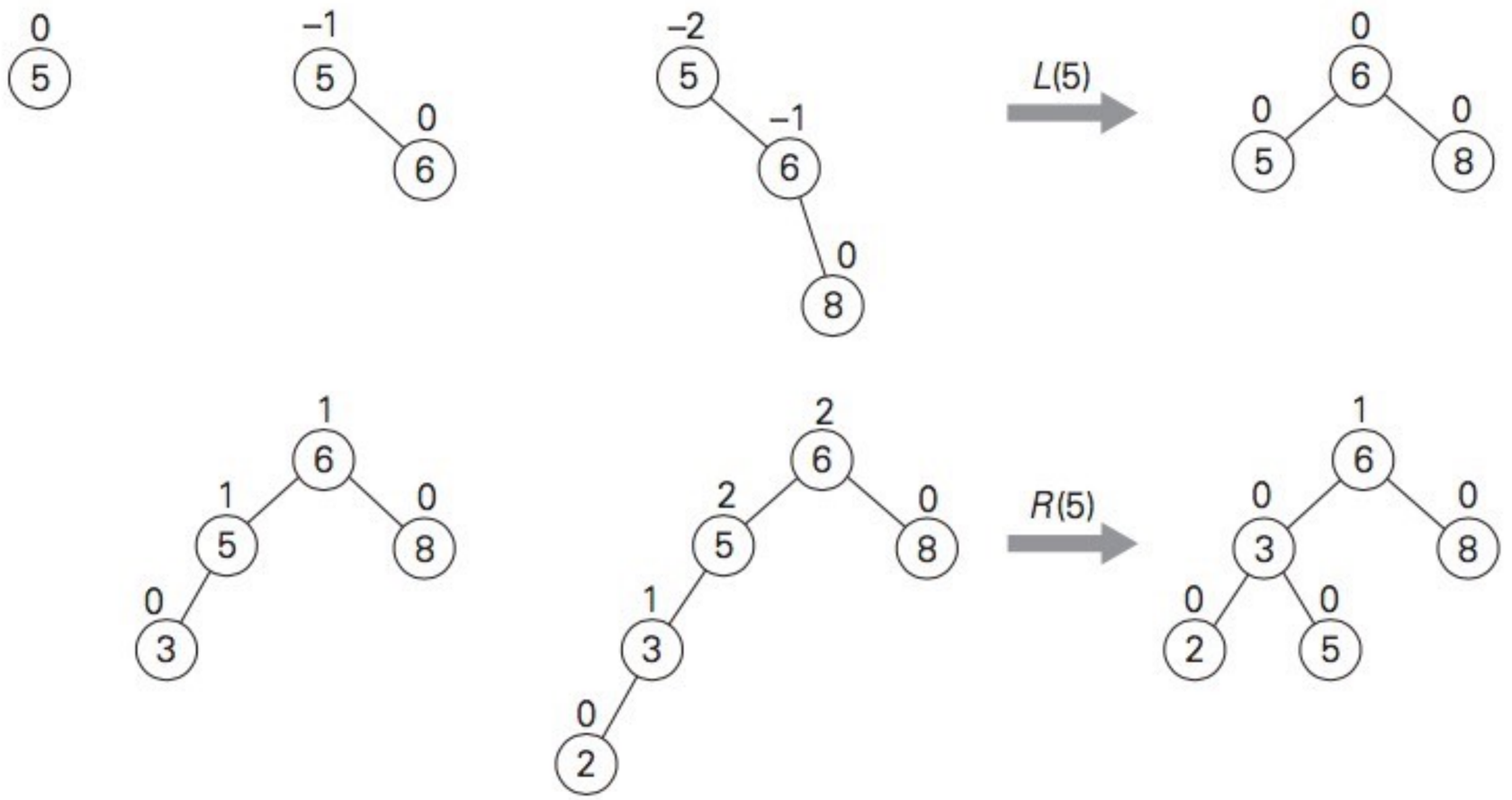


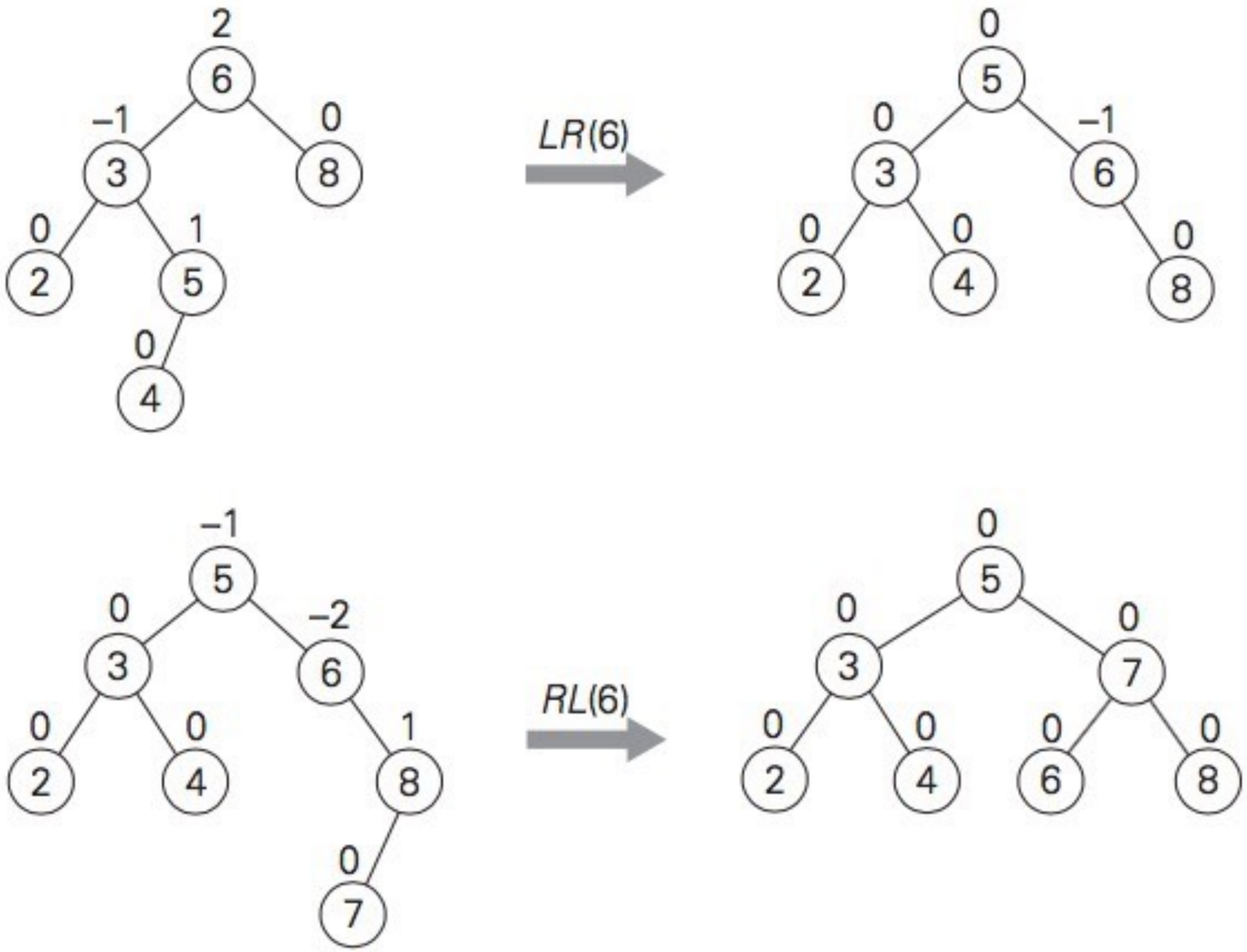
Figure 10.46 RL rotation in an AVL tree

Inserting elements into AVL Tree

Sequence inserted: 5, 6, 8, 3, 2, 4, 7



Inserting elements into AVL Tree



Other Applications of Trees

- Red-Black Tree
- B Tree
- 2-3 Tree