Tree

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. Below Figure-1 shows a tree where node A is the root node, nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

Root node

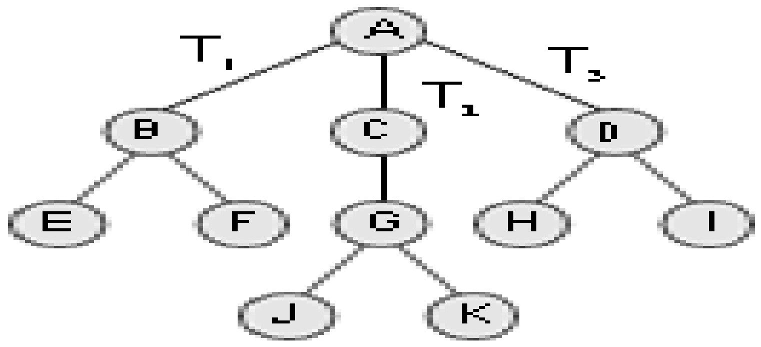


Figure-1 Tree

Contd.

Basic Terminology

Root node: The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

Sub-trees: If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.

Leaf node: A node that has no children is called the leaf node or the terminal node.

Contd.

Path: A sequence of consecutive edges is called a *path.* For example, in Fig.1, the path from the root node A to node I is given as: A, D, and I.

Ancestor node: An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig.1, nodes A, C, and G are the ancestors of node K.

Descendant node: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig.1, nodes C, G, J, and K are the descendants of node A.

Contd.

Path: A sequence of consecutive edges is called a *path.* For example, in Fig.1, the path from the root node A to node I is given as: A, D, and I.

Level number: Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

Degree: Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

In-degree: In-degree of a node is the number of edges arriving at that node.

Out-degree: Out-degree of a node is the number of edges leaving that node.

Contd.

Types of Trees

Trees are of the following types

- General trees
- Forests
- Binary trees
- Binary search trees
- Expression trees
- Tournament trees

Contd.

General trees are data structures that store elements hierarchically. The top node of a tree is the root node and each node, except the root, has a parent. A node in a general tree (except the leaf nodes) may have zero or more sub-trees. General trees which have 3 sub-trees per node are called ternary trees. However, the number of sub-trees for any node may be variable. For example, a node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

Although general trees can be represented as ADTs, there is always a problem when another sub-tree is added to a node that already has the maximum number of sub-trees attached to it.

Contd.

To overcome the complexities of a general tree, it may be represented as a graph data structure (to be discussed later), thereby losing many of the advantages of the tree processes. Therefore, a better option is to convert general trees into binary trees.

A general tree when converted to a binary tree may not end up being well formed or full, but the advantages of such a conversion enable the programmer to use the algorithms for processes that are used for binary trees with minor modifications.

Contd.

A forest is a disjoint union of trees. A set of disjoint trees (or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level 1.

We have already seen that every node of a tree is the root of some sub-tree. Therefore, all the sub-trees immediately below a node form a forest.

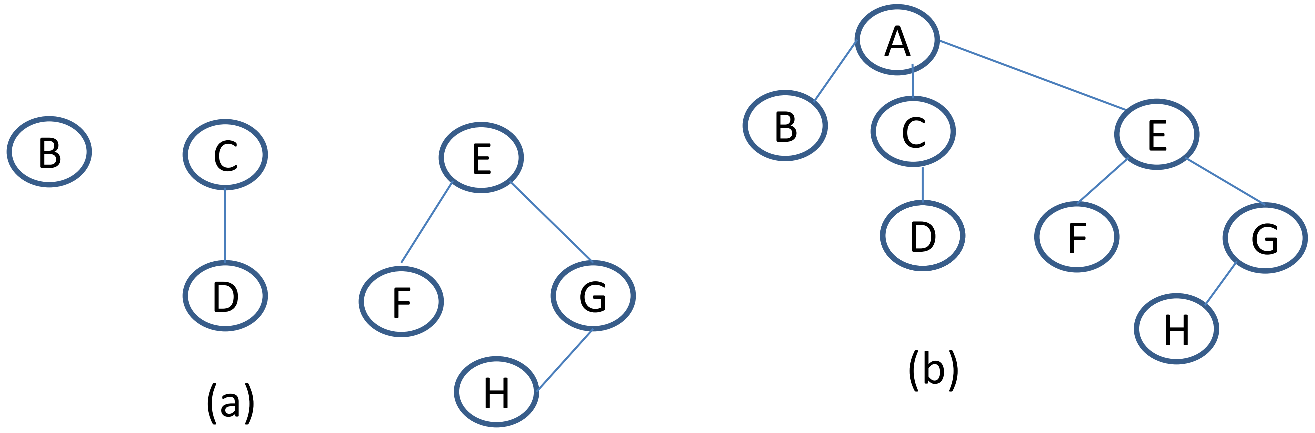


Figure-2 Forest and its corresponding tree

Contd.

A forest can also be defined as an ordered set of zero or more general trees. While a general tree must have a root, a forest on the other hand may be empty because by definition it is a set, and sets can be empty.

We can convert a forest into a tree by adding a single node as the root node of the tree. For example, Figure-2(a) shows a forest and Figure-2(b) shows the corresponding tree.

Similarly, we can convert a general tree into a forest by deleting the root node of the tree.

Binary Trees

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.

Contd.

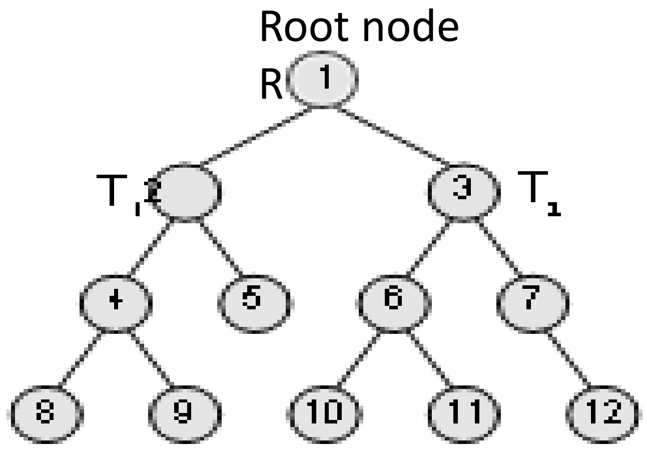


Fig3. Binary Tree

In the figure, R is the root node and the two trees T 1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.

The left sub-tree of the root node consists of the nodes: 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of nodes: 3, 6, 7, 10, 11, and 12.

In the tree, root node 1 has two successors: 2 and 3. Node 2 has two successor nodes: 4 and 5. Node 4 has two successors: 8 and 9. Node 5 has no successor. Node 3 has two successor nodes: 6 and 7. Node 6 has two successors: 10 and 11. Finally, node 7 has only one successor: 12.

Contd.

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. Even the terminal nodes contain an empty left sub-tree and an empty right sub-tree. Fig.3. nodes 5, 8, 9, 10, 11, and 12 have no successors and thus said to have empty sub-trees.

Contd.

Terminology

Parent: If N is any node in T that has left successor S1 right successor S2 , then N is called the parent of S1 and S2. Correspondingly, S1 and S2 are called the left child and the right child of N. Every node other than the root node has a parent.

Level number: Every node in the binary tree is assigned a level number. The root node is defined to be at level 0. The left and the right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as parent's level number + 1.

Contd.

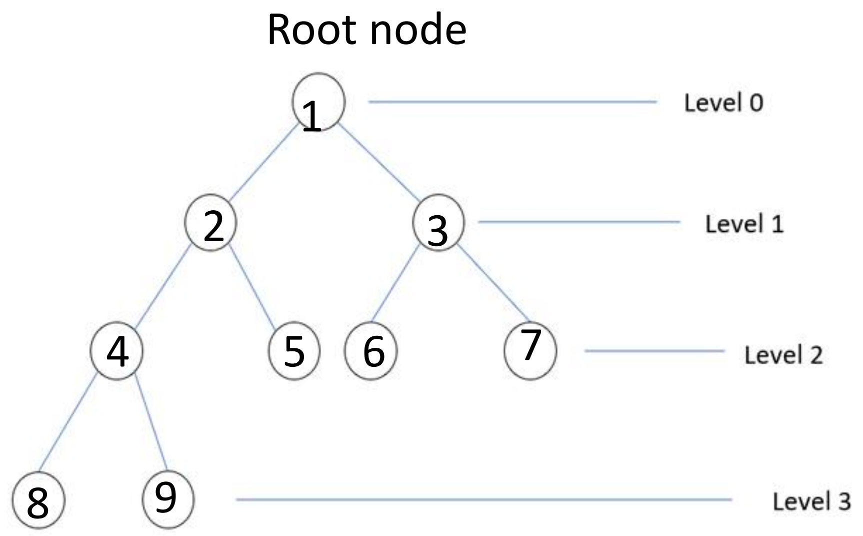


Fig4. Levels in Binary Tree

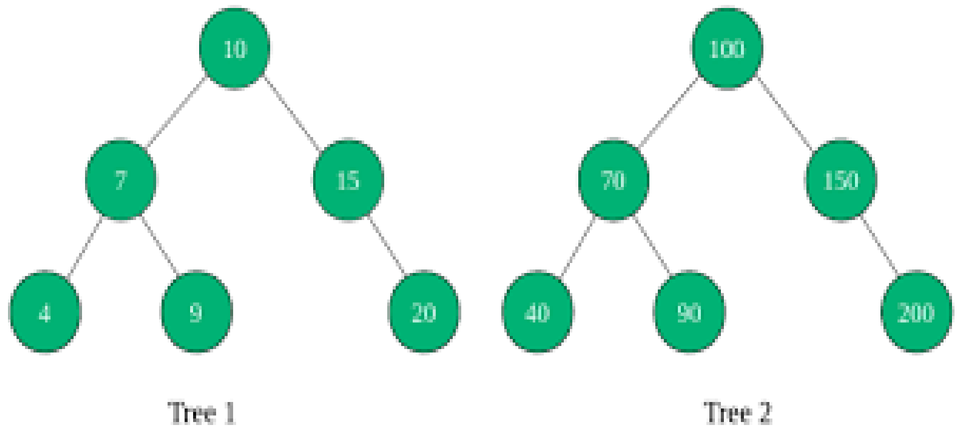
Degree of a node It is equal to the number of children that a node has. The degree of a leaf node is zero. For example, in the tree, degree of node 4 is 2, degree of node 5 is zero.

Sibling All nodes that are at the same level and share the same parent are called siblings. For example, nodes 2 and 3, nodes 4 and 5, nodes 6 and 7, nodes 8 and 9 are siblings.

Contd.

Leaf node A node that has no children is called a leaf node or a terminal node. The leaf nodes in the tree are: 8, 9, 5, 6, 7.

Similar binary trees Two binary trees T1 and T2 are said to be similar if both these trees have the same structure.



Contd.

Copies Two binary trees T1 and T2 are said to be copies if they have similar structure and if they have same content at the corresponding nodes.

Edge It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n – 1 edges because every node except the root node is connected to its parent via an edge.

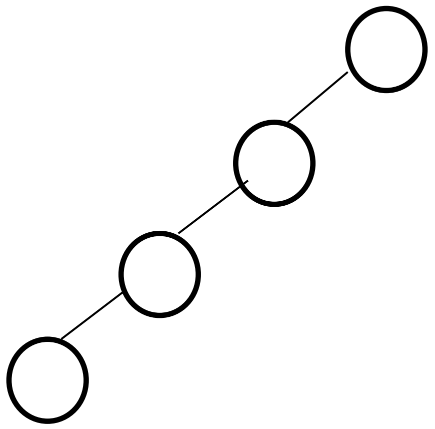
Path A sequence of consecutive edges. For example, in Fig. 4, the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

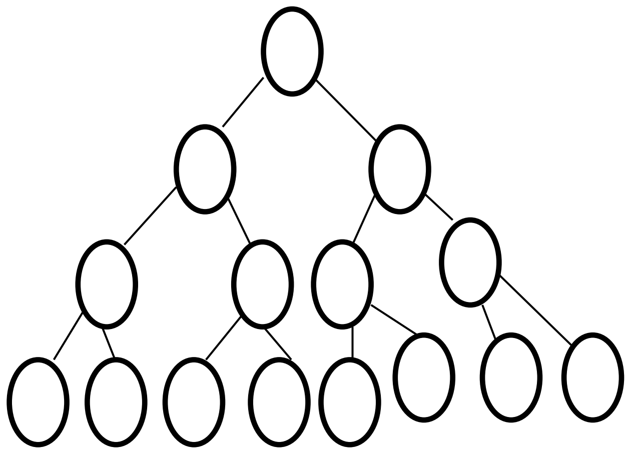
Contd.

Depth The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

Height of a tree It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1. A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. This is because every level will have at least one node and can have at most 2 nodes. So, if every level has two nodes then a tree with height h will have at the most $2^h - 1$ nodes as at level 0, there is only one element called the root. The height of a binary tree with n nodes is at least $\log_2 (n+1)$ and at most n.

Contd.


Height=4, Node =4


Height=4, Node =15

Contd.

In-degree/out-degree of a node It is the number of edges arriving at a node. The root node is the only node that has an in-degree equal to zero. Similarly, out-degree of a node is the number of edges leaving that node.

Binary trees are commonly used to implement binary search trees, expression trees, tournament trees, and binary heaps.

Complete Binary Trees

A *complete binary tree* is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.

In a complete binary tree T_n , there are exactly n nodes and level r of T can have at most 2^r nodes.

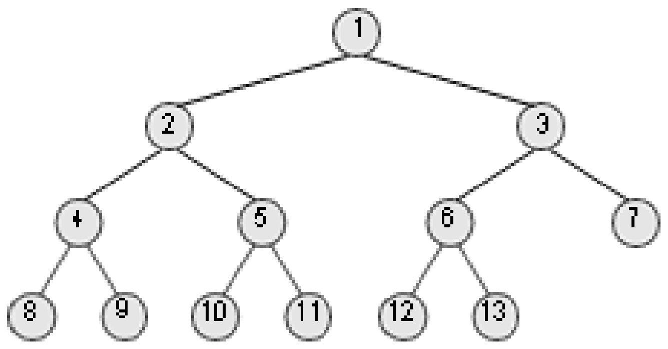


Fig.5. Complete Binary Tree

Contd.

In Fig.5. level 0 has $2^0 = 1$ node, level 1 has $2^1 = 2$ nodes, level 2 has $2^2 = 4$ nodes, level 3 has 6 nodes which is less than the maximum of $2^3 = 8$ nodes.

If K is a parent node, then its left child can be calculated as $2 \times K$ and its right child can be calculated as $2 \times K + 1$. For example, the children of the node 4 are 8 (2×4) and 9 ($2 \times 4 + 1$). Similarly, the parent of the node K can be calculated as $|K/2|$. Given the node 4, its parent can be calculated as $|4/2| = 2$. The height of a tree T_n having exactly n nodes is given as:

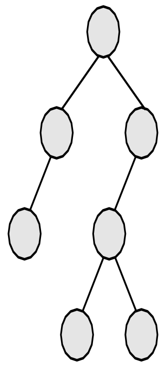
$$H_n = |\log_2 (n + 1)|$$

If a tree T has 10,00,000 nodes, then its height is 21.

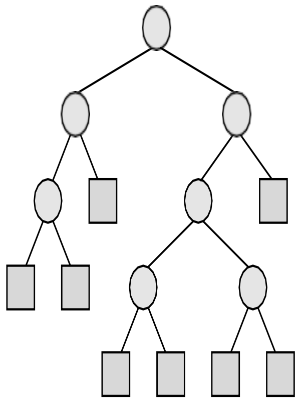
Contd.

Extended Binary Trees

A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children.



(a)



(b)

Fig.6 (a) Binary Tree (b) Extended Binary Tree

Contd.

In an extended binary tree, nodes having two children are called internal nodes and nodes having no children are called external nodes. In Fig.6(b), the internal nodes are represented using circles and the external nodes are represented using squares.

To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.

Contd.

Representation of Binary Trees in the Memory

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.

Linked representation of binary trees: In the linked representation of a binary tree, every node will have three parts: **the data element**, **a pointer to the left node**, and **a pointer to the right node**. So in C, the binary tree is built with a node type given below.

```
struct node {
struct node *left;
int data;
struct node *right;
};
```

Contd.

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty.

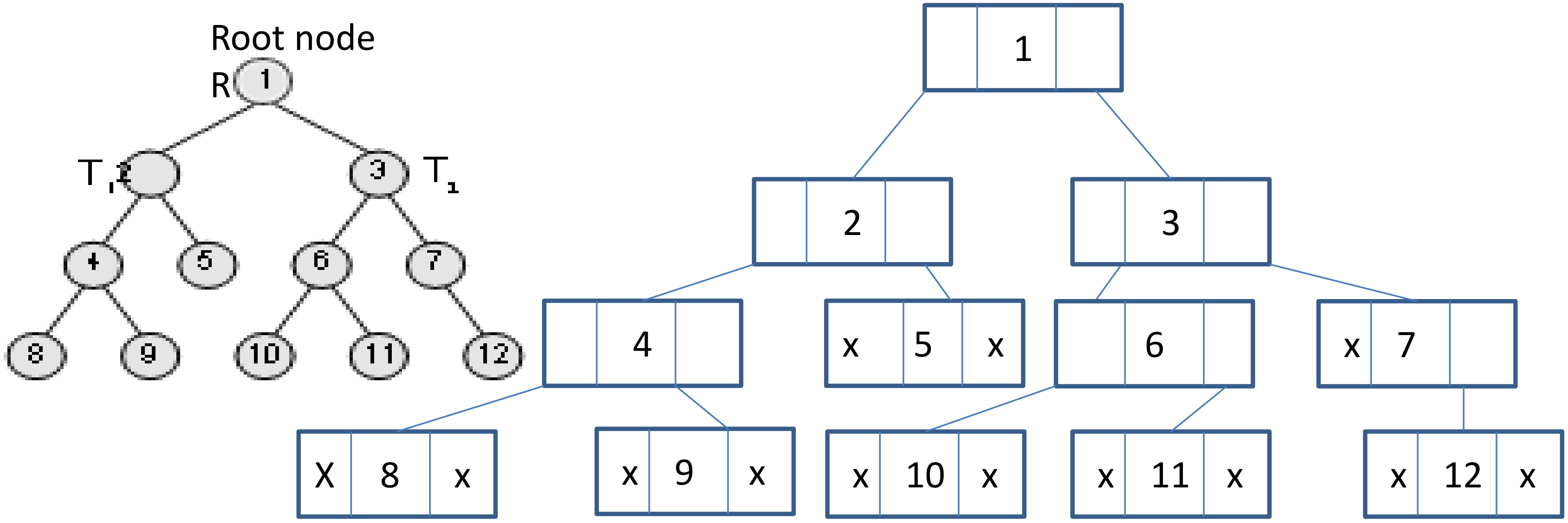


Fig.7. Linked representation of a binary tree

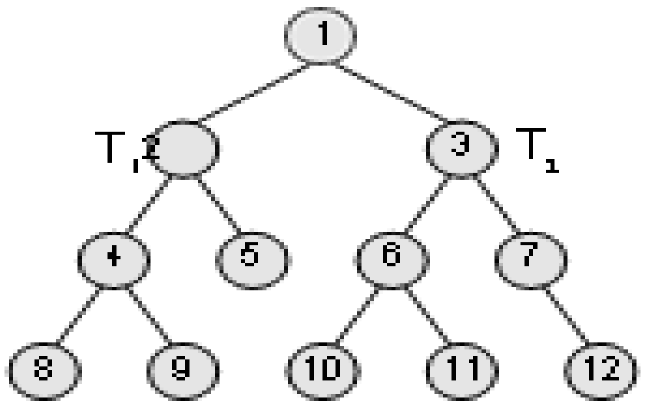
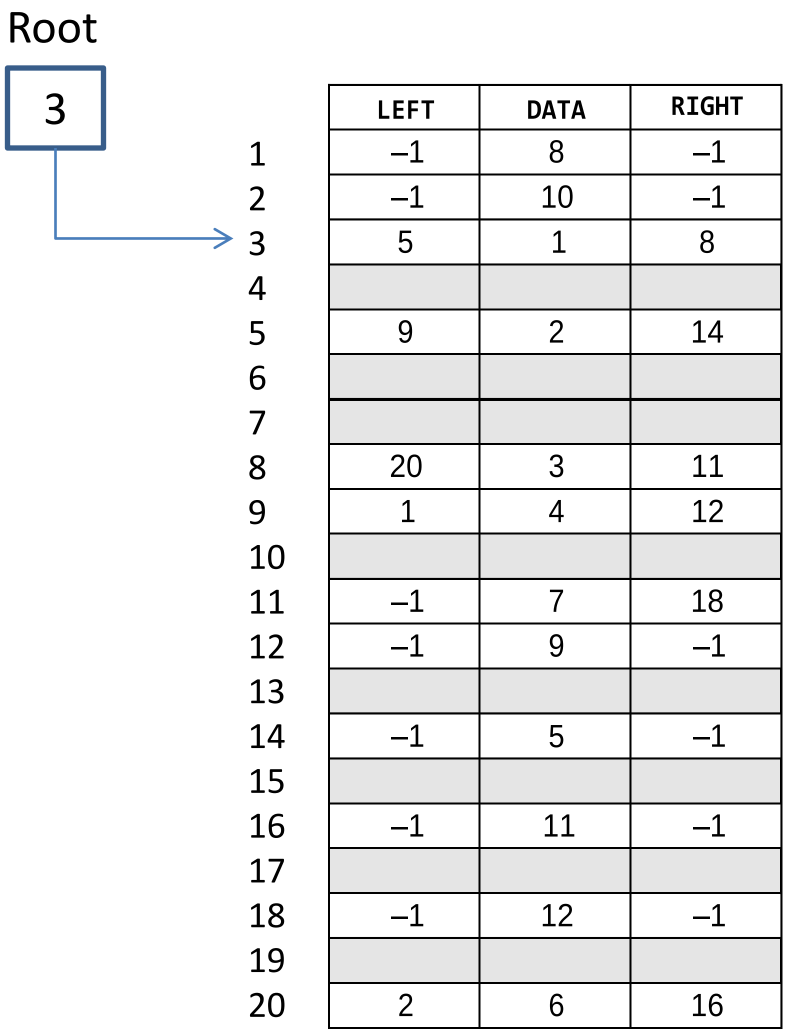


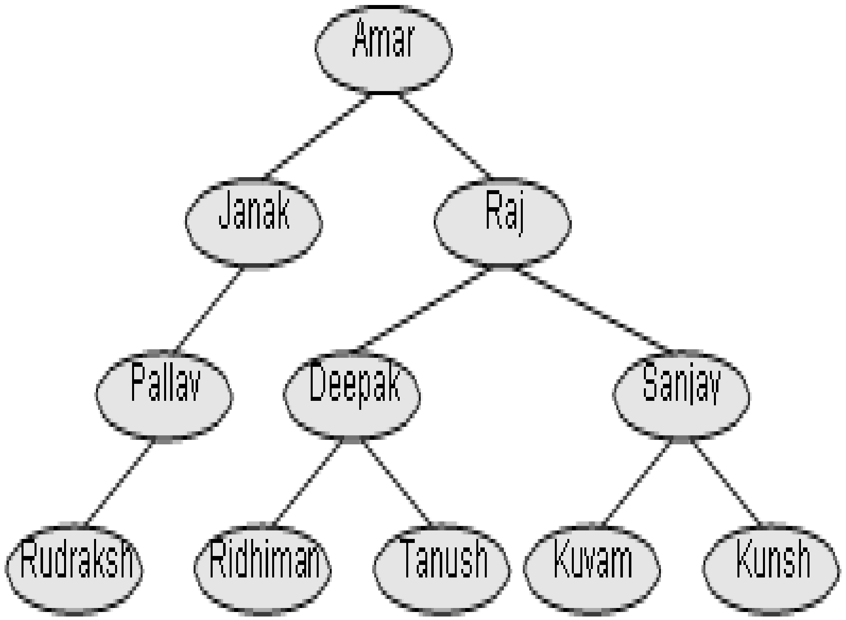
Fig.8. binary tree T



	LEFT	DATA	RIGHT
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

Fig.9. Linked representation of binary tree T

Construct the corresponding Tree from given data



Root

$\boxed{3} \rightarrow 3$

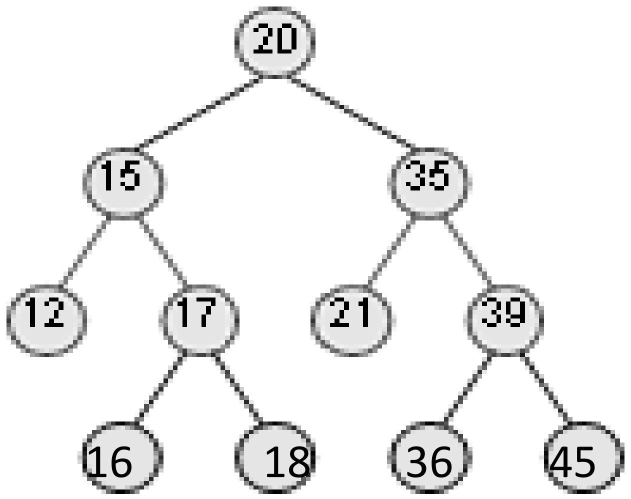
	LEFT	NAMES	RIGHT
1	12	Pallav	–1
2			
3	9	Amar	13
4			
5			
6	19	Deepak	17
7			
8			
9	1	Janak	–1
10			
11	–1	Kuvam	–1
12	–1	Rudraksh	–1
13	6	Raj	20
14			
15	–1	Kunsh	–1
16			
17	–1	Tanush	–1
18			
19	–1	Ridhiman	–1
20	11	Sanjay	15

Contd.

Sequential representation of binary trees

Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space. A sequential binary trees follows the following rules:

- A one-dimensional array, called TREE, is used to store the elements of tree
- The root of the tree will be stored in the first location, that is TREE[1] will store the data of the root element
- The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1)
- The maximum size of the array TREE is given as (2^h–1), where h is the height of the tree
- An empty tree or sub-tree is specified using NULL. If TREE[1]= NULL, then the tree is empty



TREE[]

1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

Fig.10. Binary tree and its sequential representation

Contd.

Expression Trees

Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression given as:

Exp = (a – b) + (c * d)

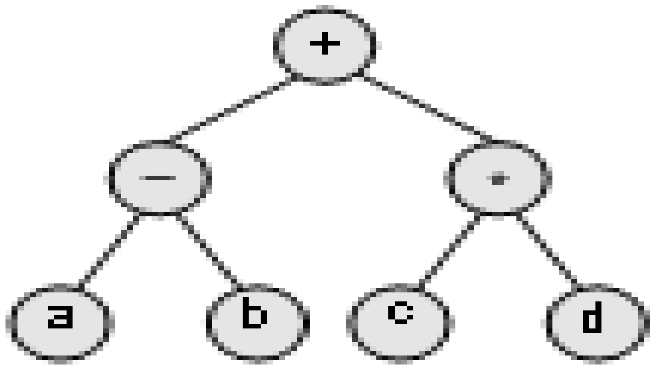


Fig.11. Expression Tree

Contd.

Given an expression, Exp = ((a + b) – (c * d)) % ((f ^ g) / (h – i)), construct the corresponding binary tree

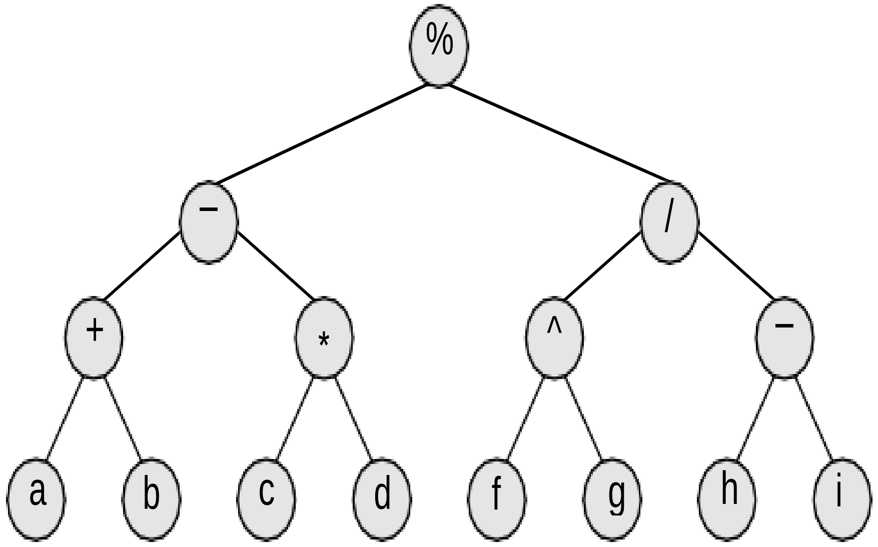


Fig.12. Expression Tree

Contd.

Given the binary tree, write down the expression that it represents.

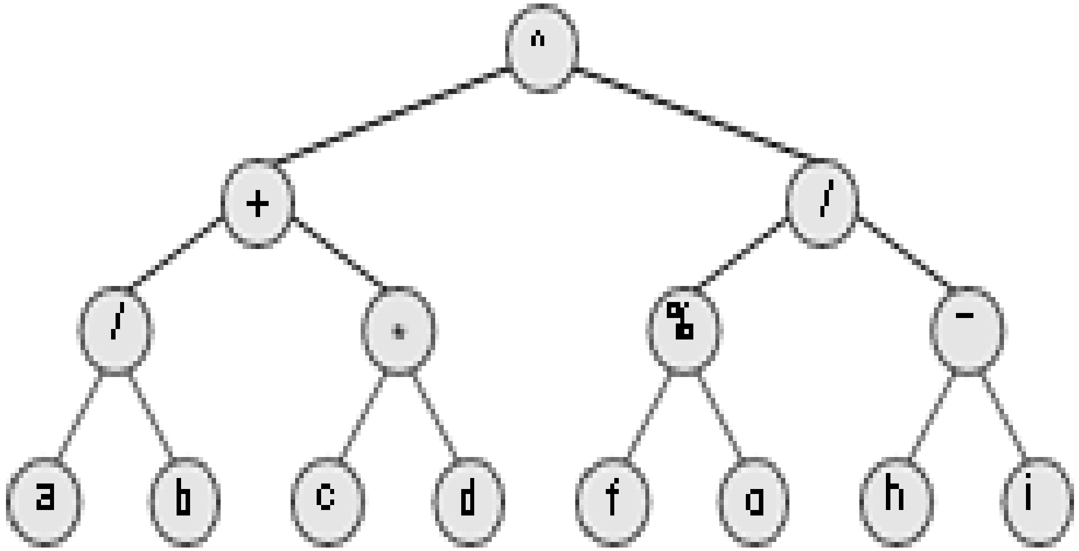


Fig.13. Expression Tree

Expression for the above binary tree is

[{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

Contd.

Given the expression, Exp = a + b / c * d – e, construct the corresponding binary tree.

Use the operator precedence chart to find the sequence in which operations will be performed. The given expression can be written as Exp = ((a + ((b/c) * d)) – e)

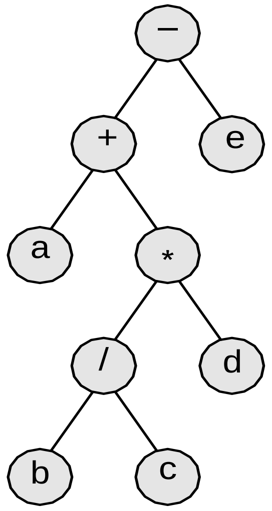


Fig.14. Expression Tree

Contd.

Tournament Trees

In a tournament, say of chess, n number of players participate. To declare the winner among all these players, a couple of matches are played and usually three rounds are played in the game.

In every match of round 1, a number of matches are played in which two players play the game against each other. The number of matches that will be played in round 1 will depend on the number of players. For example, if there are 8 players participating in a chess tournament, then 4 matches will be played in round 1. Every match of round 1 will be played between two players.

Then in round 2, the winners of round 1 will play against each other. Similarly, in round 3, the winners of round 2 will play against each other and the person who wins round 3 is declared the winner. Tournament trees are used to represent this concept.

Contd.

In a tournament tree (also called a selection tree), each external node represents a player and each internal node represents the winner of the match played between the players represented by its children nodes. These tournament trees are also called winner trees because they are being used to record the winner at each level. We can also have a loser tree that records the loser at each level.

Contd.

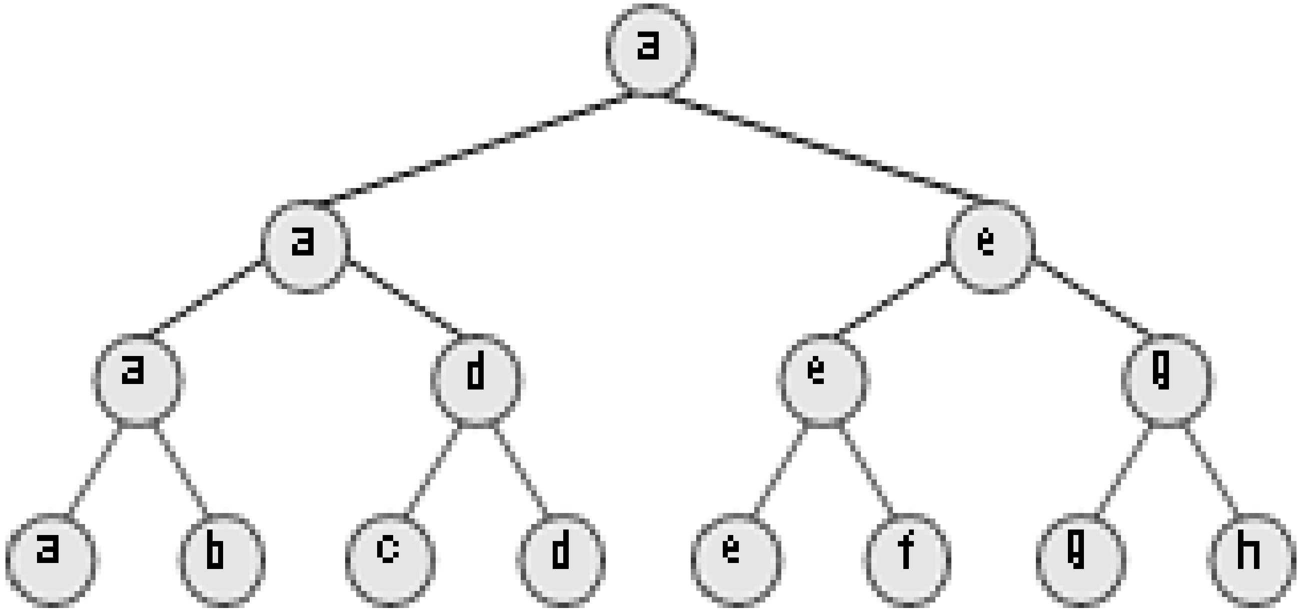


Fig.15. Tournament Tree

There are 8 players in total whose names are represented using a, b, c, d, e, f, g, and h. In round 1, a and b; c and d; e and f; and finally g and h play against each other. In round 2, the winners of round 1, that is, a, d, e, and g play against each other. In round 3, the winners of round 2, a and e play against each other. Whoever wins is declared the winner. In the tree, the root node a specifies the winner.

Contd.

Creating a Binary Tree from General Tree

The rules for converting a general tree to a binary tree are given below. Note that a general tree is converted into a binary tree and not a binary search tree.

- ***Rule 1:*** Root of the binary tree = Root of the general tree
- ***Rule 2:*** Left child of a node in the binary tree = Leftmost child of the node in the general tree
- ***Rule 3:*** Right child of a node in the binary tree = Right sibling of the node in the general tree

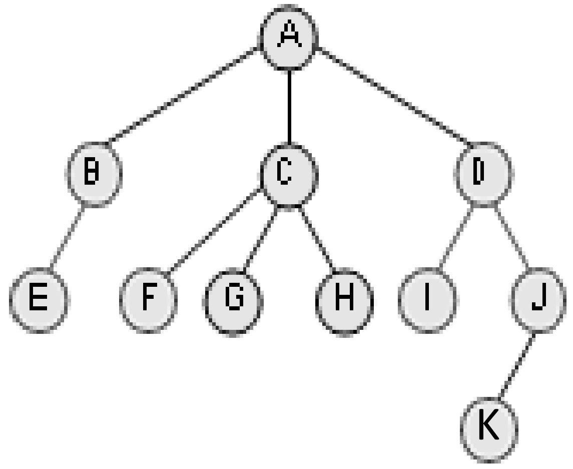


Fig.16. General Tree

Build the binary tree:

Step 1: Node A is the root of the general tree, so it will also be the root of the binary tree.

Step 2: Left child of node A is the leftmost child of node A in the general tree and right child of node A is the right sibling of the node A in the general tree. Since node A has no right sibling in the general tree, it has no right child in the binary tree.

Step 3: Now process node B. Left child of B is E and its right child is C (right sibling in general tree).

Contd.

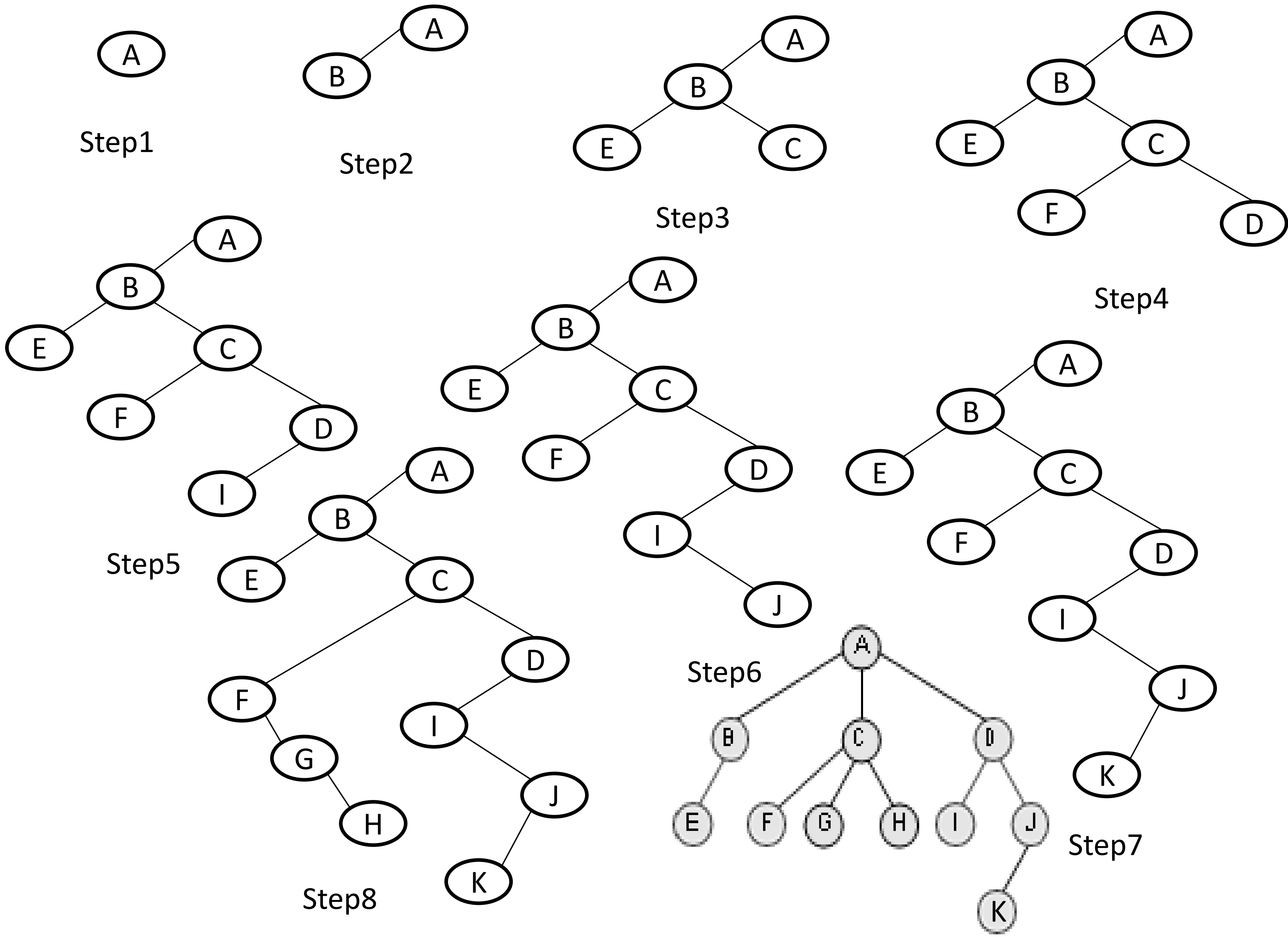
Step 4: Now process node C. Left child of C is F (leftmost child) and its right child is D(right sibling in general tree).

Step 5: Now process node D. Left child of D is I (leftmost child). There will be no right child of D because it has no right sibling in the general tree.

Step 6: Now process node I. There will be no left child of I in the binary tree because I has no left child in the general tree. However, I has a right sibling J, so it will be added as the right child of I.

Step 7: Now process node J. Left child of J is K (leftmost child). There will be no right child of J because it has no right sibling in the general tree.

Step 8: Now process all the unprocessed nodes (E, F, G, H, K) in the same fashion, so the resultant binary tree can be given as follows.


Step1
Step2
Step3
Step4
Step5
Step6
Step7
Step8

Contd.

Traversing a Binary Tree

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a non- linear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.

Contd.

Traversing a Binary Tree

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a non- linear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.

Contd.

Pre-order Traversal

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by

1. Visiting the root node
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

Contd.

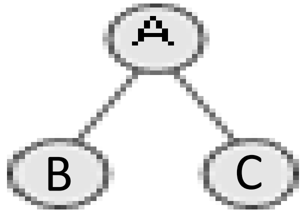


Fig.17. Binary Tree

The pre-order traversal of the tree is given as A, B, C. Root node first, the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called as *depth-first traversal*. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right).

Contd.

Algorithm for Preorder Traversal

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

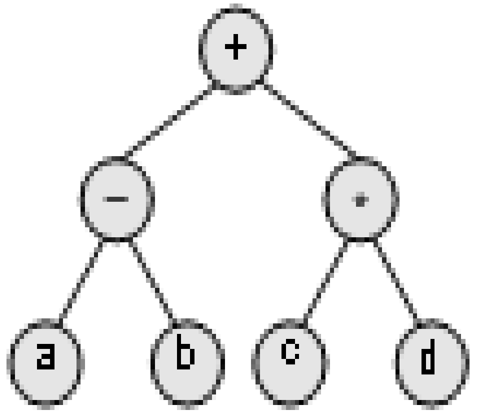
Contd.

Pre-order traversal algorithms are used to extract a prefix notation from an expression tree. For example, consider the expressions given below. When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression.

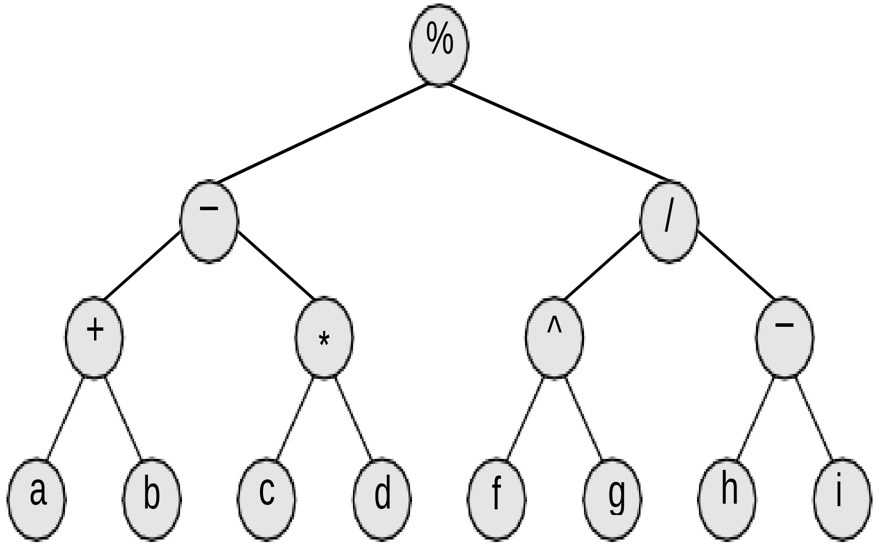
+ – a b * c d (from Fig.11)

% – + a b * c d / ^ e f – g h (from Fig.12)

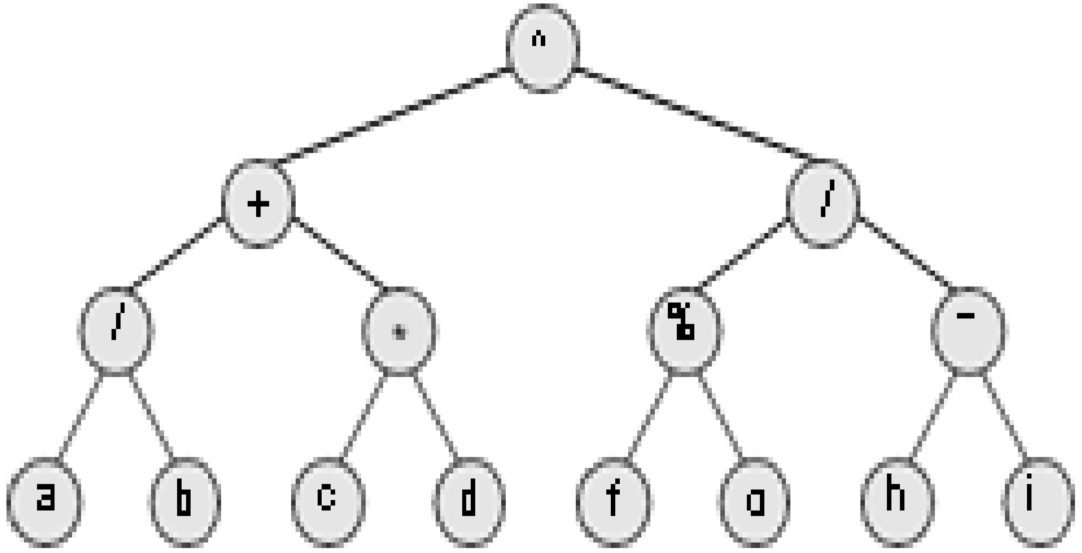
^ + / a b * c d / % f g – h i (from Fig.13)



+ – a b * c d



% – + a b * c d / ^ f g – h i



^ + / a b * c d / % f g – h i

Contd.

Find the sequence of nodes that will be visited using pre-order traversal algorithm of the following Binary Tree.

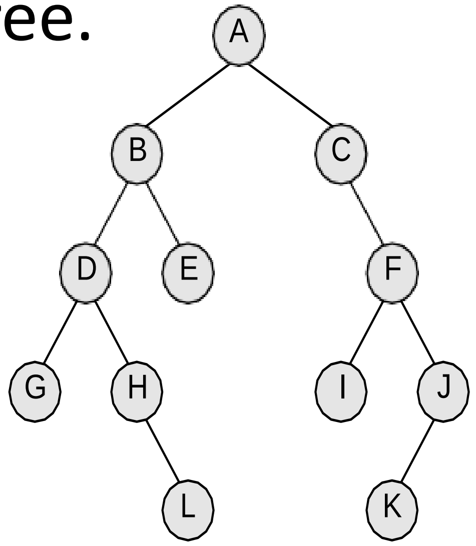


Fig.18.a)

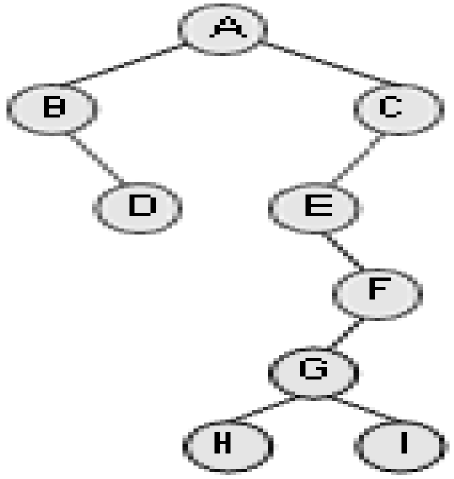


Fig.18.b)

a) TRAVERSAL ORDER: A, B, D, G, H, L, E, C, F, I, J, and K

b) TRAVERSAL ORDER: A, B, D, C, E, F, G, H, and I

Contd.

Algorithm for Preorder Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:  Write TREE -> DATA
Step 3:  PREORDER(TREE -> LEFT)
Step4: PREORDER(TREE -> RIGHT)
            [END OF LOOP]
Step 5: END.
```

Contd.

In-order Traversal

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

Contd.

The in-order traversal of the tree in Fig.17. is given as B, A, and C. Left sub-tree first, the root node next, and then the right sub-tree. In-order traversal is also called as *symmetric traversal*. In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree. The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).

Contd.

Algorithm for In-order Traversal

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

In-order traversal algorithm is usually used to display the elements of a binary search tree. Here, all the elements with a value lower than a given value are accessed before the elements with a higher value.

Contd.

Find the sequence of nodes that will be visited using in-order traversal algorithm of the following Binary Tree.

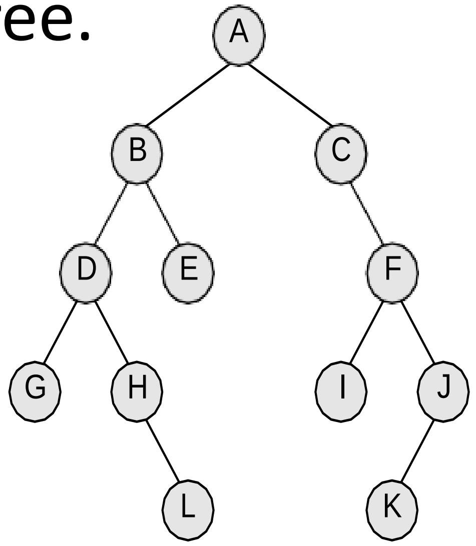


Fig.18.a)

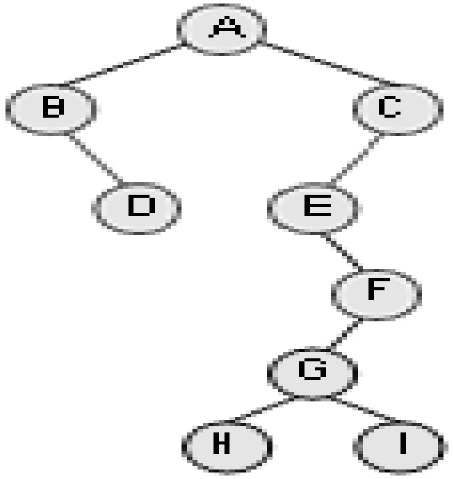


Fig.18.b)

a) TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J

b) TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C

Contd.

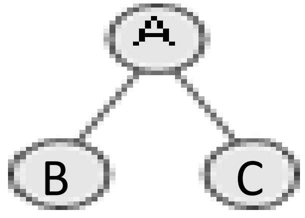
Post-order Traversal

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

Contd.

The post-order traversal of the tree shown below is given as B, C, and A. Left sub-tree first, the right sub-tree next, and finally the root node. In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word ‘post’ in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-Node)



Contd.

Algorithm for Post-order Traversal

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

Contd.

Find the sequence of nodes that will be visited using post-order traversal algorithm of the following Binary Tree.

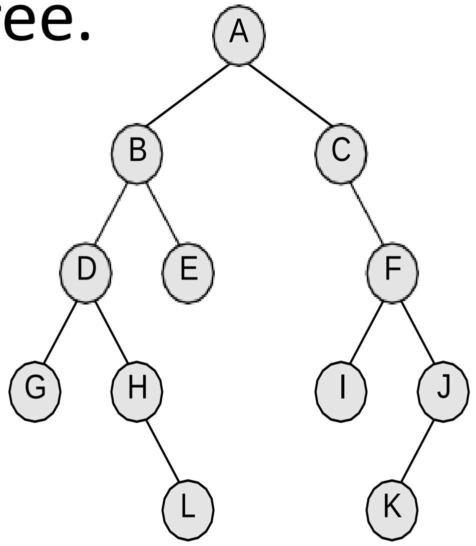


Fig.18.a)

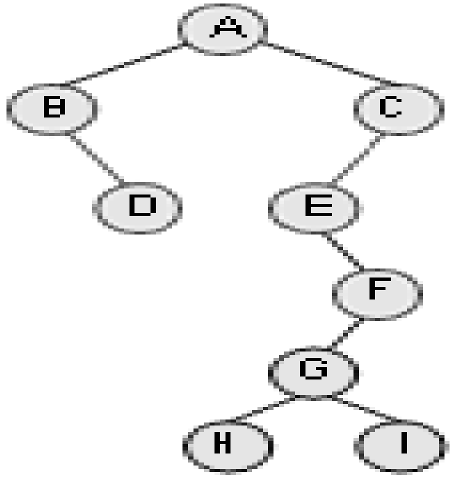


Fig.18.b)

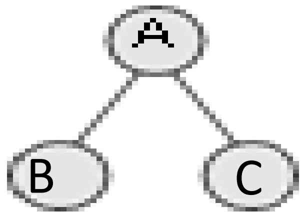
a) TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A

b) TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A

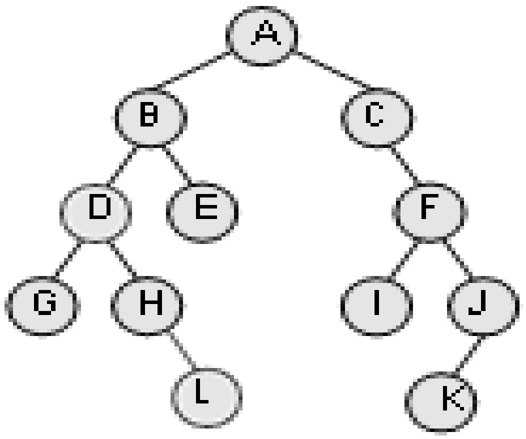
Contd.

Level-order Traversal

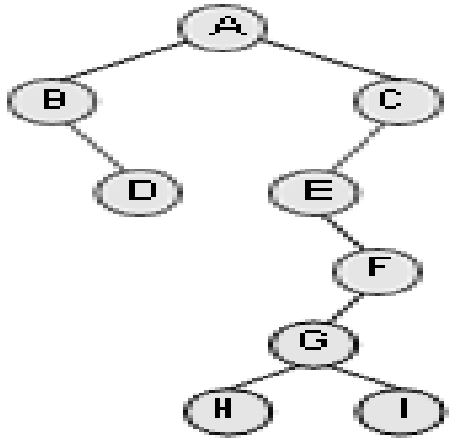
In level-order traversal, all the nodes at a level are accessed before going to the next level. This algorithm is also called as the *breadth-first traversal algorithm*. Consider the trees given below:



a)



b)



c)

TRAVERSAL ORDER:
A, B, and C

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, I, J, L, and K

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, and I

Contd.

Constructing a Binary Tree from Traversal Results

We can construct a binary tree if we are given at least two traversal results. The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal. The in-order traversal result will be used to determine the left and the right child nodes, and the pre-order/post-order can be used to determine the root node.

Contd.

For example, consider the traversal results given below:

In–order Traversal : D B E A F C G

Pre–order Traversal : A B D E C F G

Steps to construct the tree:

Step1: Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.

Step2: Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the root node in the in-order traversal sequence form the right sub-tree of the root node.

Step3: Recursively select each element from pre-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence.

Contd.

In–order Traversal: D B E A F C G

Pre–order Traversal: A B D E C F G

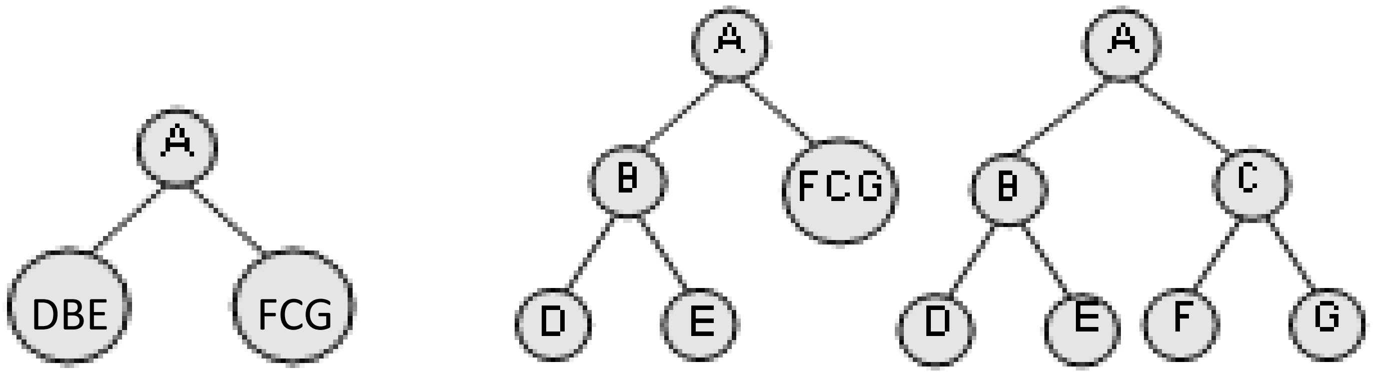


Fig.19. Construction the tree from its traversal result

Contd.

In-order Traversal : D B H E I A F J C G

Post-order Traversal: D H I E B J F G C A

Note: In post-order traversal the root node is the last node.

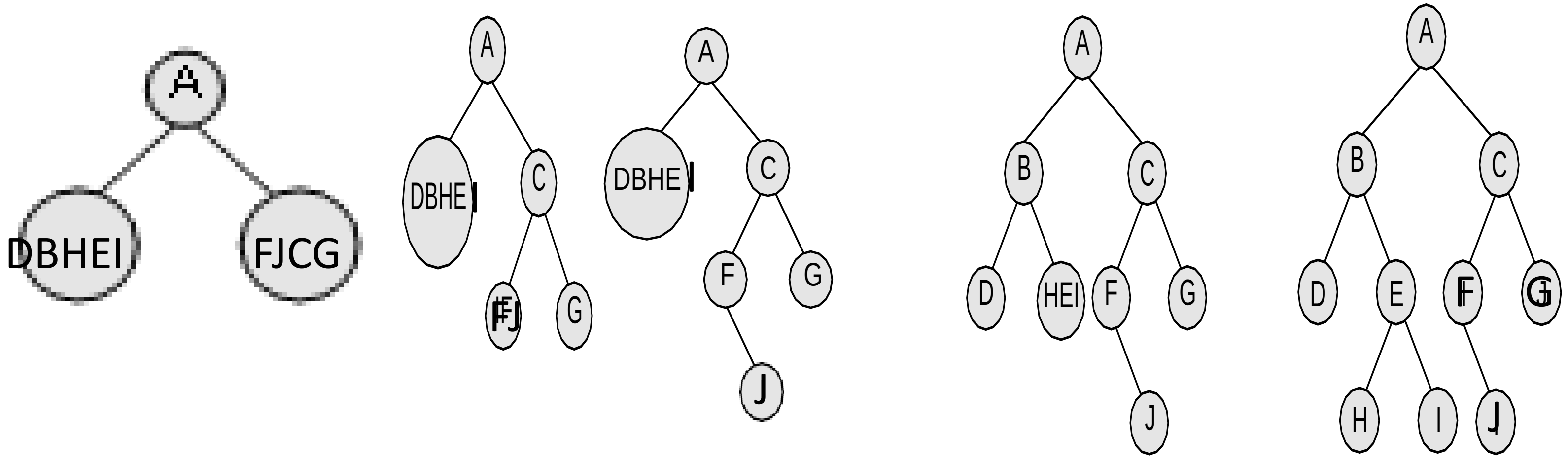
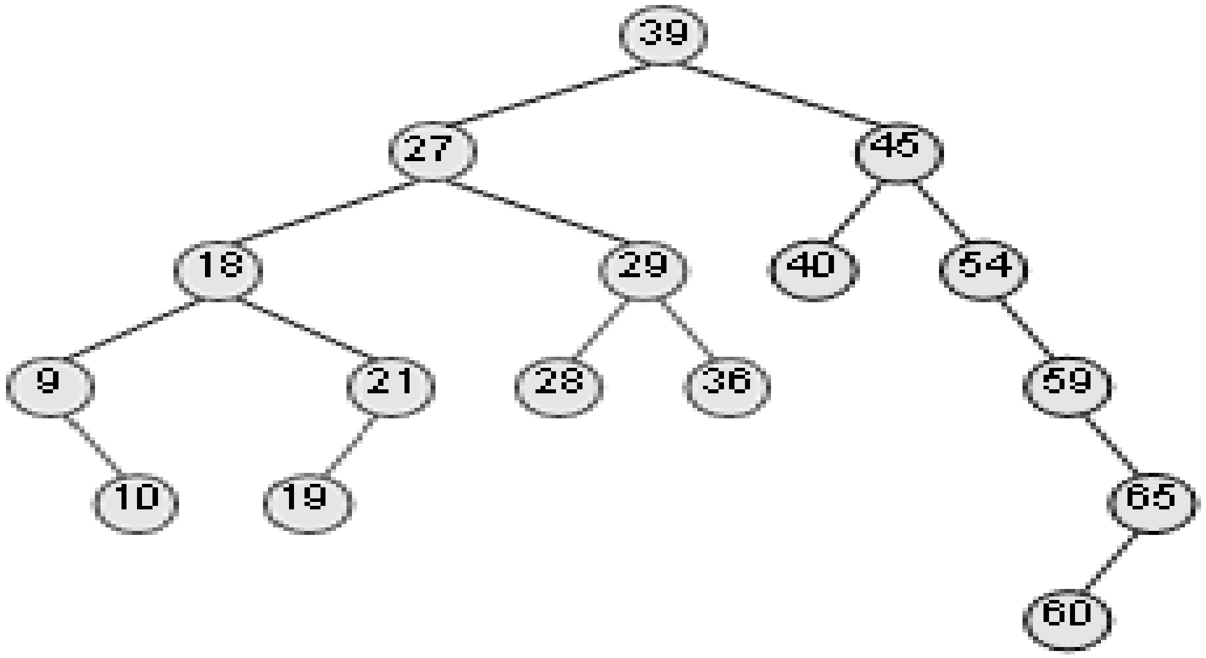


Fig.19. Construction the tree from its traversal result

BINARY SEARCH TREES

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.)

Contd.



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.

Contd.

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.

Contd.

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.

Contd.

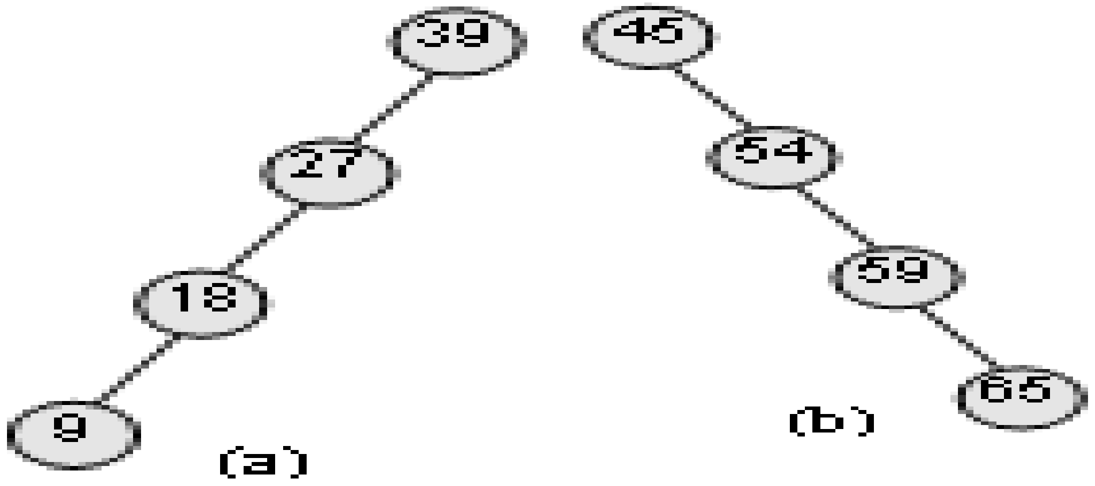
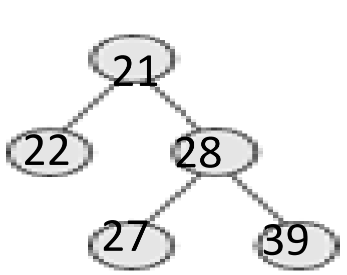
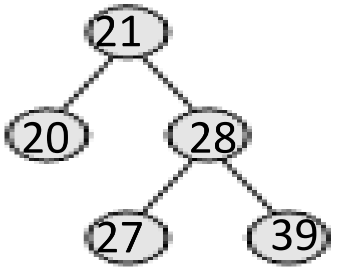
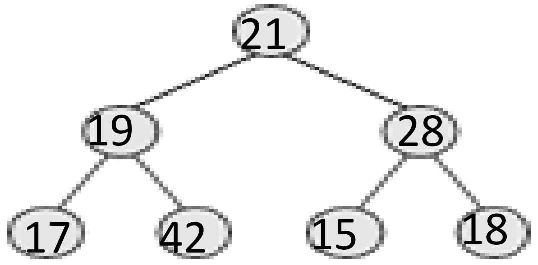


Fig.20. (a) Left skewed and (b) right skewed binary search trees

Check the following Tree is BST or not ?



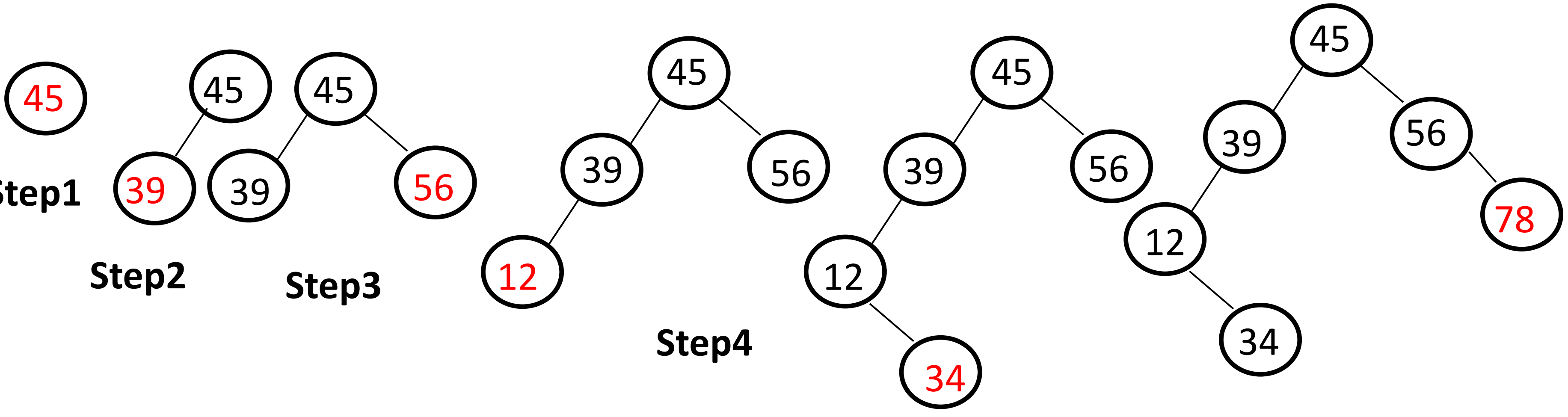




No Yes No

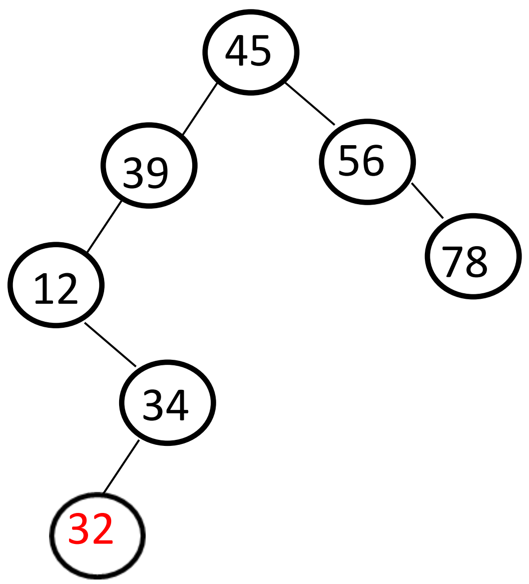
Create a binary search tree using the following data elements:

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

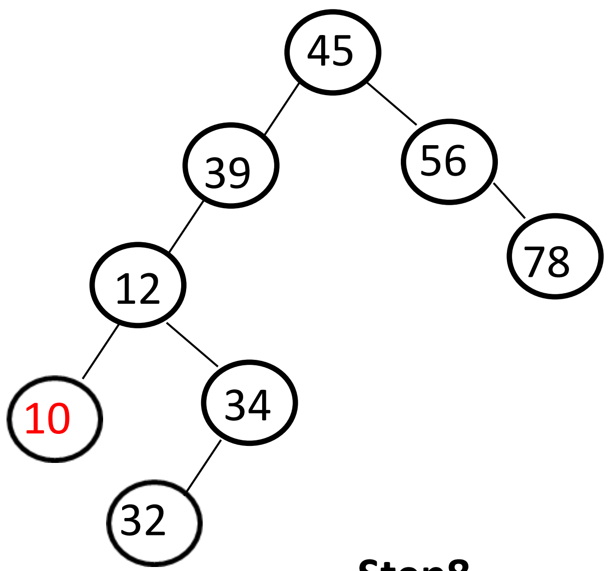


Step1 **Step2** **Step3** **Step4** **Step5** **Step6**

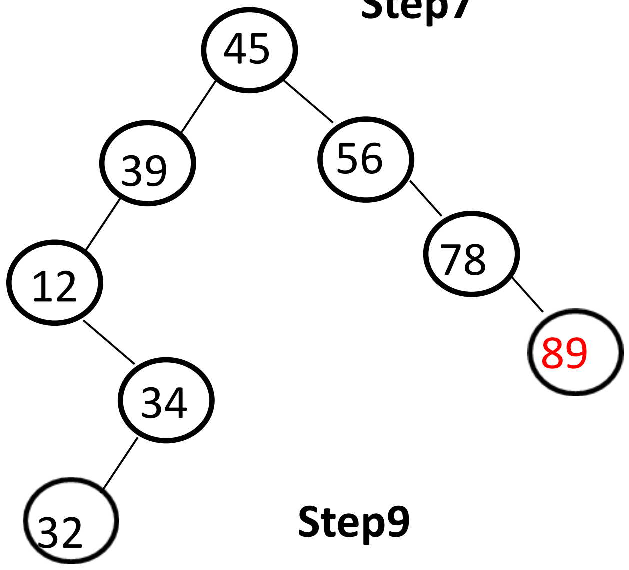
Contd.



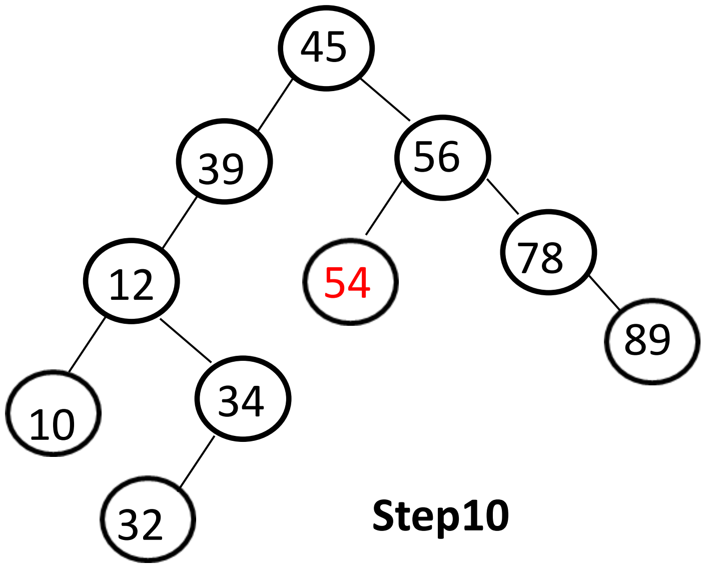
Step7



Step8



Step9



Step10

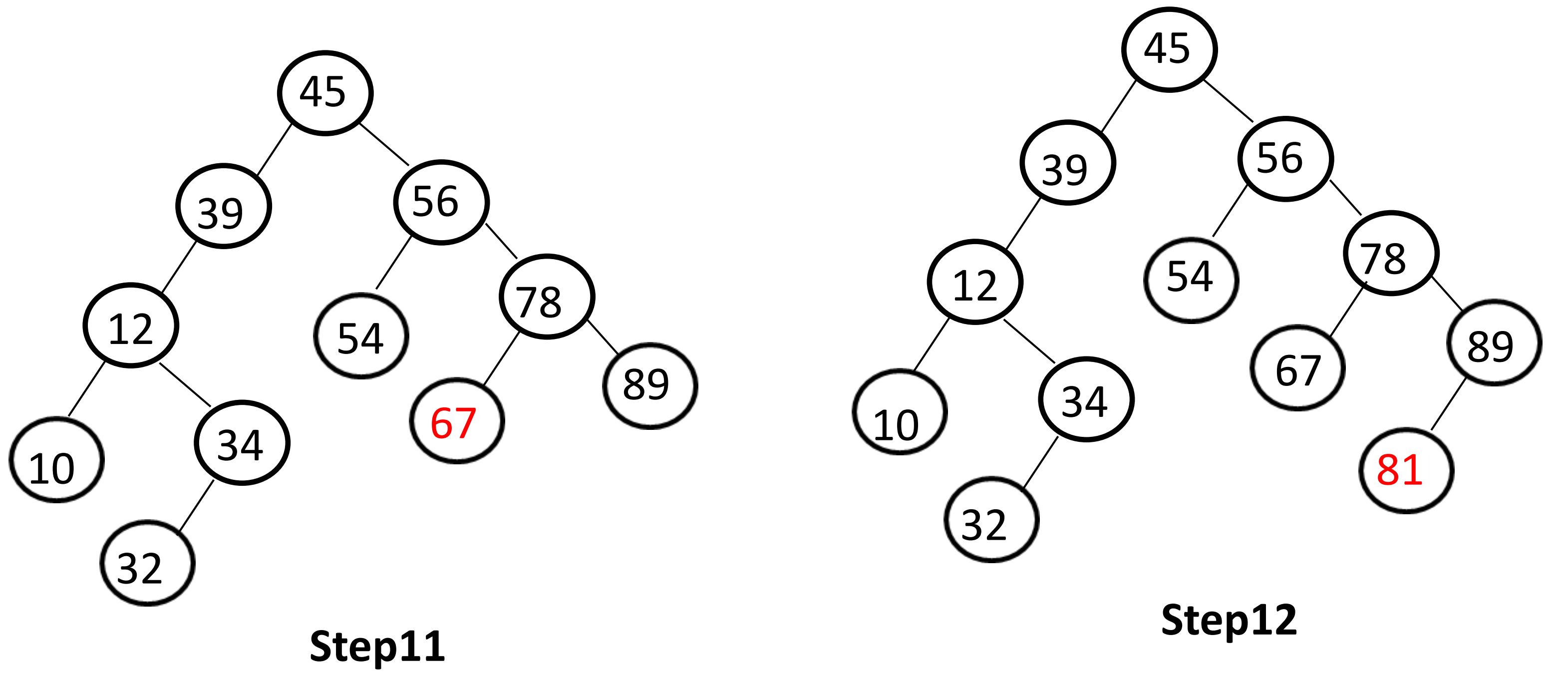


Fig21. Binary Search Tree

Contd.

Operations on Binary Search Tree

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.

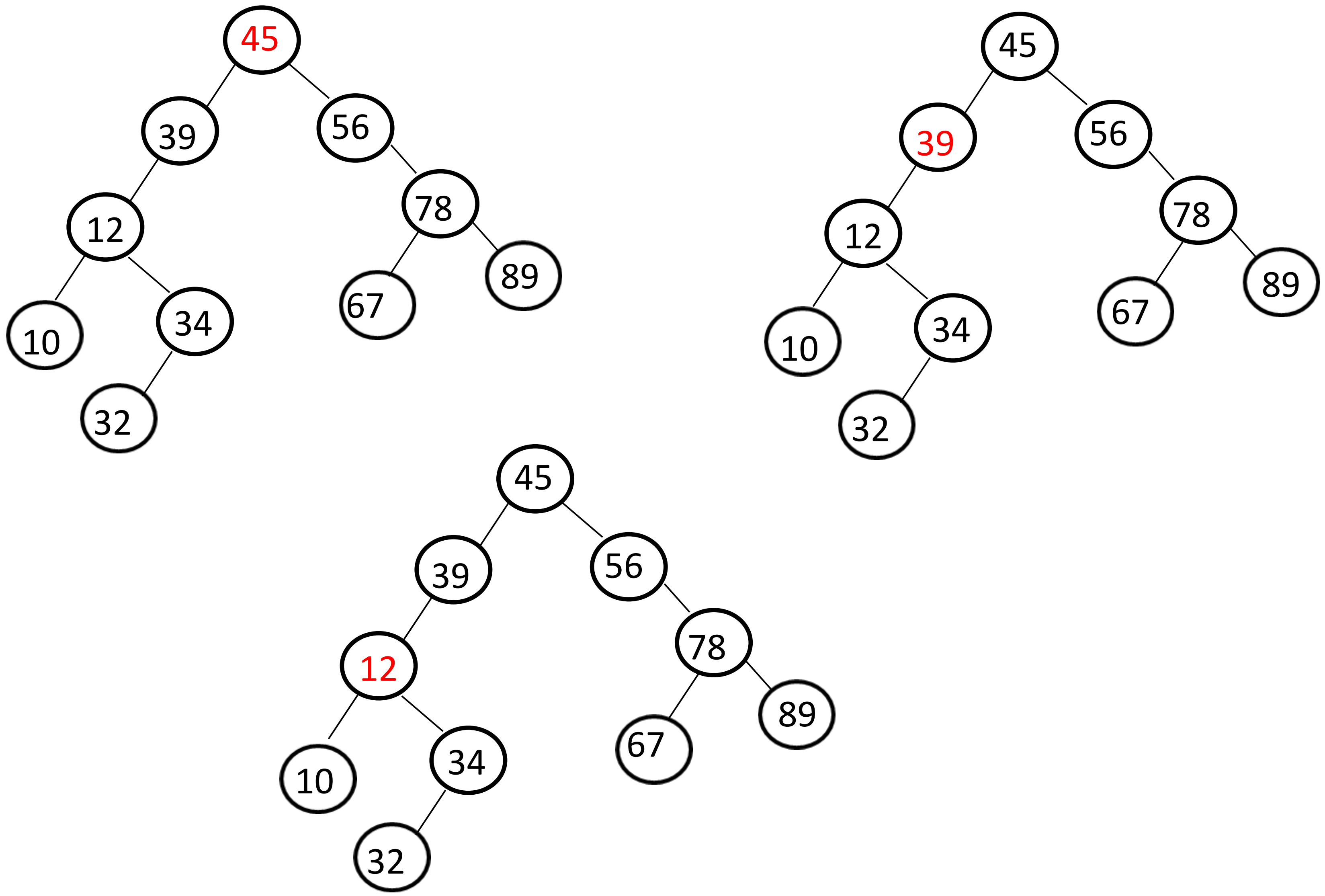


Fig22. Searching a node with value 12 in the given Binary Search Tree

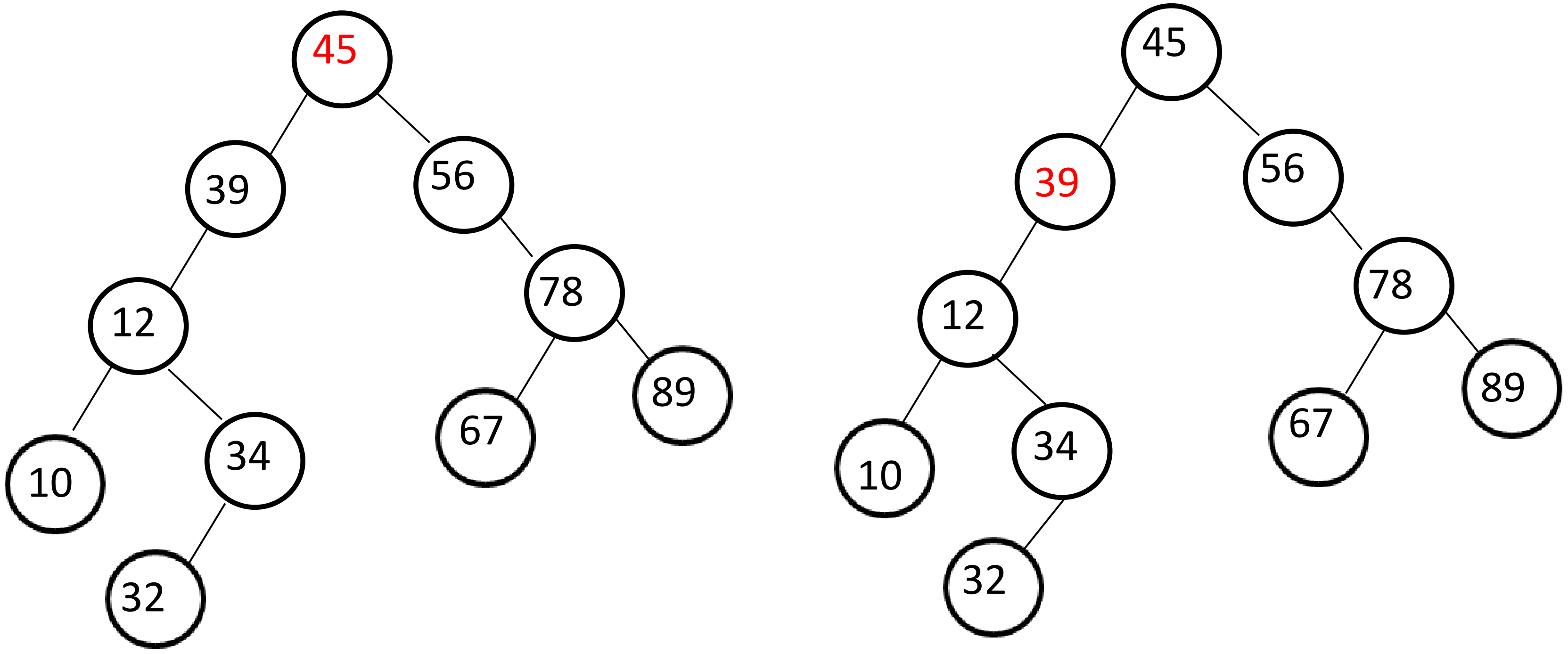


Fig23. Searching a node with the value 40 in the given binary search tree

Contd.

Algo: SearchElement(TREE, VAL)

```
Step1: 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]
Step2:  END.
```

Contd.

Inserting a New Node in a Binary Search Tree

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. 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 $O(\log_2 n)$ time to execute in the average case and $O(n)$ time in the worst case.

Contd.

Algo: Insert (TREE, VAL)

```
Step1: 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]
Step2: END.
```

Contd.

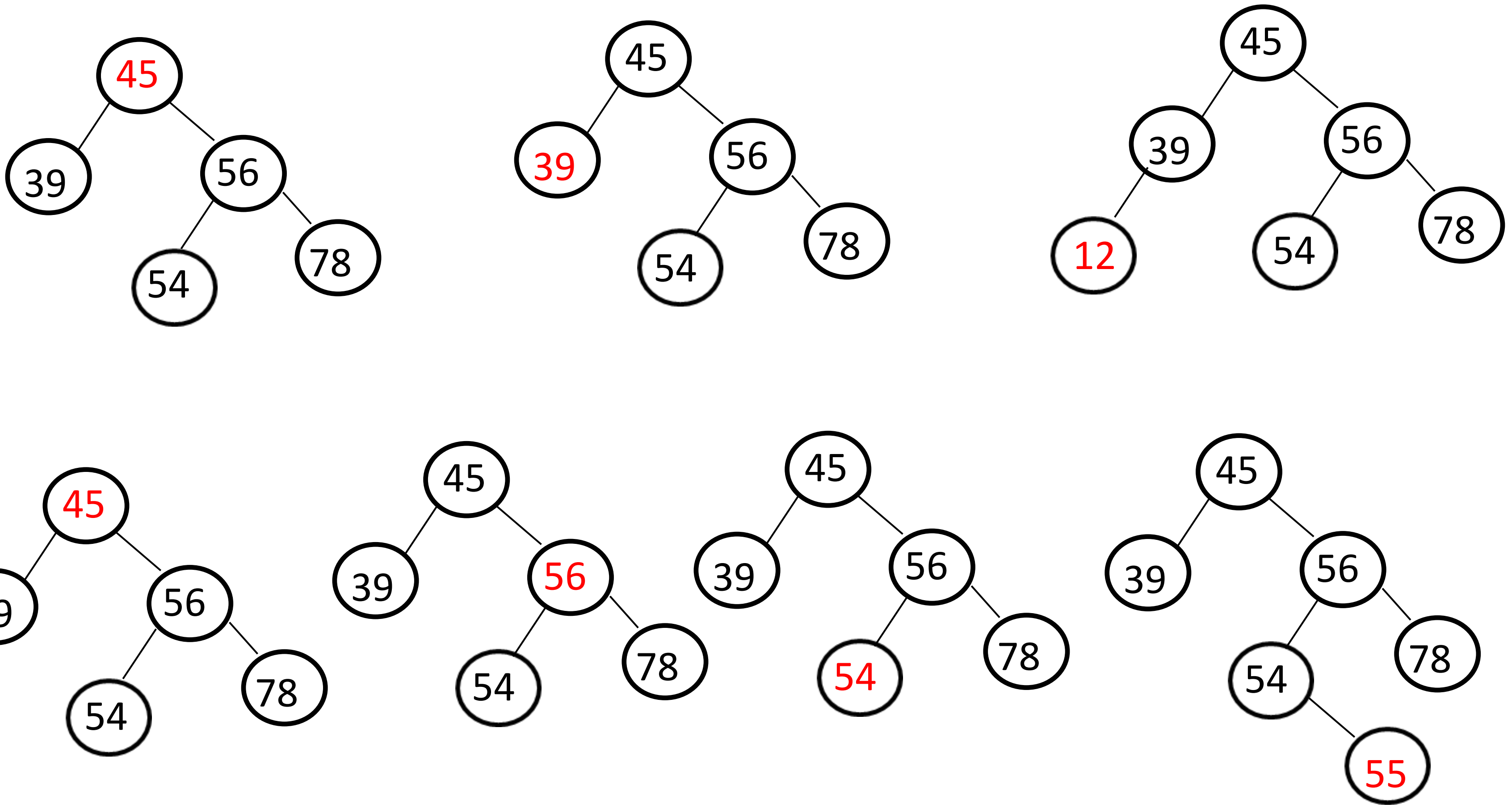


Fig24. Inserting nodes with values 12 and 55 in the given binary search tree

Contd.

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 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.

Contd.

Case 1: Deleting a Node that has No Children

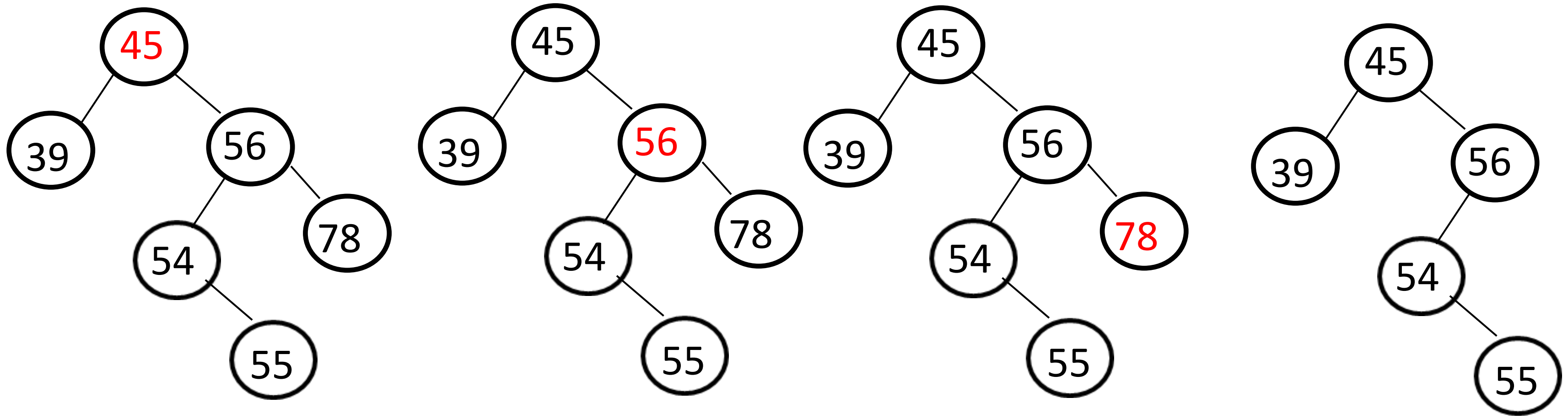


Fig25. Deleting node 78 from the given binary search tree

We can simply remove this node without any issue. This is the simplest case of deletion.

Contd.

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.

Contd.

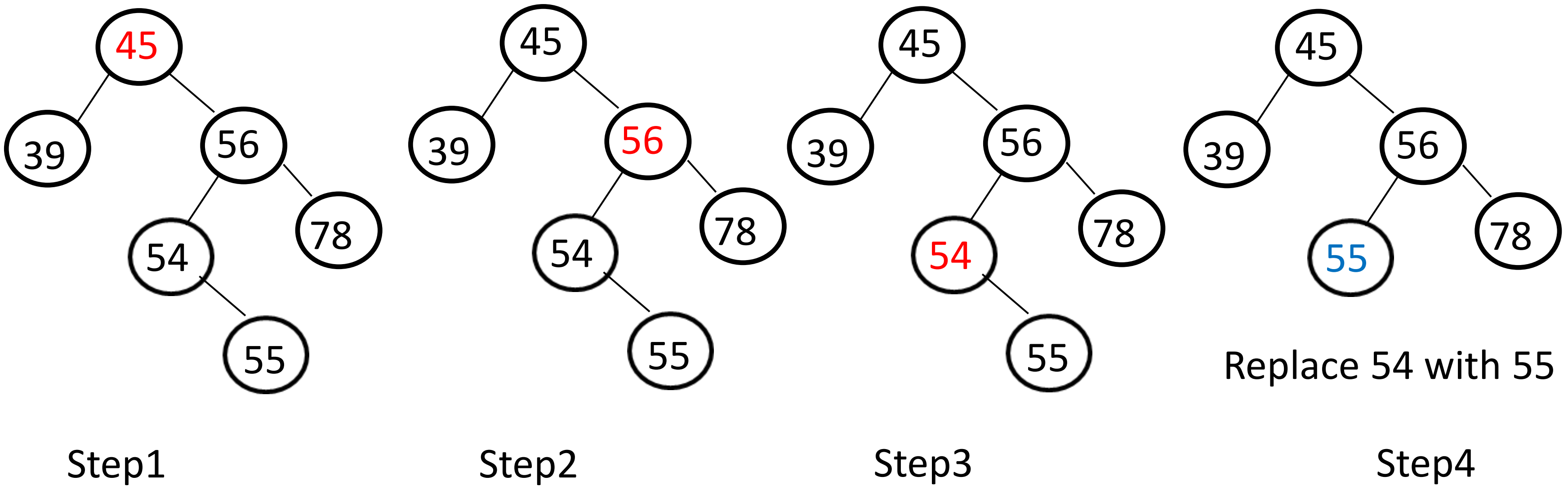


Fig26. Deleting node 54 from the given binary search tree

Contd.

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.

Contd.

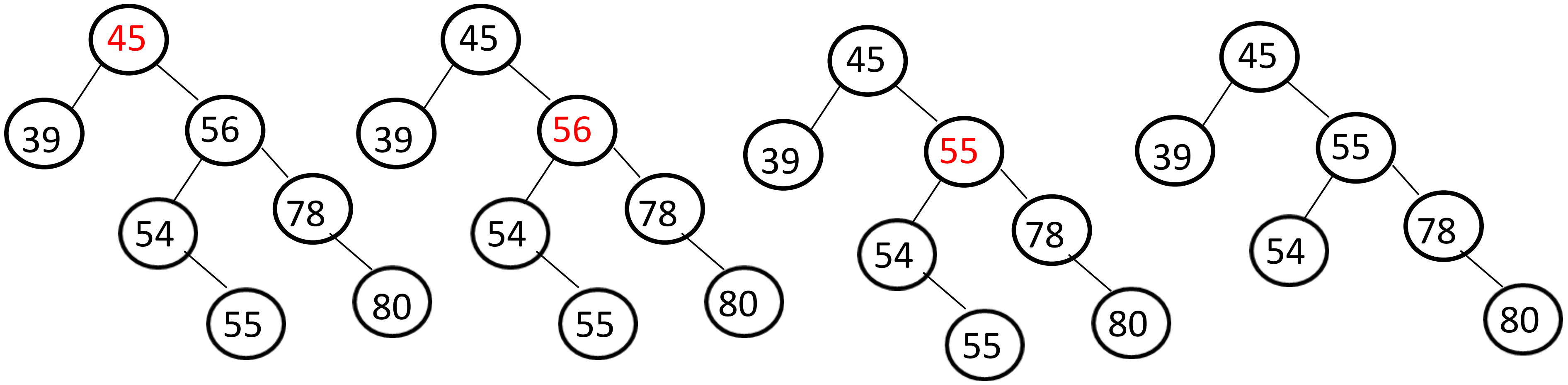


Fig.27. Deleting node 56 from the given binary search tree

This deletion could also be handled by replacing node 56 with its in-order successor.

Contd.

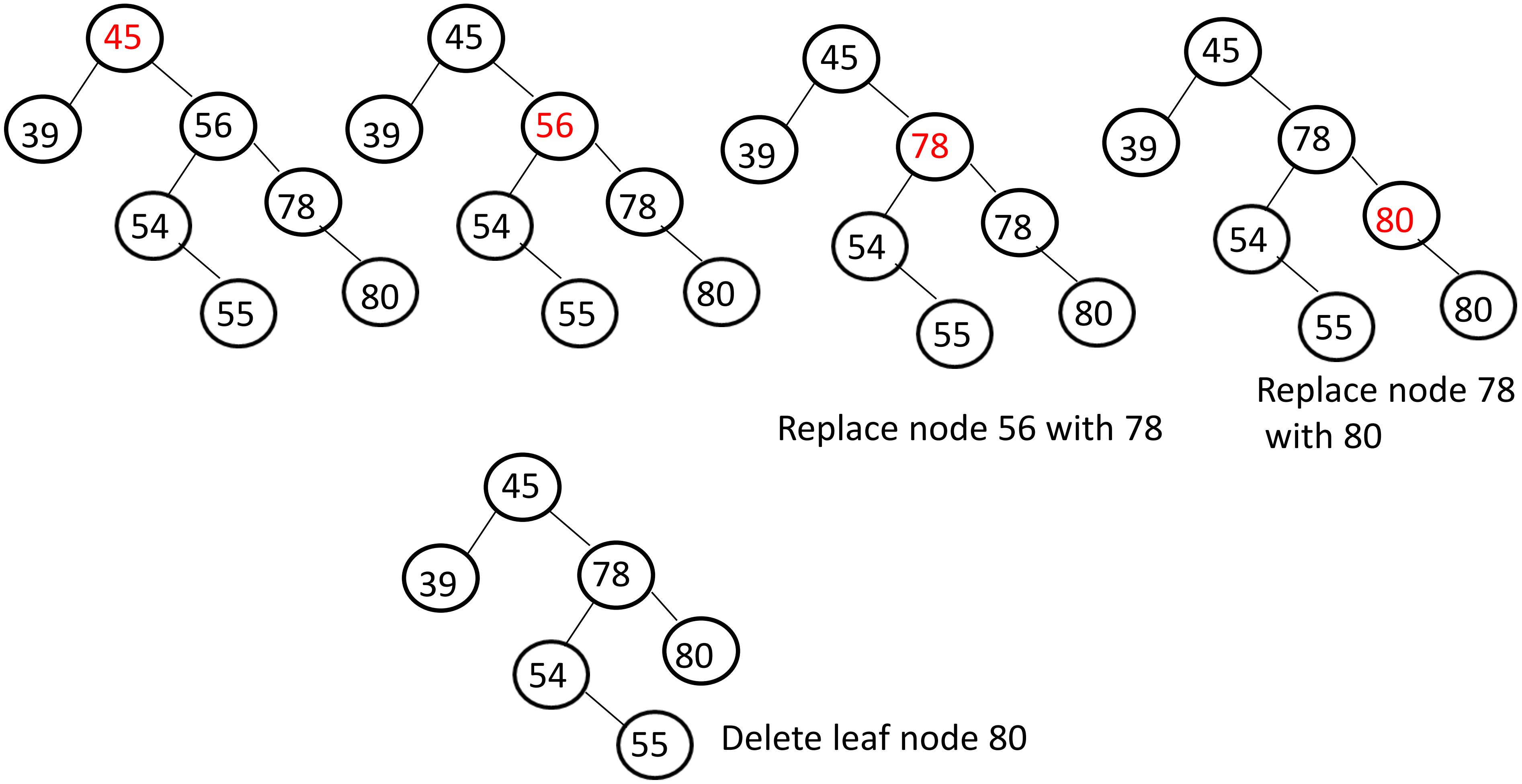


Fig.28. Deleting node 56 from the given binary search tree

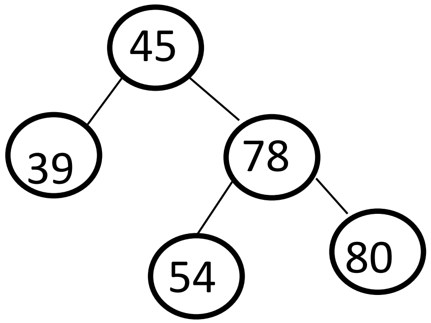
Algorithm to delete a node from a BST

Delete (TREE, VAL)

```
Step1: 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.
```

Determining the Height of a Binary Search Tree

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



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= 1 + 1 = 2

Contd.

Algorithm: Height (TREE)

```
Step1: 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]
Step2: END.
```

Determining the Number of Nodes

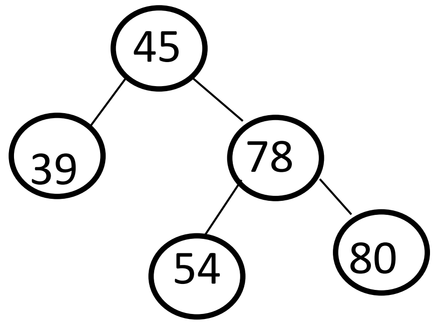
Determining the number of nodes in a binary search tree is similar to determining its height. To calculate the total number of elements/nodes in the tree, we count the number of nodes in the left sub-tree and the right sub-tree.

Number of nodes =

totalNodes(left subtree) + totalNodes(right subtree) + 1

Determining the Number of Nodes

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



Total nodes of left subtree = 1

Total nodes of right subtree = 3

Total nodes of tree = (1 + 3) + 1

= 5

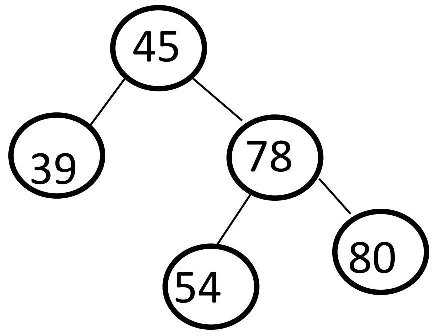
Contd.

Algorithm: TotalNodes(TREE)

```
Step1: IF TREE = NULL
              Return 0
        ELSE
              Return TotalNodes(TREE -> LEFT)  +
                      TotalNodes(TREE -> RIGHT) + 1
       [END OF IF]
Step2: END.
```

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).



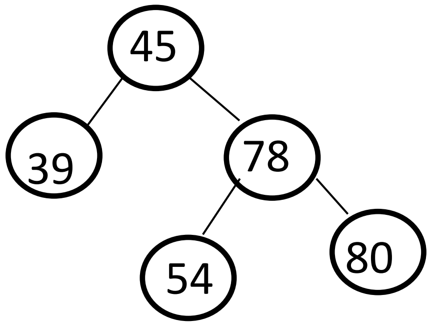
Number of internal nodes =

totalInternalNodes(left subtree) +

totalInternalNodes(right subtree) + 1

Contd.

Consider the tree shown below. The total number of internal nodes in the tree can be calculated as:



Total internal nodes of left subtree = 0

Total internal nodes of right subtree = 1

Total internal nodes of tree = (0 + 1) + 1 = 2

Contd.

Algorithm: TotalInternalNodes(TREE)

```
Step1: IF TREE = NULL
              Return 0
       [END OF IF]
       IF TREE -> LEFT = NULL AND TREE -> RIGHT= NULL
              Return 0
       ELSE
              Return TotalInternalNodes(TREE -> LEFT)
              + TotalInternalNodes(TREE -> RIGHT) + 1
       [END OF IF]
Step 2: END.
```

Determining the Number of External 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.

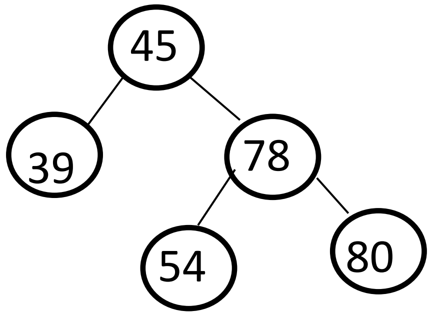
Number of external nodes =

TotalExternalNodes(left subtree) +

TotalExternalNodes(right subtree)

Contd.

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



Total external nodes of left subtree = 1

Total external nodes of right subtree = 2

Total internal nodes of tree = (1 + 2) = 3

Contd.

Algorithm: TotalExternalNodes(TREE)

```
Step1:  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]
Step2:  END.
```

Finding the Mirror Image of a Binary Search Tree

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

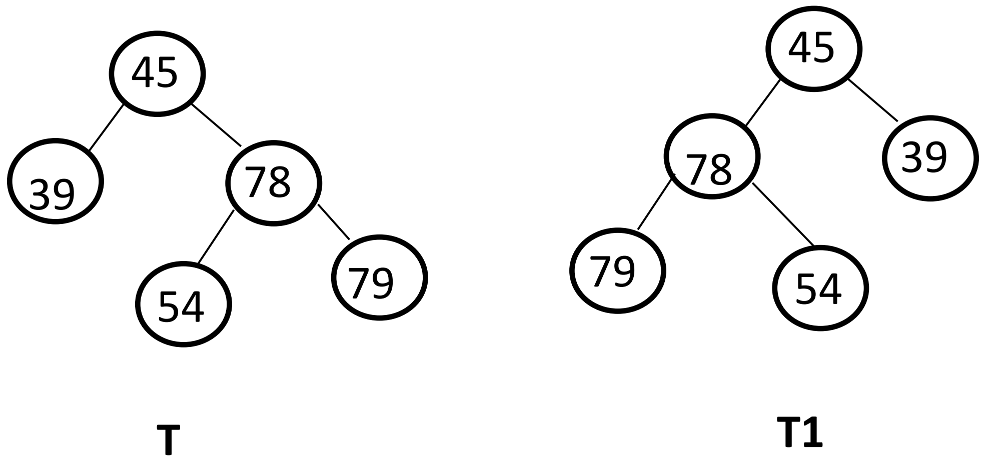


Fig29. BST T and it’s mirror image T1

Contd.

Algorithm: MirrorImage(TREE)

```
Step1: 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]
Step2: END.
```

Contd.

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.

Algorithm: DeleteTree(TREE)

```
Step1: IF TREE != NULL
              DeleteTree (TREE -> LEFT)
              DeleteTree (TREE -> RIGHT)
              Free (TREE)
       [END OF IF]
Step2: END.
```

Contd.

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.

Contd.

Algorithm: FindSmallestElement(TREE)

```
Step1: IF TREE = NULL OR TREE -> LEFT = NULL
                Returen TREE
        ELSE
                ReturnFindSmallestElement(TREE->LEFT)
        [END OF IF]
Step2: END.
```

Contd.

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.

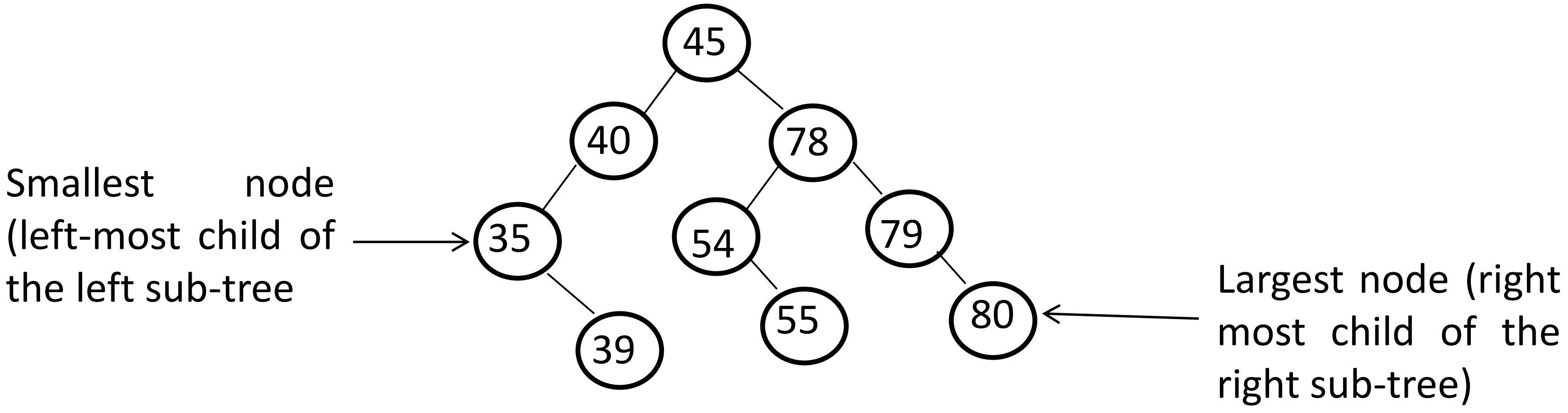

Smallest node (left-most child of the left sub-tree
45
40
78
35
54
79
39
55
80
Largest node (right most child of the right sub-tree)

Fig30. BST

Contd.

Algorithm: FindLargestElement(TREE)

```
Step1: IF TREE = NULL OR TREE -> RIGHT = NULL
                Return TREE
           ELSE
                Return FindLargestElement(TREE-> RIGHT)
          [END OF IF]
Step2: END.
```

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.

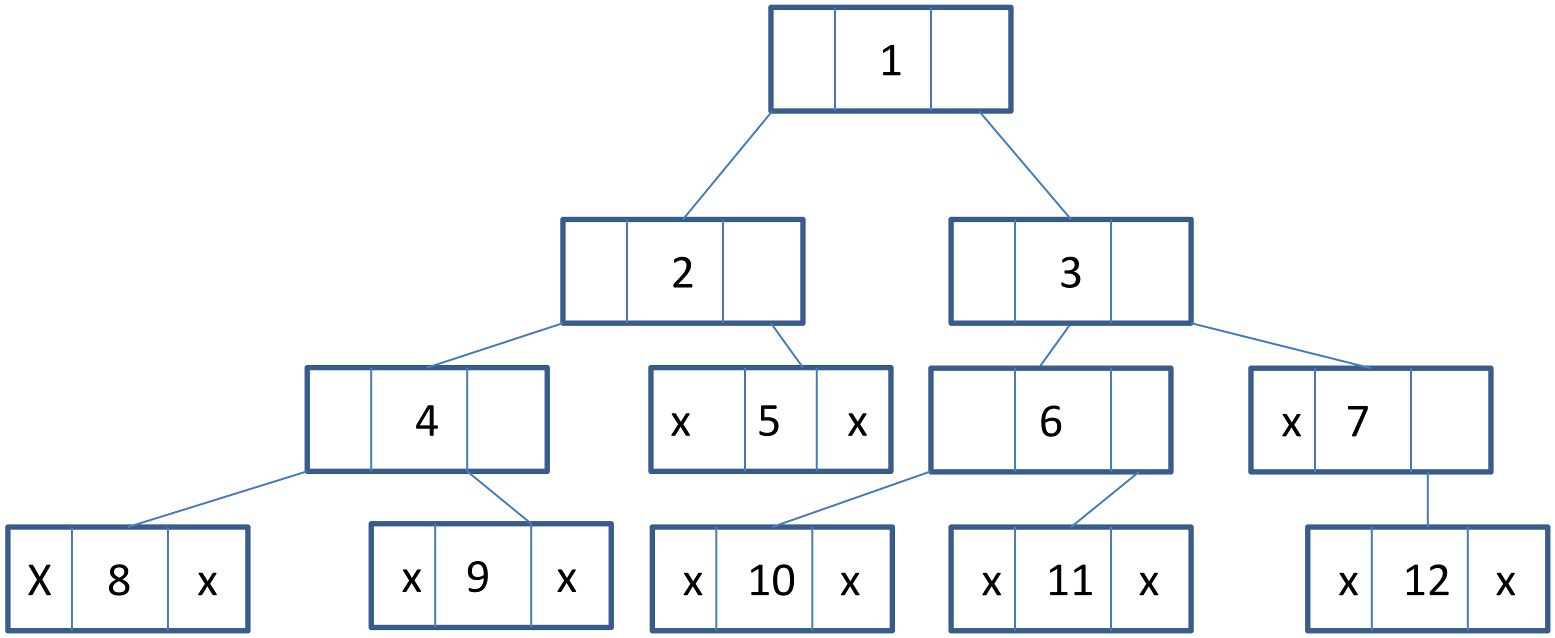


Fig31. Linked representation of a binary tree

Contd.

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. A threaded binary tree may correspond to one-way threading or a two-way threading.

Contd.

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

Contd.

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.

Contd.

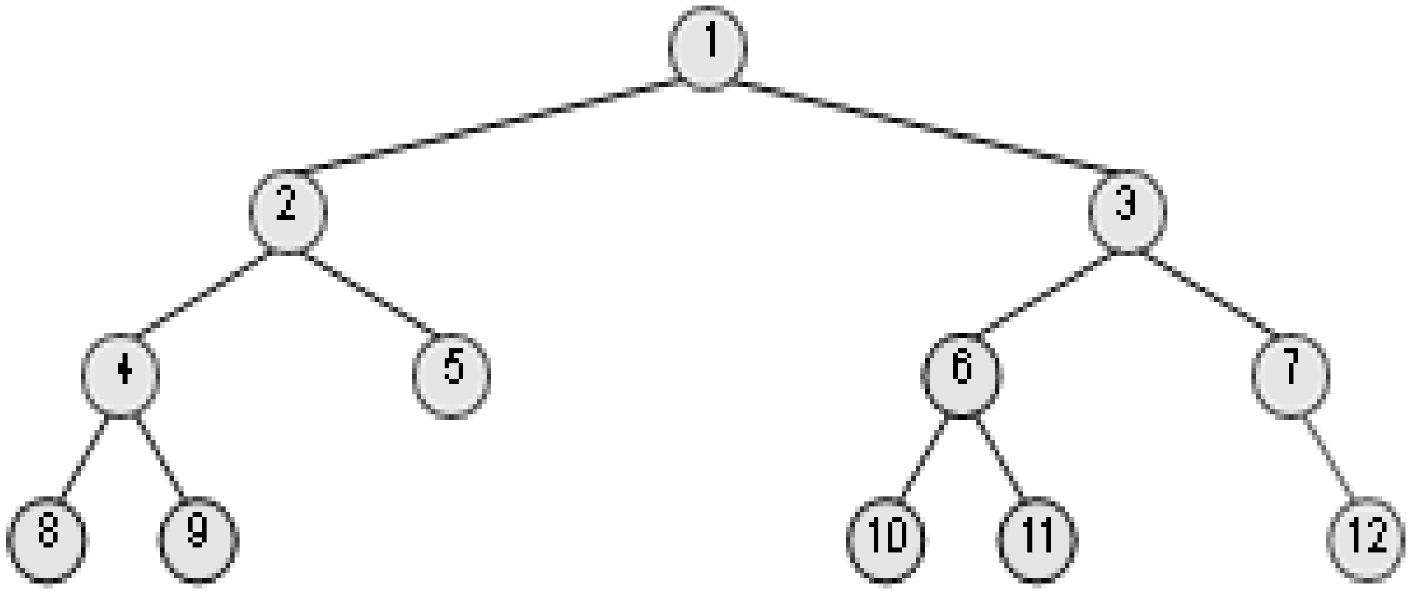


Fig32.a) Binary tree without threading

The in-order traversal of the tree is :
8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12

Contd.

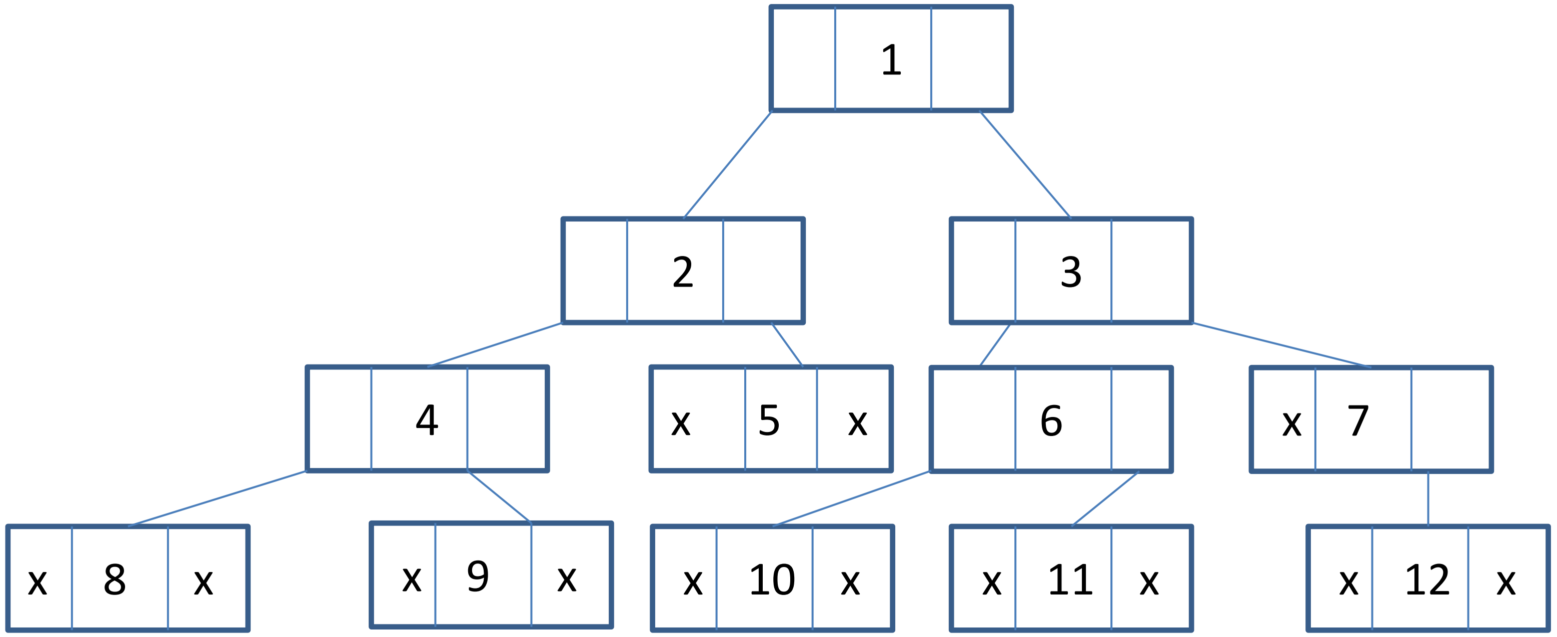


Fig32.b) Linked representation of the binary tree(without threading)

One-way Threading

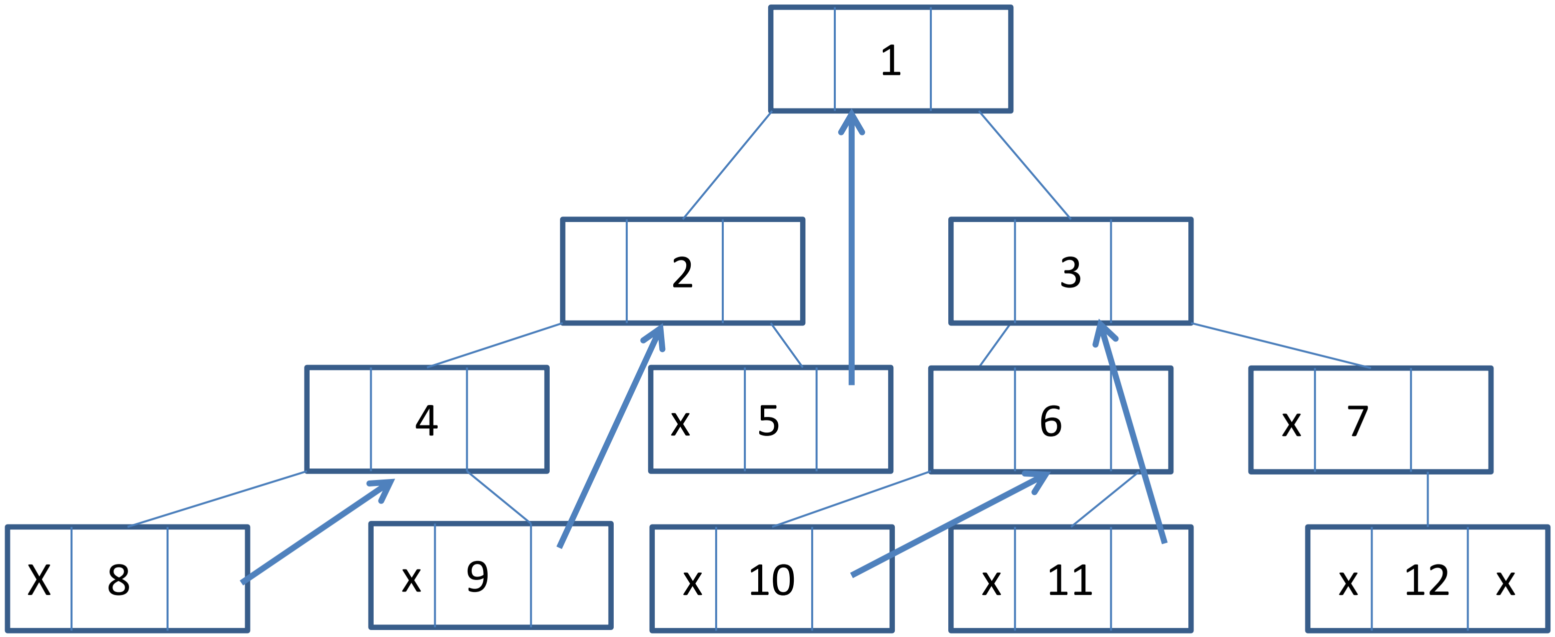


Fig33.a)) Linked representation of the binary tree with one-way threading

In order traversal: 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12

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 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.

Contd.

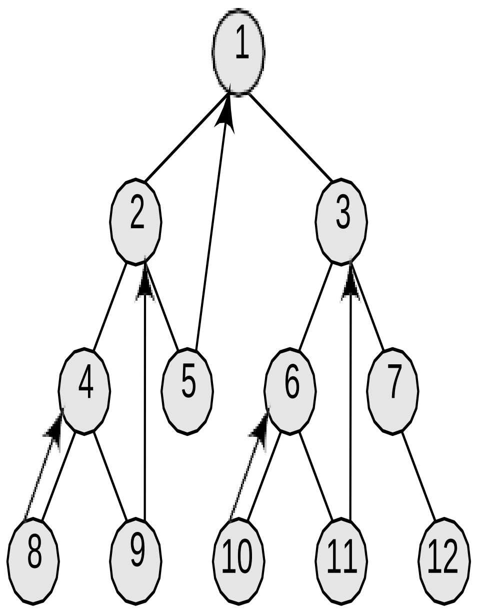
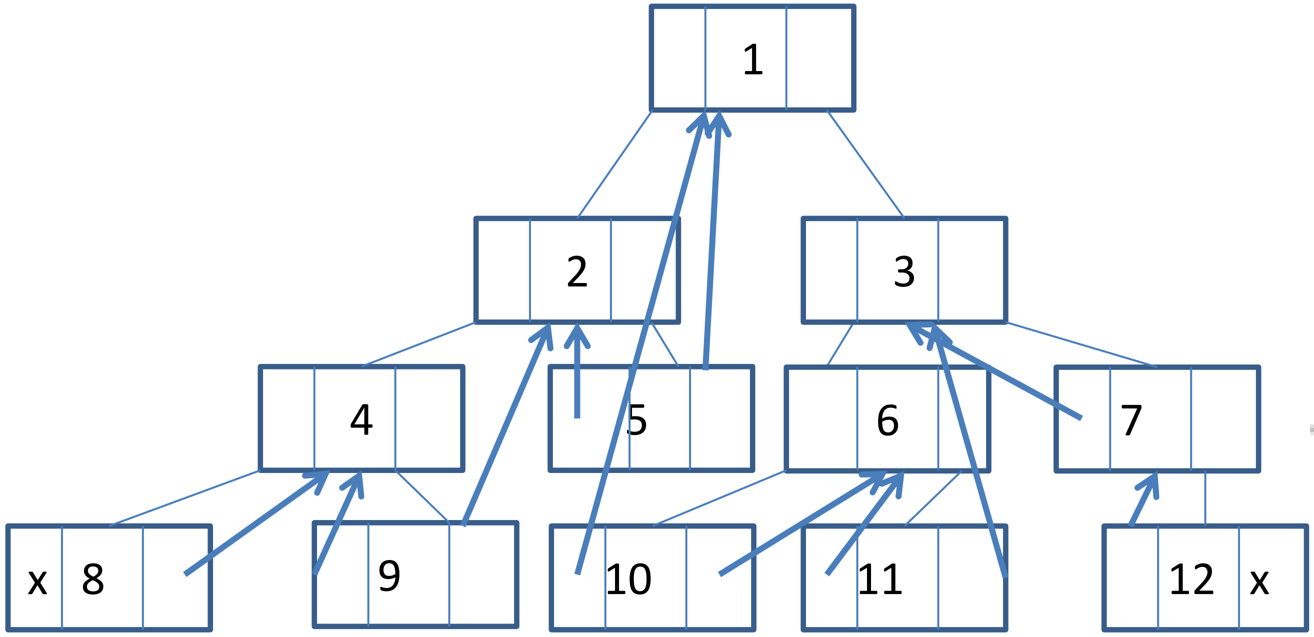


Fig33.b) Binary tree with one-way threading

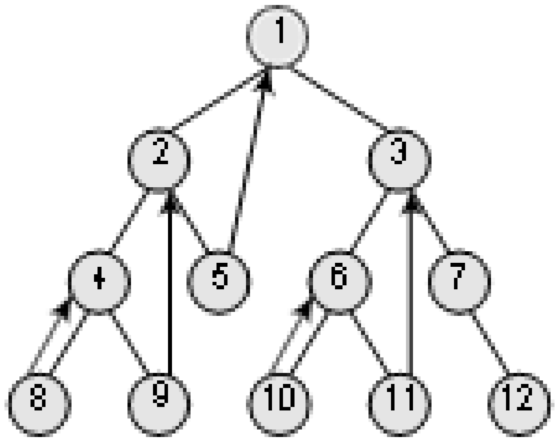
Contd.

Two-way Threading

In order traversal: 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12



a)

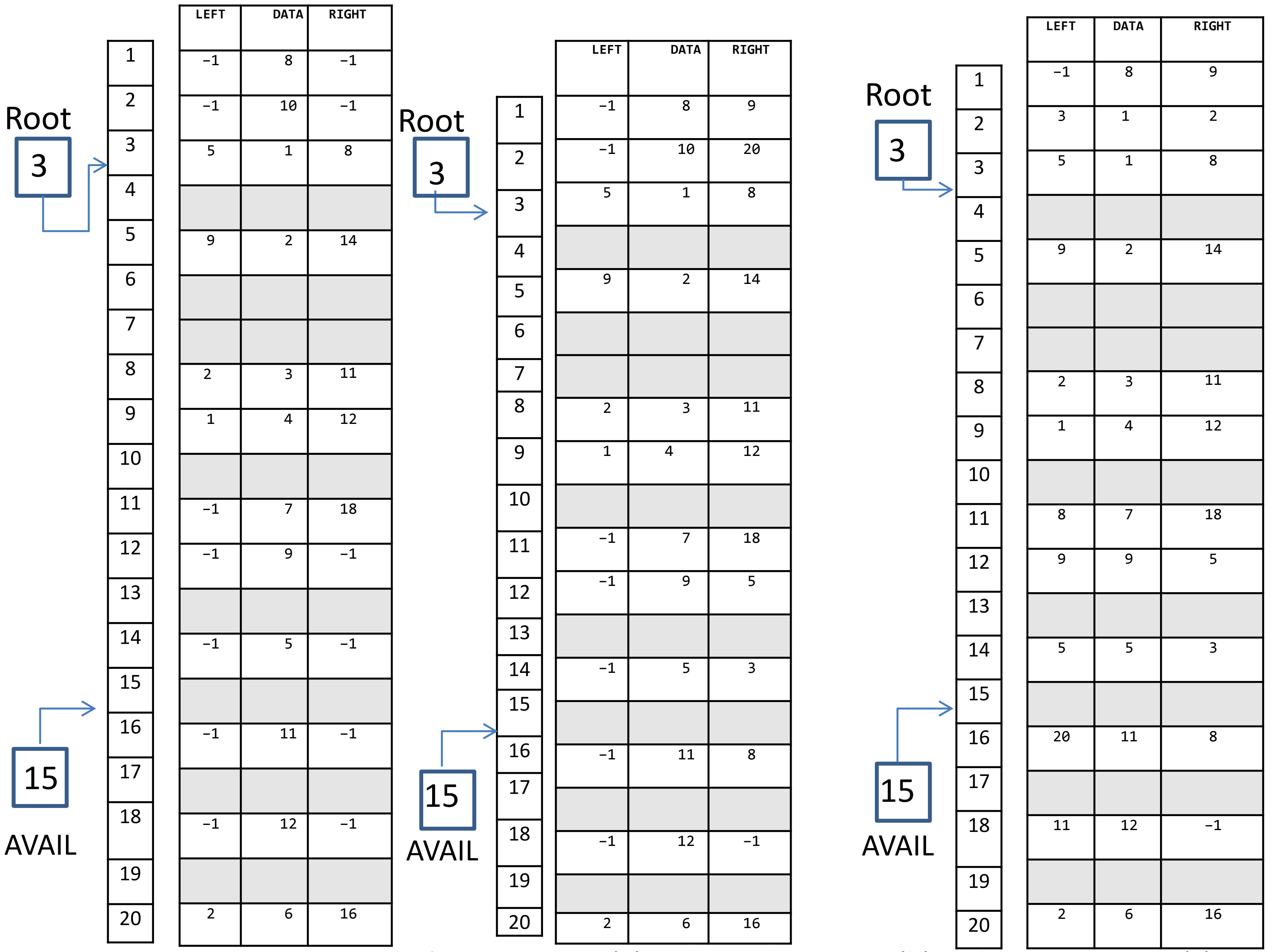


b)

Fig34.a) Linked representation of the binary tree with threading (b) binary tree with two-way threading

Contd.

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.


Root
3
AVAIL
15

(a)

	LEFT	DATA	RIGHT
1	−1	8	−1
2	−1	10	−1
3	5	1	8
4			
5	9	2	14
6			
7			
8	2	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

(b)

	LEFT	DATA	RIGHT
1	−1	8	9
2	−1	10	20
3	5	1	8
4			
5	9	2	14
6			
7			
8	2	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

(c)

	LEFT	DATA	RIGHT
1	−1	8	9
2	3	1	2
3	5	1	8
4			
5	9	2	14
6			
7			
8	2	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

Fig.34. Memory representation of binary trees: (a) without threading, (b) with one-way, and (c) two-way threading

Contd.

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.

Contd.

Algo: In-order traversal of a threaded binary tree

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.

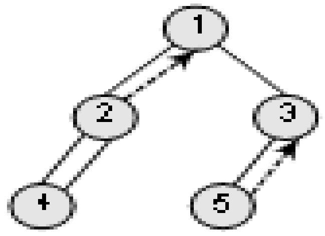
Step 3: If the current node has a right child, go to Step 4 else go to Step 5.

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.

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

Contd.

AVL TREES

AVL tree is a self balancing binary search tree invented by G.M.Adelson-Velsky and E.M.Landis in1962. 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 $O(\log_2 n)$ 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 $O(\log_2 n)$.

Contd.

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.

Balance factor =

Height (left subtree) – Height (right subtree)

Contd.

- 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*.

Contd.

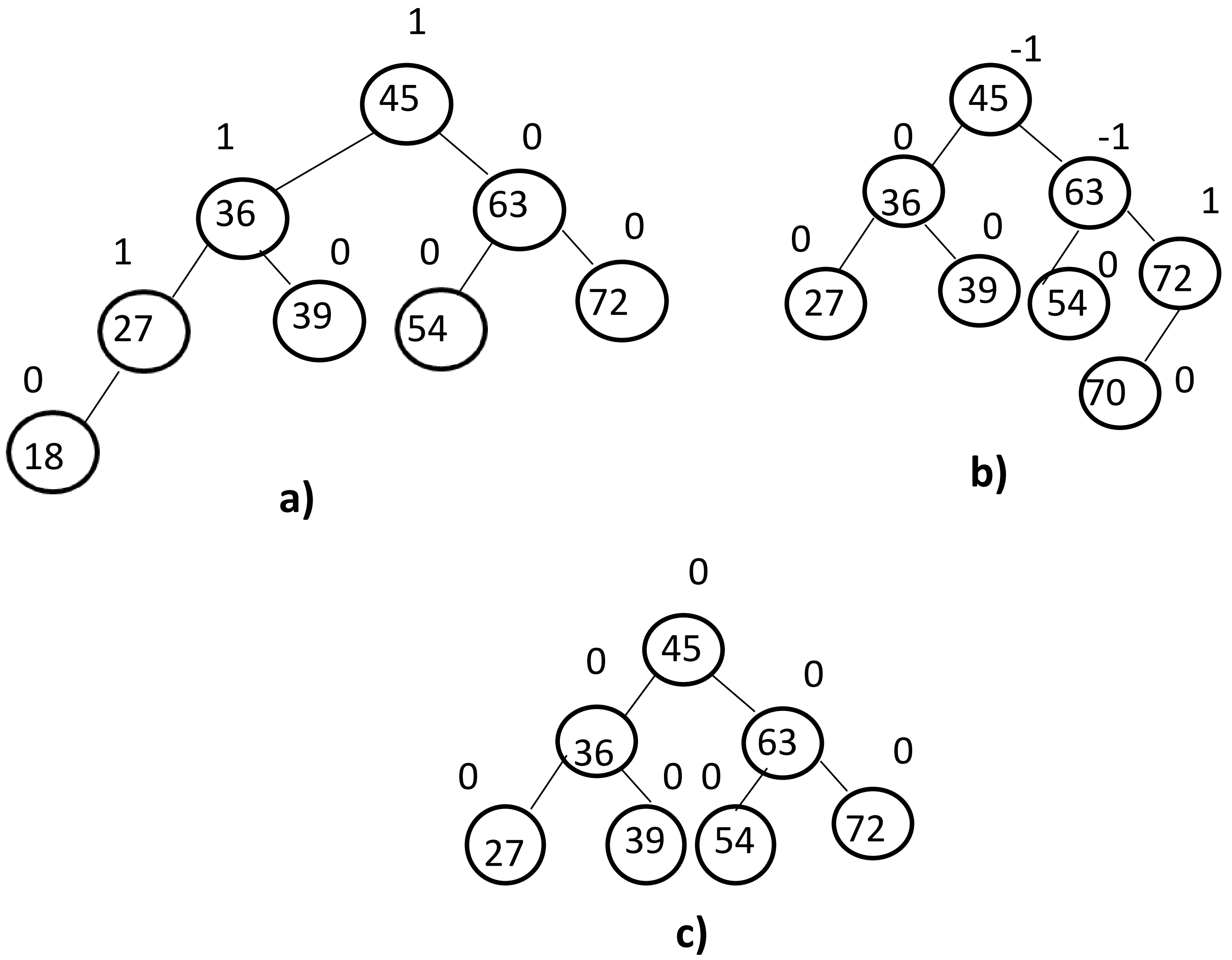


Fig. 35. a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

Contd.

Look at Fig.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, where as 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, where as 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).

Contd.

The trees given in Fig.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.

Contd.

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 $O(\log_2 n)$ time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

Contd.

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.

Contd.

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*

Contd.

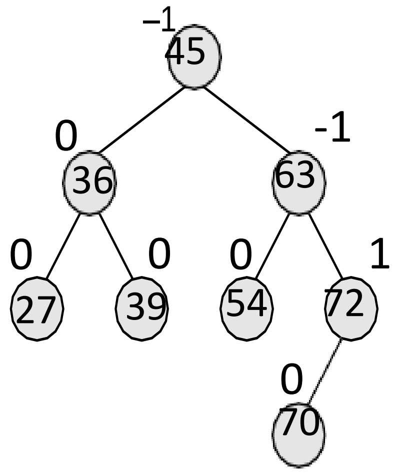


Fig.37. AVL Tree

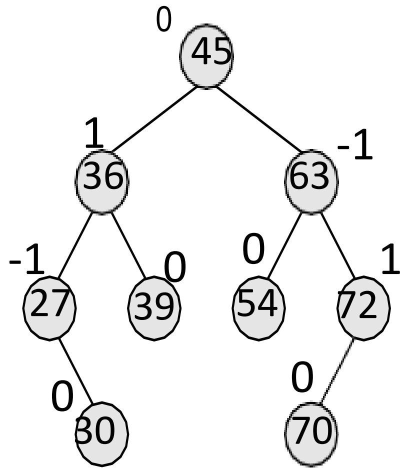


Fig.38. AVL Tree after inserting a node with the value 30

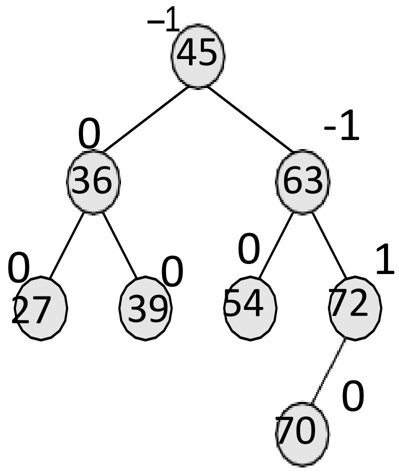


Fig.39. AVL Tree

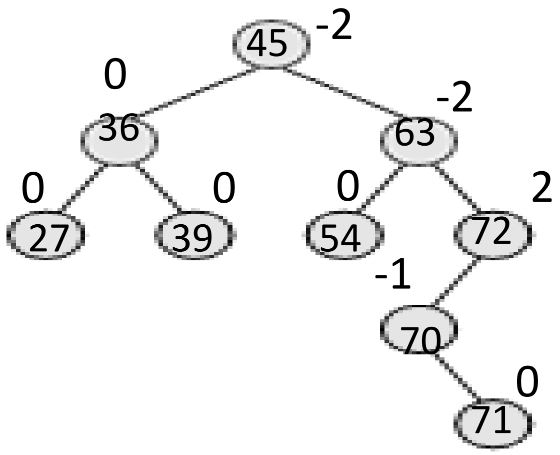


Fig.40. AVL tree after inserting a node with the value 71

Contd.

After inserting a new node with the value 71, the new tree will be as shown in Fig.40. Note that there are three nodes in the tree that have their balance factors 2, –2, and –2, thereby disturbing the AVL ness 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 Fig.40 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.

Contd.

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.

Contd.

LL Rotation

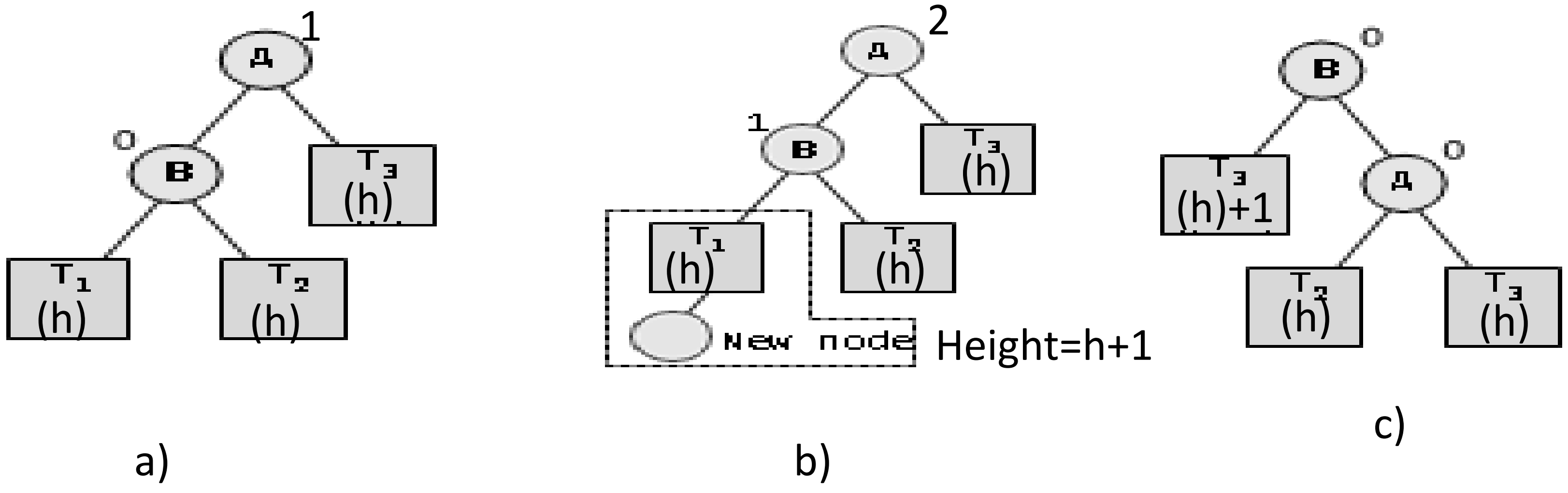


Fig.41. LL rotation in an AVL tree

Tree (a) is an AVL tree. In tree (b), a new node is inserted in the left sub-tree of the 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 Fig.(c).

1

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

Contd.

Consider the AVL tree given in the following Fig. and insert 18 into it

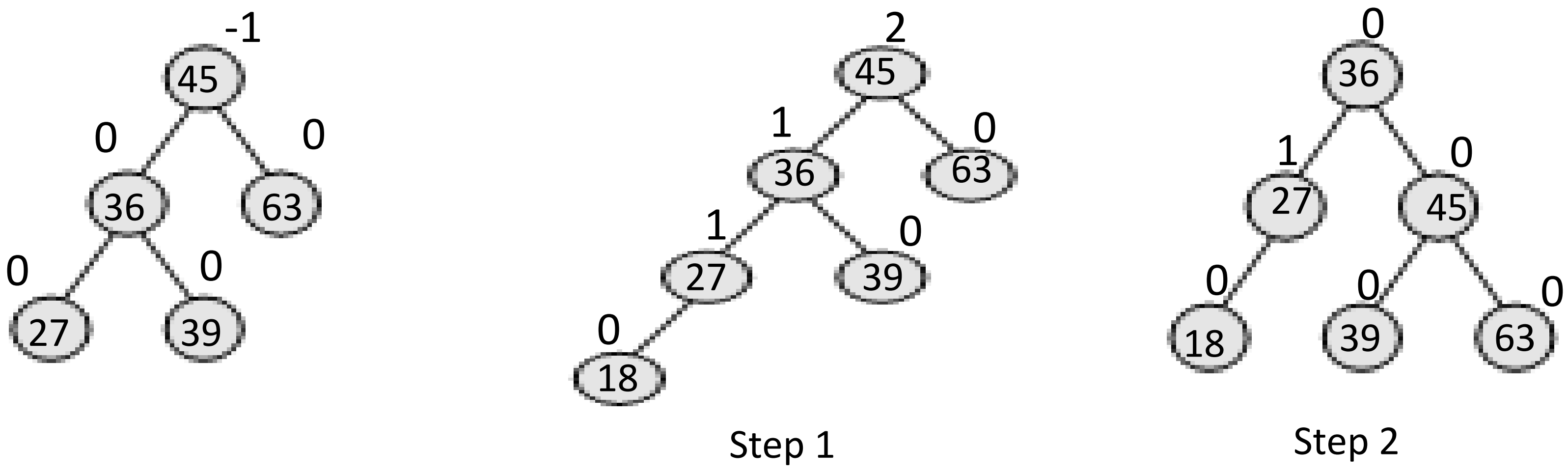


Fig.42 AVL Tree

Contd.

RR Rotation

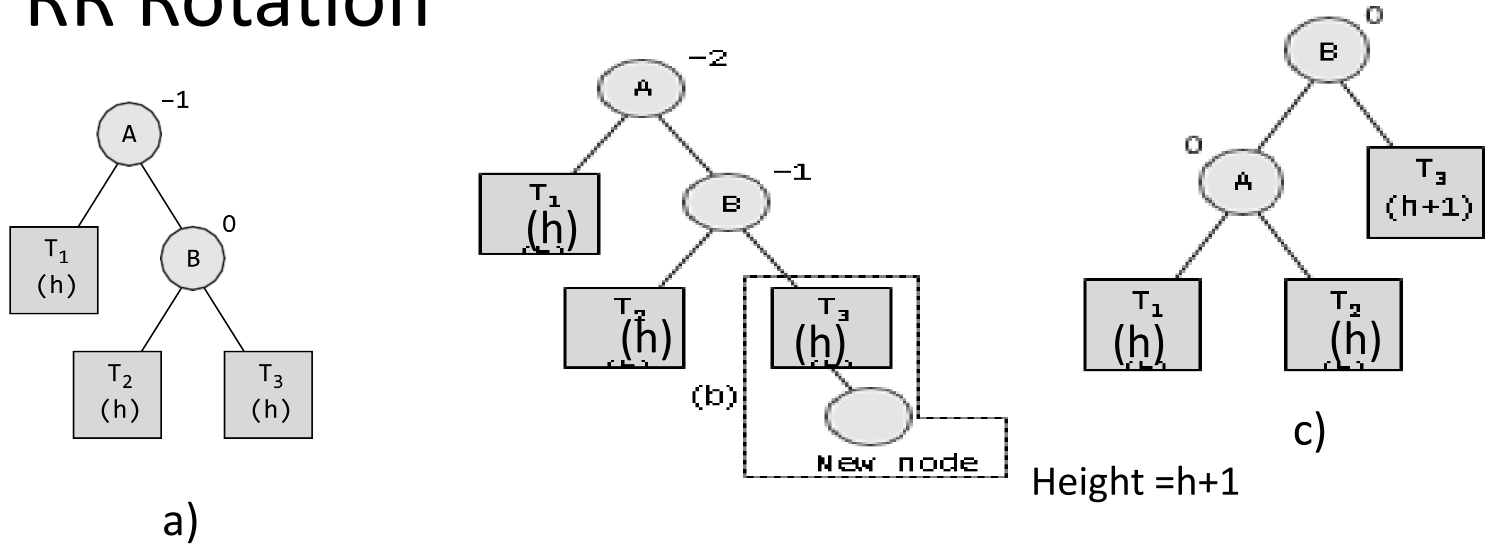


Fig.43. RR rotation in an AVL Tree

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 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 RR rotation as shown in tree (c). Note that the new node has now become a part of tree T3.

While rotation, node B becomes the root, with A and T3 as its left and right child. T1 and T2 become the left and right sub-trees of A.

Contd.

Consider the AVL tree given in the following Fig. and insert 89 into it

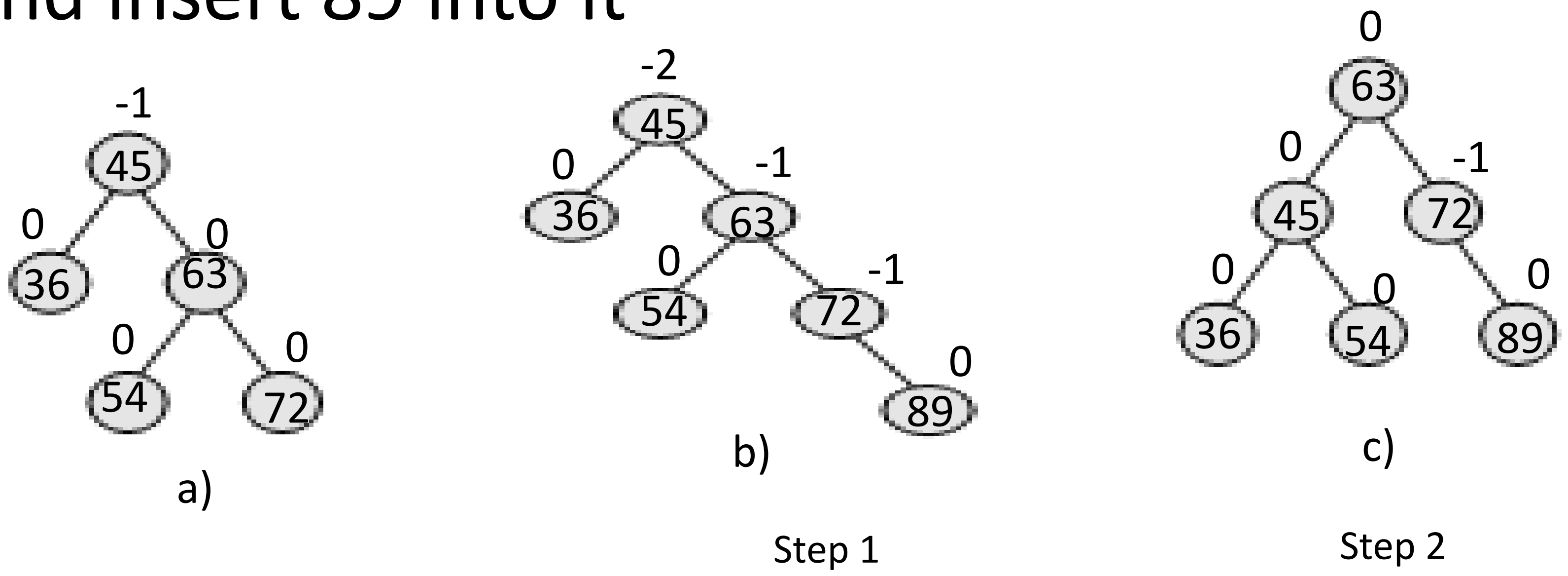


Fig.44 AVL Tree

Contd.

LR and RL Rotations

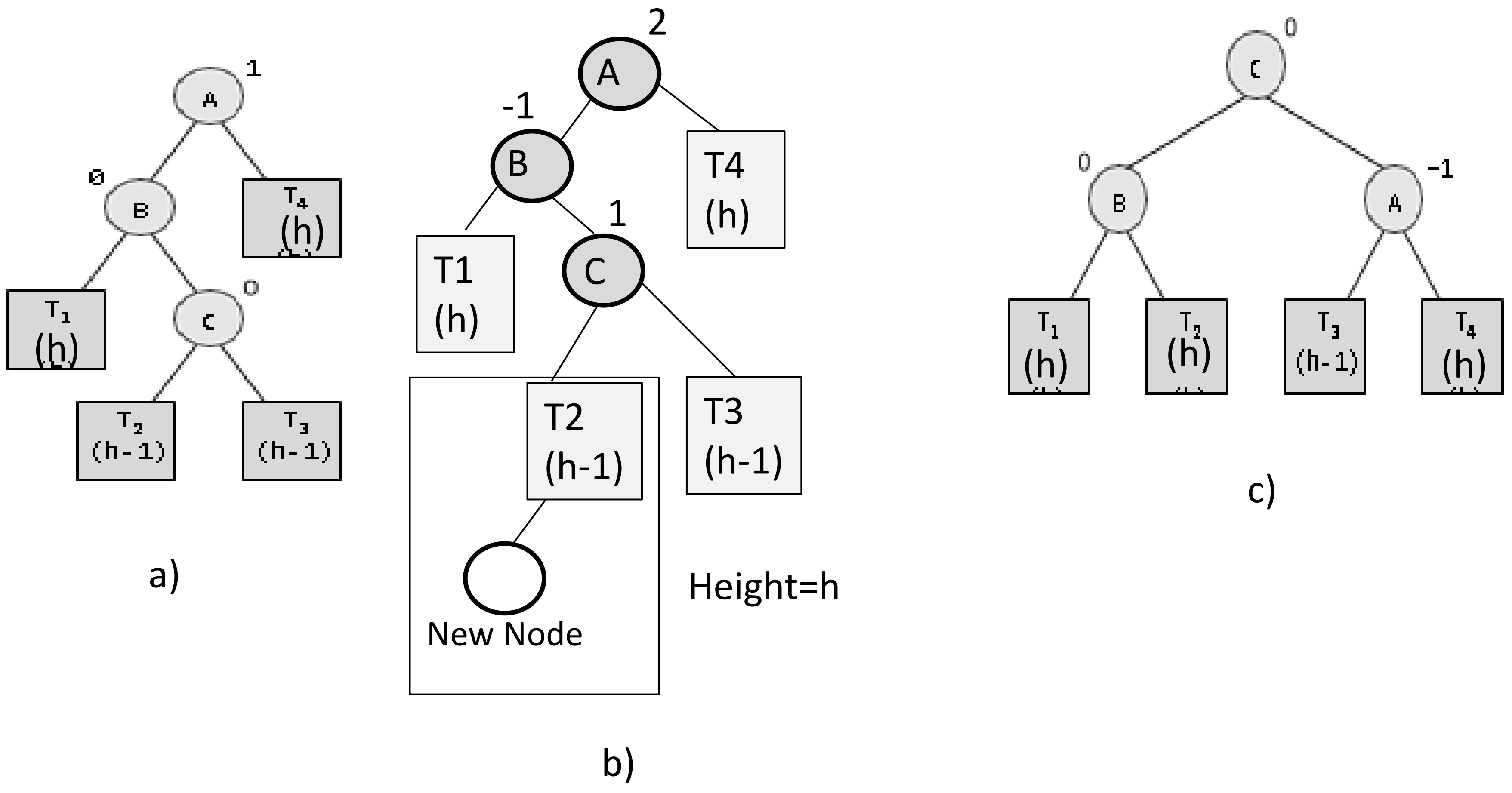


Fig.45. LR rotation in an AVL tree

Contd.

Tree (a) in previous Fig.45. 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 T2.

While rotation, node C becomes the root, with B and A as its left and right children. Node B has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node A.

Contd.

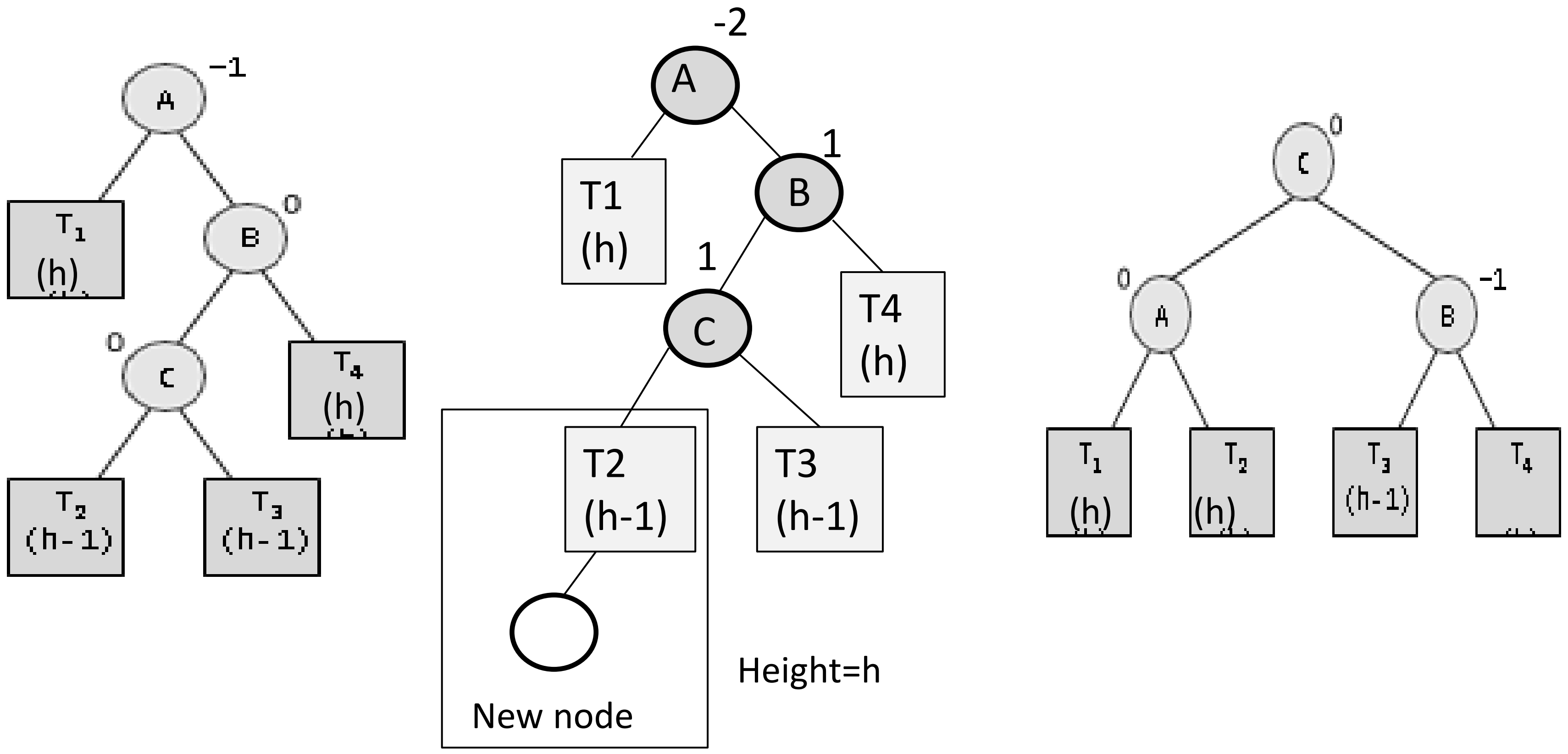


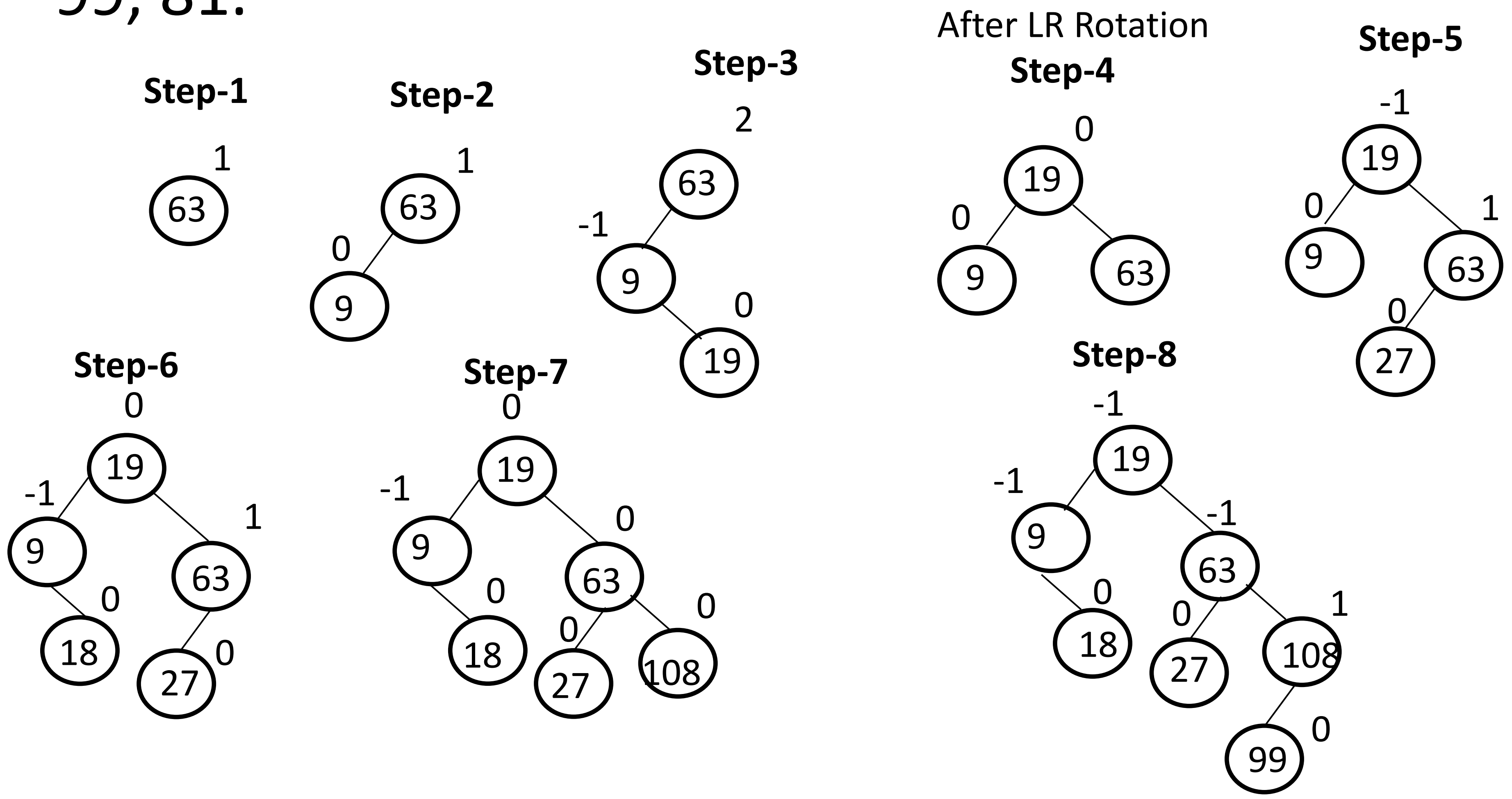
Fig.46. RL rotation in an AVL tree

Contd.

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 T2. While rotation, node C becomes the root, with A and B as its left and right children. Node A has T1 and T2 as its left and right sub-trees and T3 and T4 become the left and right sub-trees of node B.

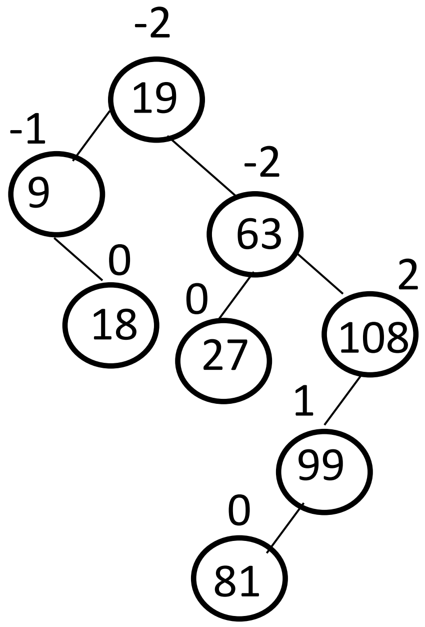
Contd.

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



Contd.

Step-9



After LL Rotation

Step-10

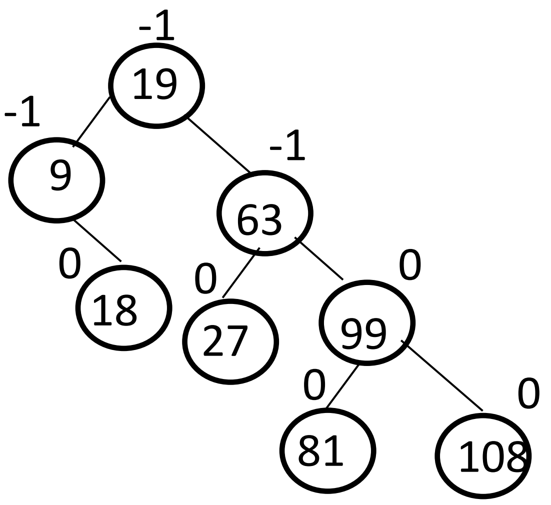


Fig.47. AVL Tree

Contd.

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.

Contd.

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. L rotations are the mirror images of R rotations.

Contd.

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.

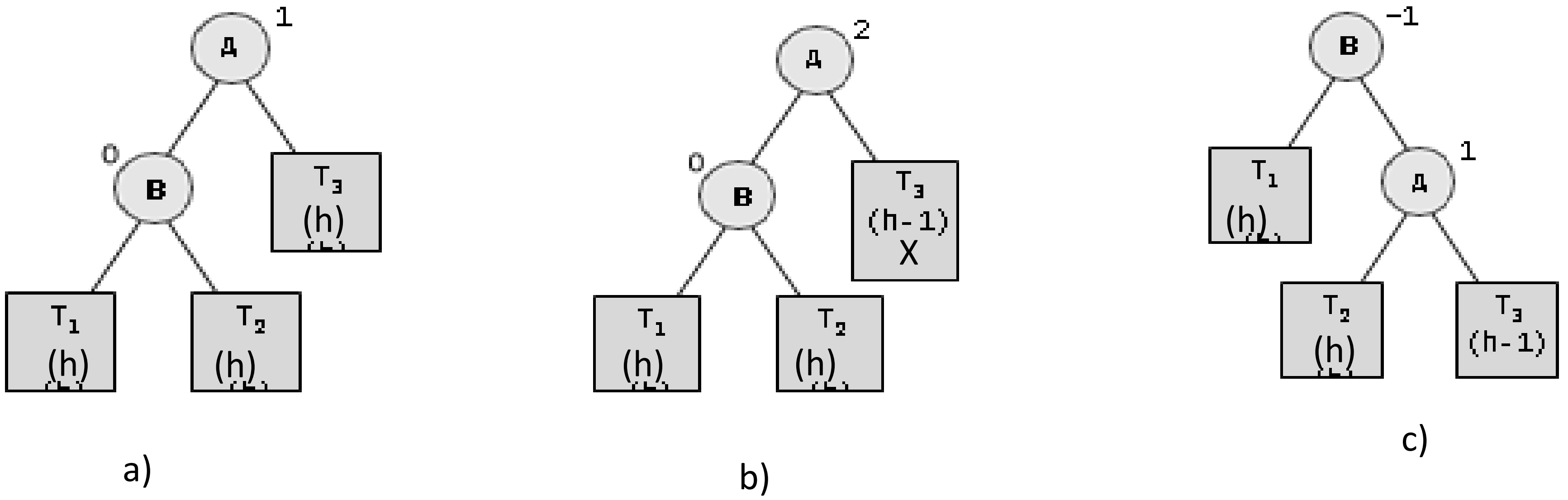


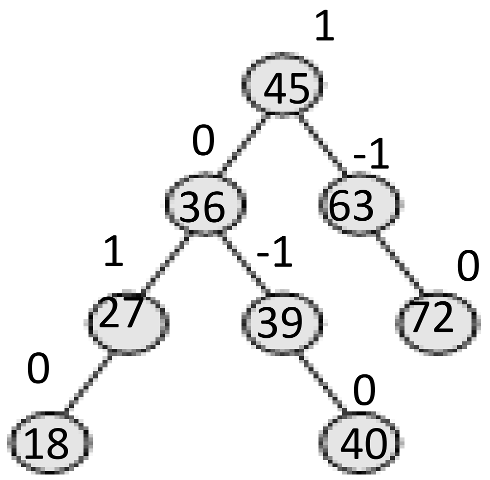
Fig.48. R0 rotation in an AVL tree

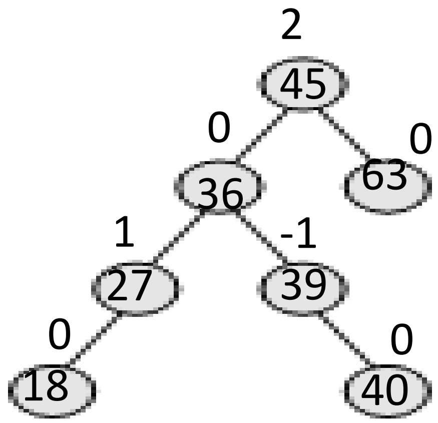
Contd.

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 T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

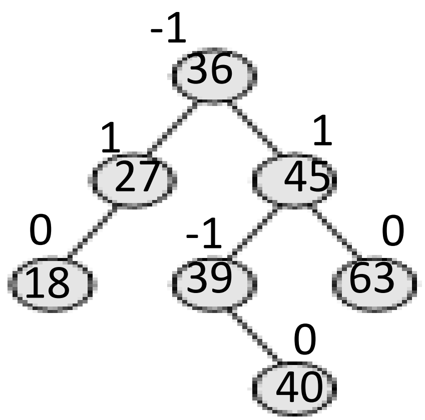
Contd.

Consider the AVL tree given in the following Fig. and delete 72 from it.





Step-1



Step-2

Fig.49. AVL Tree

Contd.

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.

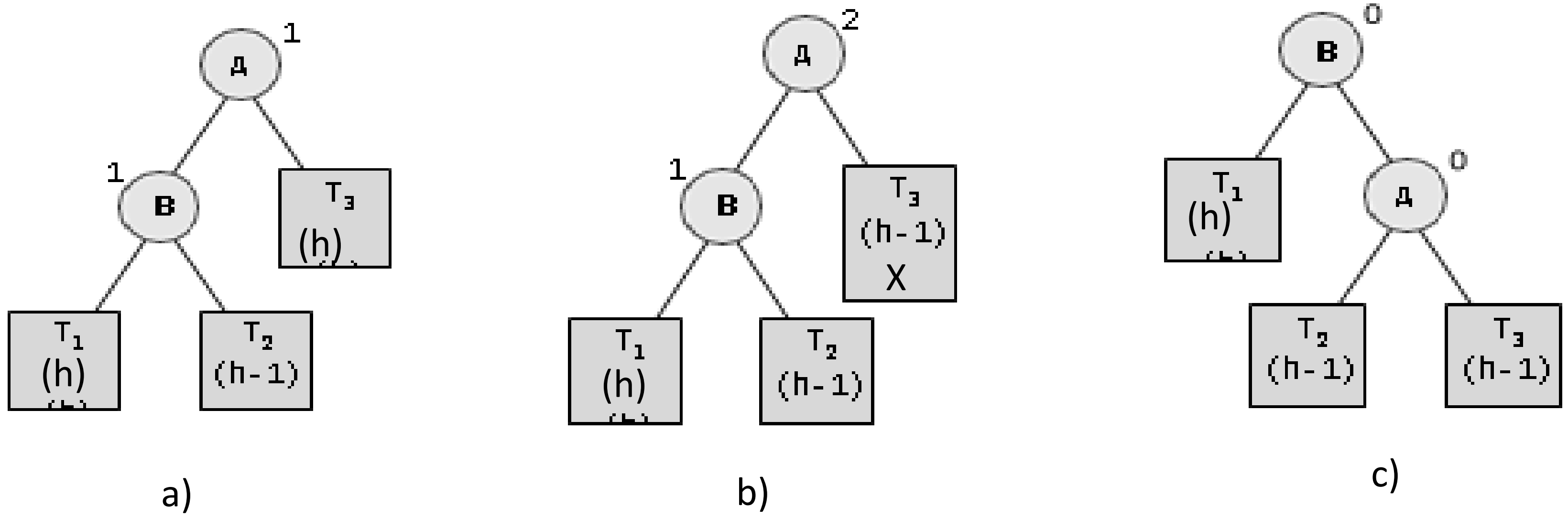


Fig.50. R1 rotation in an AVL tree

Contd.

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 T1 and A as its left and right children. T2 and T3 become the left and right sub-trees of A.

Contd.

Consider the AVL tree given in the following Fig. and delete 72 from it.

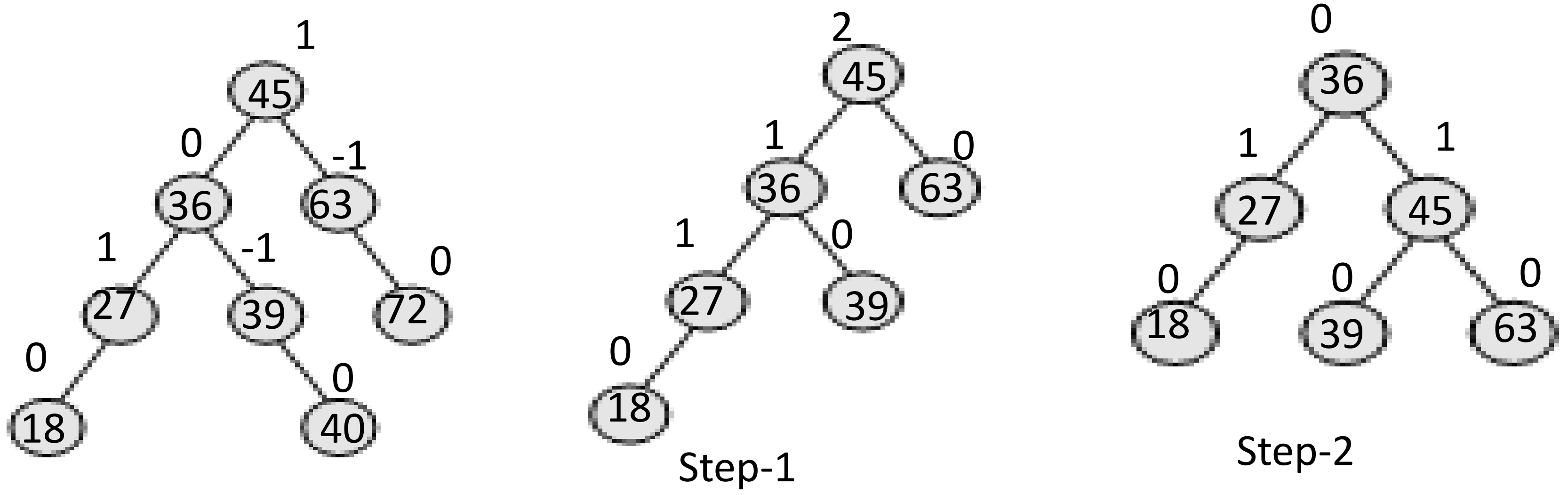
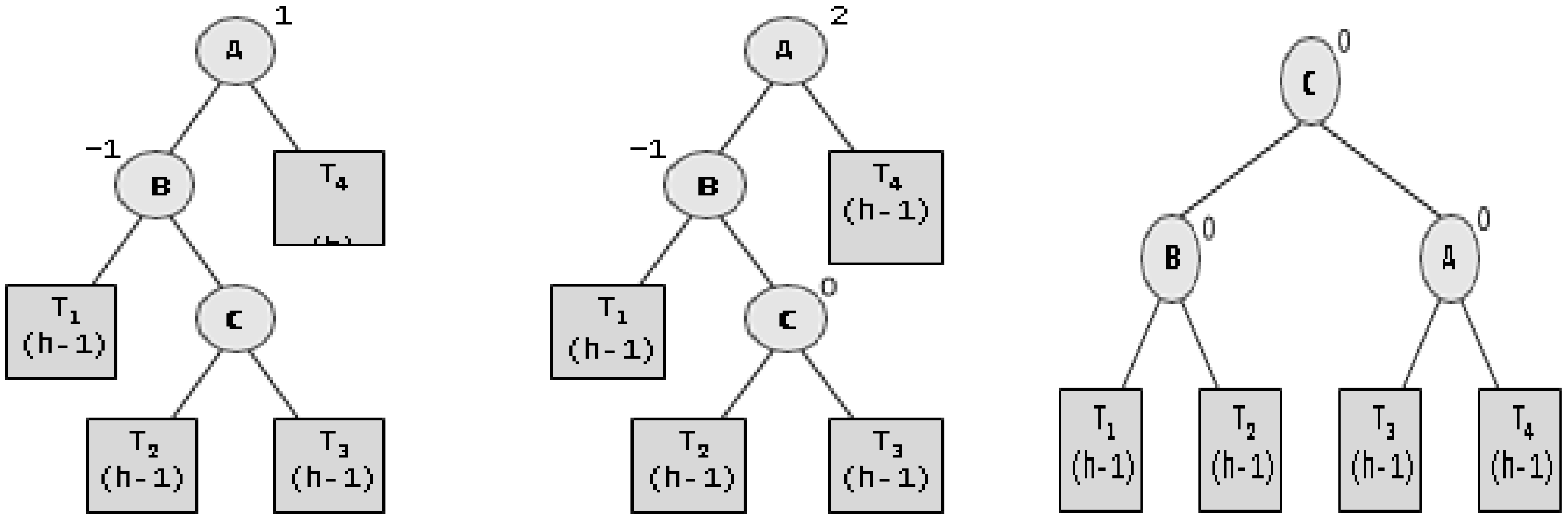


Fig.51. AVL Tree

Contd.

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.



a) b) c)

Fig52. R-1 Rotation in an AVL tree

Contd.

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 T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.