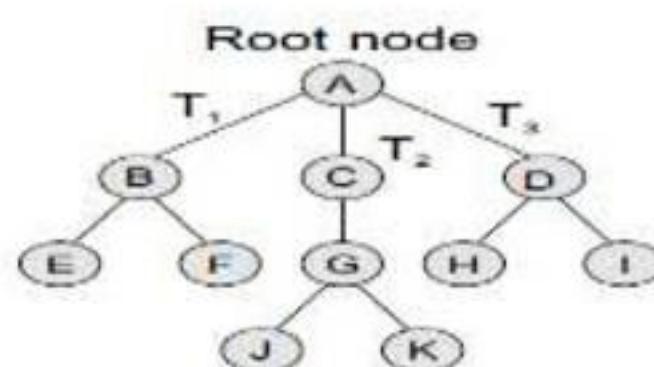


6 Trees

- Tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
 - Tree is a sequence of nodes.
 - There is a starting node known as root node.
 - Every node other than the root has a parent node.
 - Nodes may have any number of children.
- Figure 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.
- 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.
- **Definition (recursively):** A tree is a finite set of one or more nodes such that
 - There is a specially designated node called root.
 - The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree. T_1, \dots, T_n are called the subtrees of the root.



Characteristics of trees:

- Non-linear data structure
- Combines advantages of an ordered array
- searching as fast as in ordered array
- insertion and deletion as fast as in linked list

Application:

- Manipulate hierarchical data.
- Make information easy to search.
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms
- Form of a multi-stage decision-making (see business chess).

6.1 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 T_1, T_2, \dots, T_n are called the sub-trees of R.

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

Path: A sequence of consecutive edges is called a path. For example, in above figure the path from the root node A to node I is given as: A, D, and I. There is a single unique path from the root to any node.

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 above figure 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 above figure nodes C, G, J, and K are the descendants of node A.

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

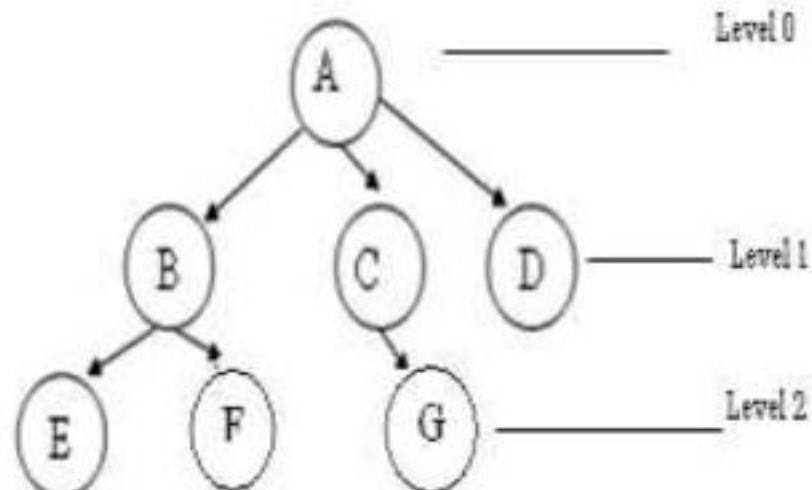
Degree of a Tree: The degree of a tree is the maximum degree of nodes in a given tree.

Height of a node: The height of a node is the maximum path length from that node to a leaf node. A leaf node has a height of 0.

Height of a tree: The height of a tree is the height of the root.

Depth of a node: Depth of a node is the path length from the root to that node. The root node has a depth of 0.

Depth of a tree: Depth of a tree is the maximum level of any leaf in the tree. This is equal to the longest path from the root to any leaf.



- ✓ A is the root node
- ✓ B is the parent of E and F
- ✓ D is the sibling of B and C
- ✓ E and F are children of B
- ✓ E, F, G, D are external nodes or leaves
- ✓ A, B, C are internal nodes
- ✓ Depth of F is 2
- ✓ the height of tree is 2
- ✓ the degree of node A is 3
- ✓ The degree of tree is 3

6.2 Binary Trees

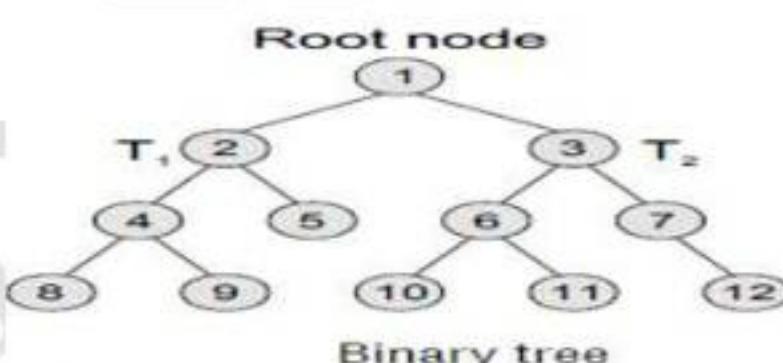
➤ The simplest form of tree is a binary tree. A binary tree consists of

- a node (called the root node) and
- left and right sub-trees. Both the sub-trees are themselves binary trees.

A binary tree is a hierarchical (tree) data structure in which each node has at most two children. Typically the child nodes are called left and right. One common use of binary trees is binary search trees; another is binary heaps.

➤ **Definition:**

A binary tree is a finite set of elements that is either empty or it consists of a node called root and two other disjoint subsets. And these two subsets are themselves binary trees called left sub tree and the right sub tree. Each element of a binary tree is called a node of the tree.



- Figure shows a binary tree. In the figure, R is the root node and the two trees T₁ 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.
- Note that 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.

6.3 Types of binary tree

- Strictly binary tree
- Complete binary tree
- Almost complete binary tree

6.3.1 Strictly binary tree

➤ A tree is said to be strictly binary tree if every non-leaf node has non empty left and right sub trees.

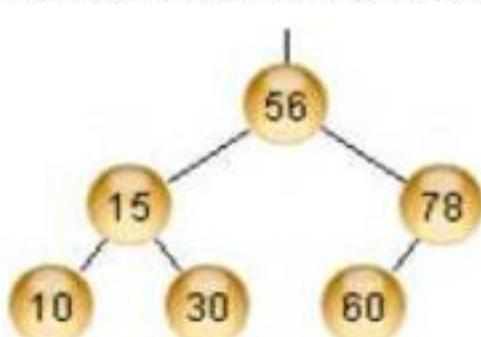


Figure 1

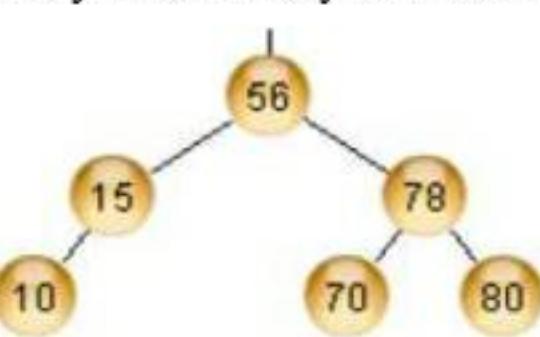


Figure 2

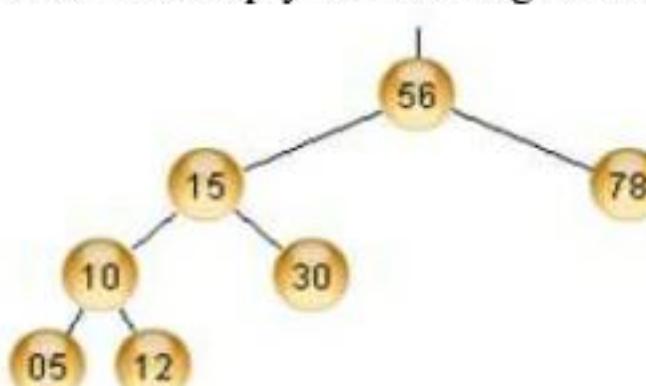


Figure 3

In the above diagram, **figure 1** is not a strictly binary tree because node **78** does not contain a right sub tree though it contains left sub tree. Similarly, **figure 2** is not a strictly binary tree because node **15** does not have right sub tree. But **figure 3** is a strictly binary tree because each non leaf node has right and left sub trees.

6.3.2 Complete binary tree

- A complete binary tree of depth d is the strictly binary tree whose leaves are at level d .

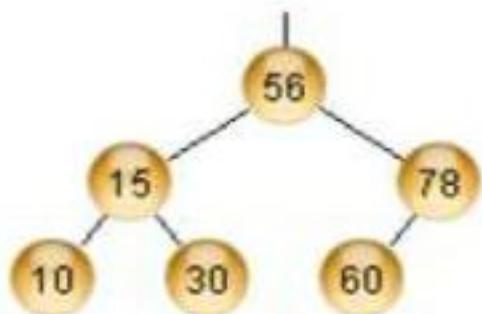


Figure 1

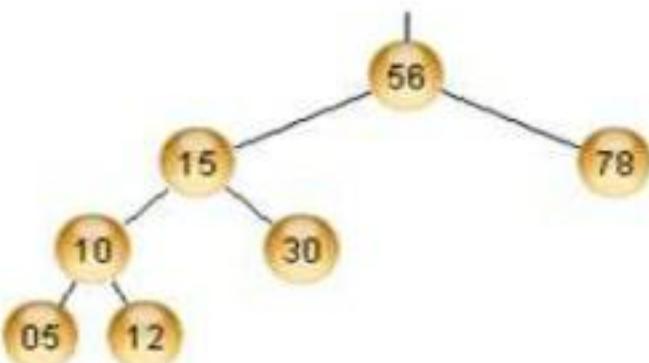


Figure 2

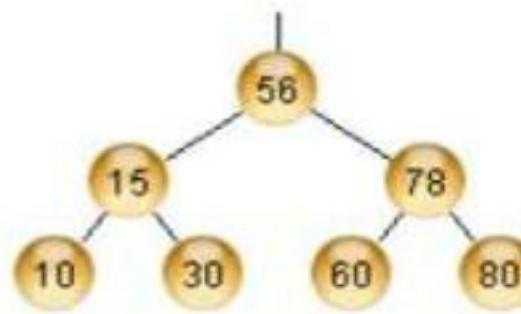


Figure 3

In the above diagram, **figure 1** is not a complete binary tree because it is not a strictly binary tree. **Figure 2** is a strictly binary tree but still it is not a complete binary tree because every leaf is not at the same level. Leaf nodes **78** and **30** from **figure 2** are not at the depth level of a binary tree. **Figure 3** is a complete binary tree because it is strictly binary tree and every leaf node is at level d . So, “every complete binary tree is a strictly binary tree but every strictly binary tree is not a complete binary tree”. If d is the depth of a complete binary tree, what is the total number of nodes there?

6.3.3 Almost complete binary tree

- A binary tree of depth d is an almost complete binary tree if:
 - The tree is complete Binary tree (All nodes) till level $d-1$.
 - At level d (i.e. the last level) if a node is present, then all the nodes to the left of that node should also be present.

OR

- For any node n in the tree with a right descendant at level d , n must have a left son and every left descendant of n is either a leaf at level d or has two sons.
- Any node n at level less than $d-1$ has two sons.

Explanation of point 1:

If any non-leaf node has right descendant that it must have left descendant. If right descendant goes to level d (depth) that each left descendant must have two sons or is a leaf at level d .

Binary trees	Strictly Binary Tree	Complete Binary Tree	Almost Complete Binary Tree
	No	No	No
	No	No	Yes
	Yes	No	No
	Yes	No	Yes
	Yes	Yes	Yes

Note:

If a binary tree has only left sub trees, then it is called left skewed binary tree. Figure 1 is a left skewed binary tree. If a binary tree has only right sub trees, then it is called right skewed binary tree. Figure 2 is a right skewed binary tree.

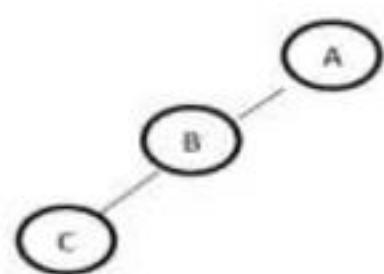


Figure 1

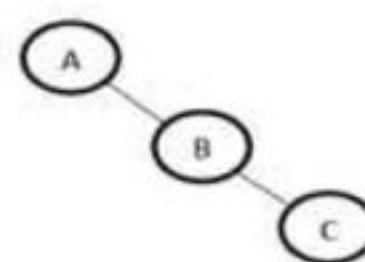


Figure 2

6.4 BINARY TREE REPRESENTATION

- There are two ways of representing binary tree T in memory:
 - Sequential representation using arrays
 - Linked list representation

6.4.1 ARRAY REPRESENTATION

- An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth d. Then at most $2^{d+1}-1$ nodes can be there in T. (i.e., SIZE = $2^{d+1}-1$). Consider a binary tree in Figure 1 of depth 2. Then SIZE = $2^{2+1}-1 = 7$.

Then the array A [7] is declared to hold the nodes.

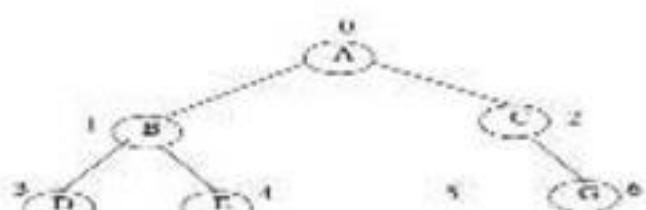


Figure 1: Binary Tree of Depth 2

A[]	A	B	C	D	E	F
	[0]	[1]	[2]	[3]	[4]	[5]

Figure 2: Array Representation of Binary Tree of Figure 1

To perform any operation on Binary tree we have to identify the father, the left child and right child of node.

1. The father of a node having index n can be obtained by $(n - 1)/2$. For example to find the father of D, where array index n = 3. Then the father nodes index can be obtained = $(n - 1)/2$

$$\begin{aligned}
 &= 3 - 1/2 \\
 &= 2/2 \\
 &= 1
 \end{aligned}$$
i.e., A[1] is the father D, which is B.
2. The left child of a node having index n can be obtained by $(2n+1)$. For example to find the left child of C, where array index n = 2. Then it can be obtained by
$$\begin{aligned}
 &= (2n + 1) \\
 &= 2*2 + 1 \\
 &= 4 + 1 \\
 &= 5
 \end{aligned}$$
i.e., A[5] is the left child of C, which is NULL. So no left child for C.
3. The right child of a node having array index n can be obtained by the formula $(2n+ 2)$. For example to find the right child of B, where the array index n = 1. Then = $(2n + 2)$

$$\begin{aligned}
 &= 2*1 + 2 \\
 &= 4
 \end{aligned}$$
i.e., A[4] is the right child of B, which is E.
4. If the left child is at array index n, then its right brother is at $(n + 1)$. Similarly, if the right child is at index n, then its left brother is at $(n - 1)$.

The array representation is more ideal for the complete binary tree. The Fig8 is not a complete binary tree. Since there is no left child for node C, i.e., A[5] is vacant. Even though memory is allocated for A [5] it is not used, so wasted unnecessarily.

- 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 tree 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[0] will store the data of the root element.
- The children of a node stored in location K will be stored in locations $(2 \times K+1)$ and $(2 \times K+2)$.
- The maximum size of the array TREE is given as $(2^{h+1}-1)$, where h is the height of the tree.
- An empty tree or sub-tree is specified using NULL. If TREE[0] = NULL, then the tree is empty.

6.4.2 LINKED LIST REPRESENTATION

- 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 bnode {
    struct bnode *left;
    int data;
    struct bnode *right;
};
```

- 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. Consider the binary tree given in Figure 1. The schematic diagram of the linked representation of the binary tree is shown in Figure 2.
- In Figure 2, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

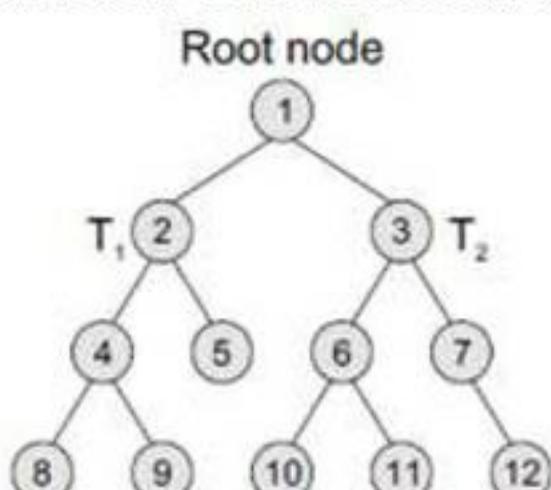


Figure 1: Binary Tree

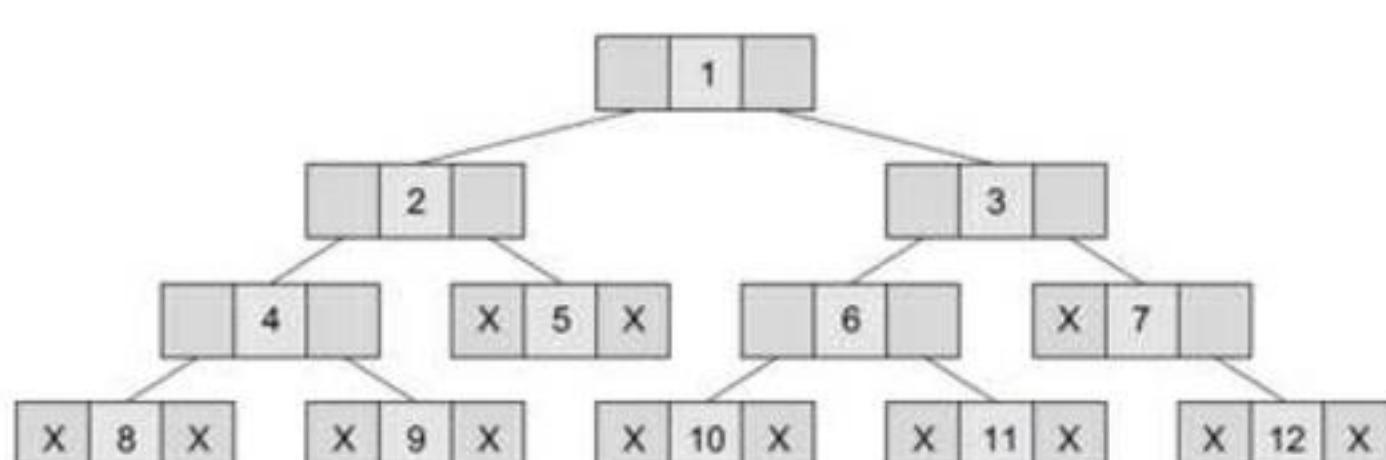


Figure 2: Linked representation of a binary tree

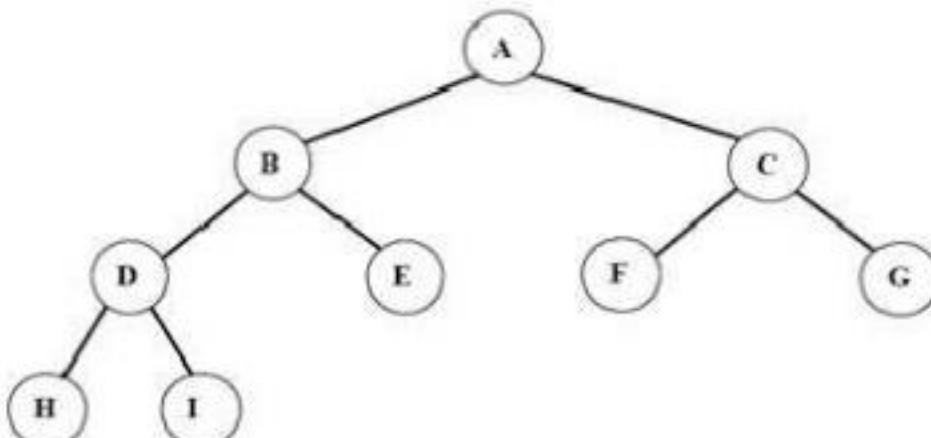
6.5 Operations on Binary tree:

- ✓ **father(n,T):** Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- ✓ **LeftChild(n,T):** Return the left child of node n in tree T. Return NULL if n does not have a left child.
- ✓ **RightChild(n,T):** Return the right child of node n in tree T. Return NULL if n does not have a right child.
- ✓ **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- ✓ **Sibling(n,T):** return the sibling node of node n in tree T. Return NULL if n has no sibling.
- ✓ **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- ✓ **Size(T):** Return the number of nodes in tree T
- ✓ **MakeEmpty(T):** Create an empty tree T
- ✓ **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- ✓ **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- ✓ **Preorder(T):** Traverses all the nodes of tree T in preorder.
- ✓ **postorder(T):** Traverses all the nodes of tree T in postorder
- ✓ **Inorder(T):** Traverses all the nodes of tree T in inorder.

6.6 Binary Tree Traversal

The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner. There are basically three different ways in which we traverse a binary tree.

- Pre-order traversal
- In-order traversal
- Post-order traversal



6.6.1 Pre-order traversal

- A systematic way of visiting all nodes in a binary tree that visits a node, then visits the nodes in the left sub tree of the node, and then visits the nodes in the right sub tree of the node.
 1. Visit the root node
 2. Traverse the left sub tree in preorder
 3. Traverse the right sub tree in preorder
- The preorder traversal output of the given tree is: A B D H I E C F G
The preorder is also known as depth first order.
- C implementation of preorder traversal technique is as follows:

```

void preorder(struct bnode *tree)
{
    if(tree!=NULL)
    {
        printf("%d\n", tree->data);
        preorder(tree->left);
        preorder(tree->right);
    }
}
  
```

6.6.2 In-order traversal

- A systematic way of visiting all nodes in a binary tree that visits the nodes in the left sub tree of a node, then visits the node, and then visits the nodes in the right sub tree of the node.
 1. Traverse the left sub tree in order
 2. Visit the root node
 3. Traverse the right sub tree in order
- In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively.
- The inorder traversal output of the given tree is: H D I B E A F C G
- C implementation of inorder traversal technique is as follows:

```

void inorder(struct bnode *tree)
{
    if(tree!=NULL)
    {
        inorder(tree->left);
        printf("%d\n", tree->data);
        inorder(tree->right);
    }
}
  
```

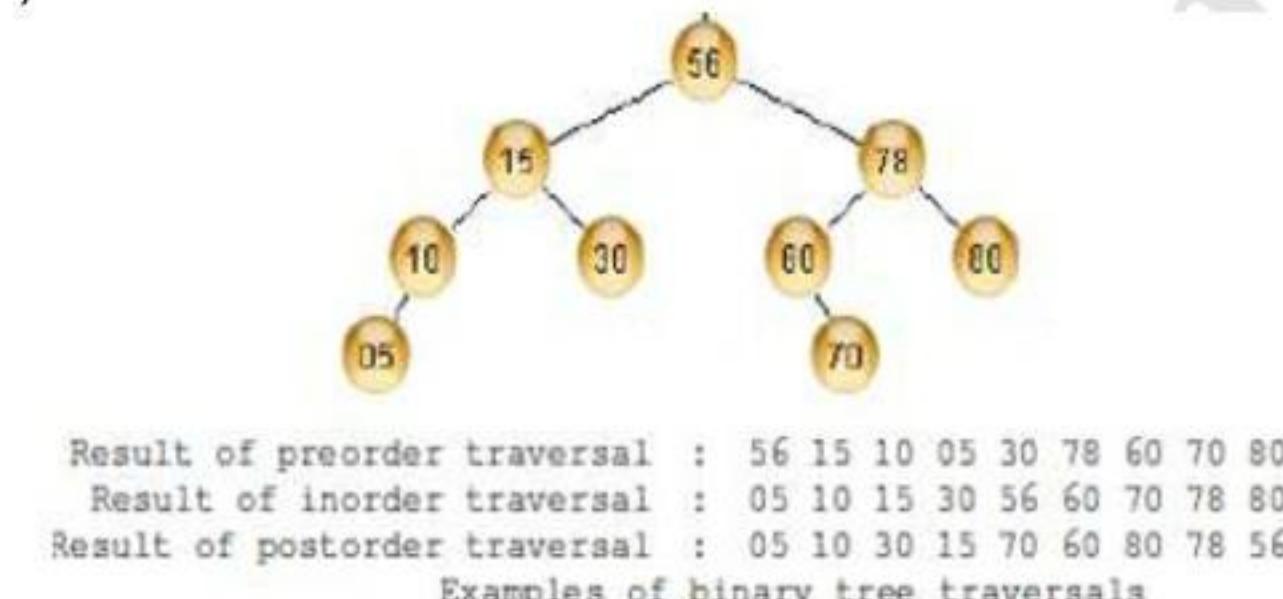
6.6.3 Post-order traversal

- A systematic way of visiting all nodes in a binary tree that visits the node in the left sub tree of the node, then visits the node in the right sub tree in the node, and then visits the node.

 1. Traverse the left sub tree in post order
 2. Traverse the right sub tree in post order
 3. Visit the root node

- In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root.
- The post-order traversal output of the given tree is: H I D E B F G C A
- C implementation of post order traversal technique is as follows:

```
void postorder(struct bnode *tree)
{
    if(tree!=NULL)
    {
        postorder(tree->left);
        postorder(tree->right);
        printf("%d\n", tree->data);
    }
}
```



4 Rebuild a binary tree from Inorder and Preorder/Postorder traversals

This is well known problem where given any two traversals of a tree such as inorder & preorder or inorder & postorder we need to rebuild the tree.

Example: Given preorder and inorder traversal construct a binary tree

Preorder Traversal: 56, 15, 10, 05, 30, 78, 60, 70, 80

Inorder Traversal: 05, 10, 15, 30, 56, 60, 70, 78, 80

[Since we know that the first node in preorder is its root, we can easily locate the root node in the inorder traversal and hence we can obtain left subtree and right subtree from the inorder]

Iteration 1:

Root- {56}

Left subtree- {05, 10, 15, 30}

Right subtree- {60, 70, 78, 80}

Iteration 2:

Root- {15}

Left subtree- {05, 10}

Right subtree- {30}

Root- {78}

Left subtree- {60, 70}

Right subtree- {80}

Iteration 3:

Root- {10}

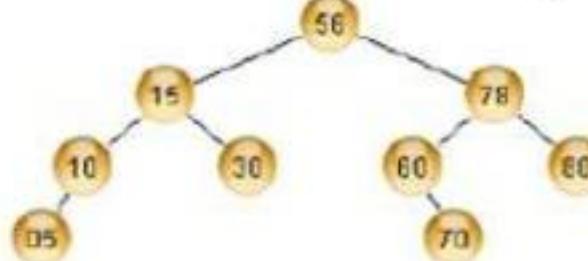
Left subtree- {05}

Right subtree-

Root- {60}

Left subtree-

Right subtree- {70}



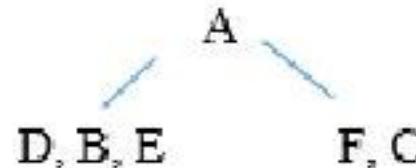
Example: Given inorder and postorder traversal construct a binary tree

Inorder sequence: **D B E A F C**

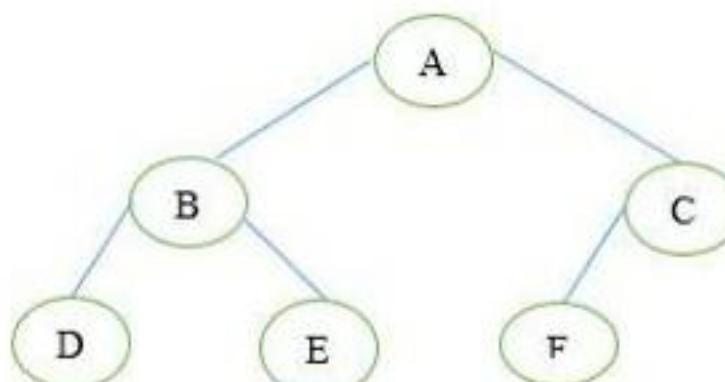
Postorder sequence: **D E B F C A**

[In a postorder sequence, rightmost element is the root of the tree. So we know **A** is root.

Search for **A** in Inorder sequence. Once we know position of **A** (or index of **A**) in inorder sequence, we also know that all elements on left side of **A** are in left subtree and elements on right are in right subtree]



Recursively follow above step and we get the complete tree



6.7 Binary Search Tree (BST)

- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.
- **Definition:**
A binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the following conditions:
 - All keys (if any) in left sub-tree of the root are smaller than that of a root.
 - The key in the root is always less than all the keys in the right sub-tree.
 - The left and right sub-trees of the root are again binary search trees.

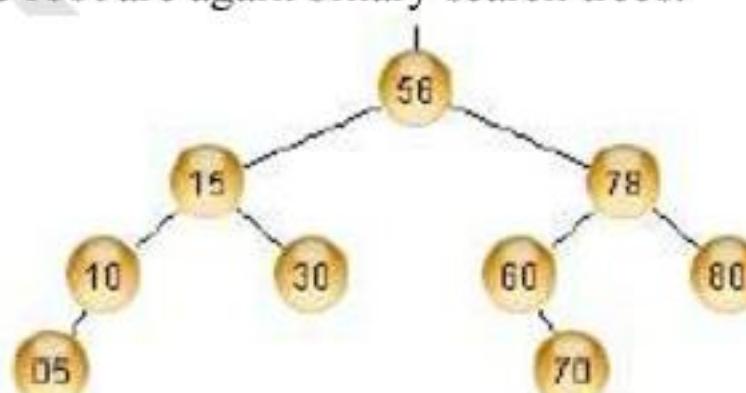


Figure: A Binary Search Tree

- The major advantage of binary search trees is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Operations on Binary search tree (BST):

Following operations can be done in BST:

- **Search(k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- **Insert(k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- **Delete(k, T):** Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

6.7.1 Searching through the BST

- Problem: Search for a given target value in a BST.
- Idea: Compare the target value with the element in the root node.
 - ✓ If the target value is equal, the search is successful.
 - ✓ If target value is less, search the left subtree.
 - ✓ If target value is greater, search the right subtree.
 - ✓ If the subtree is empty, the search is unsuccessful.

BST search algorithm

To find which if any node of a BST contains an element equal to target. *curr* is pointer to hold the node address.

1. Set *curr* to the BST's root.
2. Repeat:
 - 2.1. If *curr* is null:
 - 2.1.1. Terminate with answer none.
 - 2.2. Otherwise, if target is equal to *curr*'s element:
 - 2.2.1. Terminate with answer *curr*.
 - 2.3. Otherwise, if target is less than *curr*'s element:
 - 2.3.1. Set *curr* to *curr*'s left child.
 - 2.4. Otherwise, if target is greater than *curr*'s element:
 - 2.4.1. Set *curr* to *curr*'s right child.
3. End

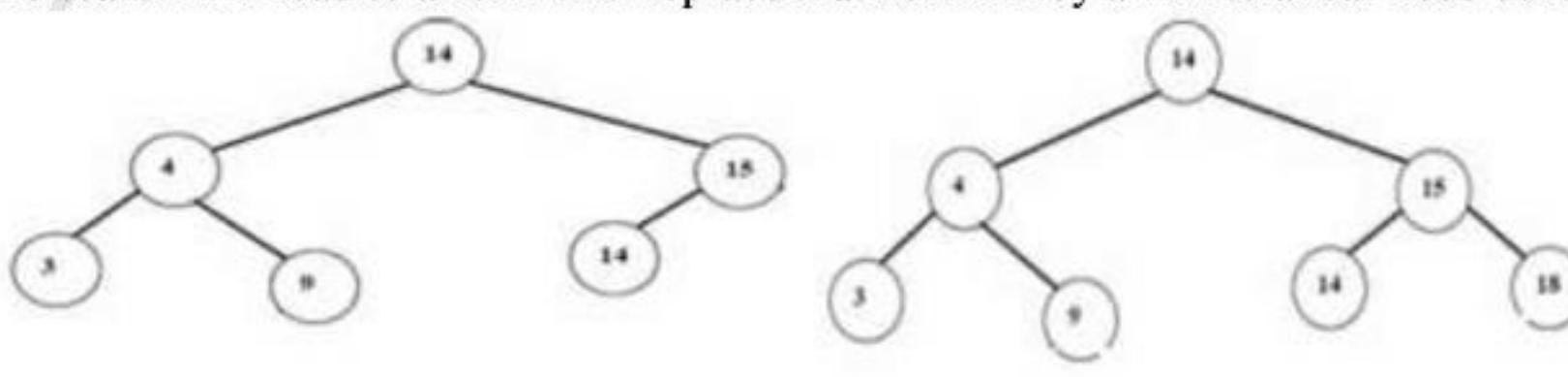
C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
    if(root == NULL)
    {
        printf("The number does not exist");
        exit(1);
    }
    else if (key == root->info)
    {
        printf("The searched item is found");
    }
    else if(key < root->info)
    {
        return BinSearch(root->left, key);
    }
    else
        return BinSearch(root->right, key);
}
```

6.7.2 Insertion of a node in BST

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted.

- **Idea:** To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.



insert(18)

• BST insertion algorithm

To insert the element *elem* into a BST:

1. Set *parent* to null, and set *curr* to the BST's root.
2. Repeat:
 - 2.1. If *curr* is null:
 - 2.1.1. Replace the null link from which *curr* was taken (either the BST's root or *parent*'s left child or *parent*'s right child) by a link to a newly-created leaf node with element *elem*.
 - 2.1.2. Terminate.
 - 2.2. Otherwise, if *elem* is equal to *curr*'s element:
 - 2.2.1. Terminate.
 - 2.3. Otherwise, if *elem* is less than *curr*'s element:
 - 2.3.1. Set *parent* to *curr*, and set *curr* to *curr*'s left child.
 - 2.4. Otherwise, if *elem* is greater than *curr*'s element:
 - 2.4.1. Set *parent* to *curr*, and set *curr* to *curr*'s right child.

3. End

• C function for BST insertion

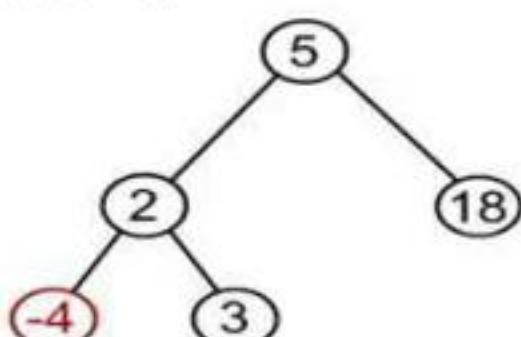
```
void insert(struct bnode *root, int item)
{
    if(root=NULL)
    {
        root=(struct bnode*)malloc (sizeof(struct bnode));
        root->left=root->right=NULL;
        root->info=item;
    }
    else
    {
        if(item<root->info)
            root->left=insert(root->left, item);
        else
            root->right=insert(root->right, item);
    }
}
```

6.7.3 Deleting a node from the BST

While deleting a node from BST, there may be three cases:

1. The node to be deleted may be a leaf node:

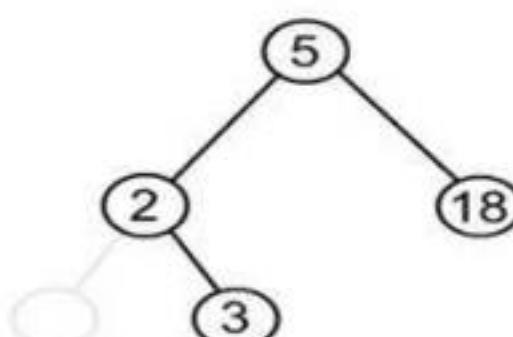
In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.

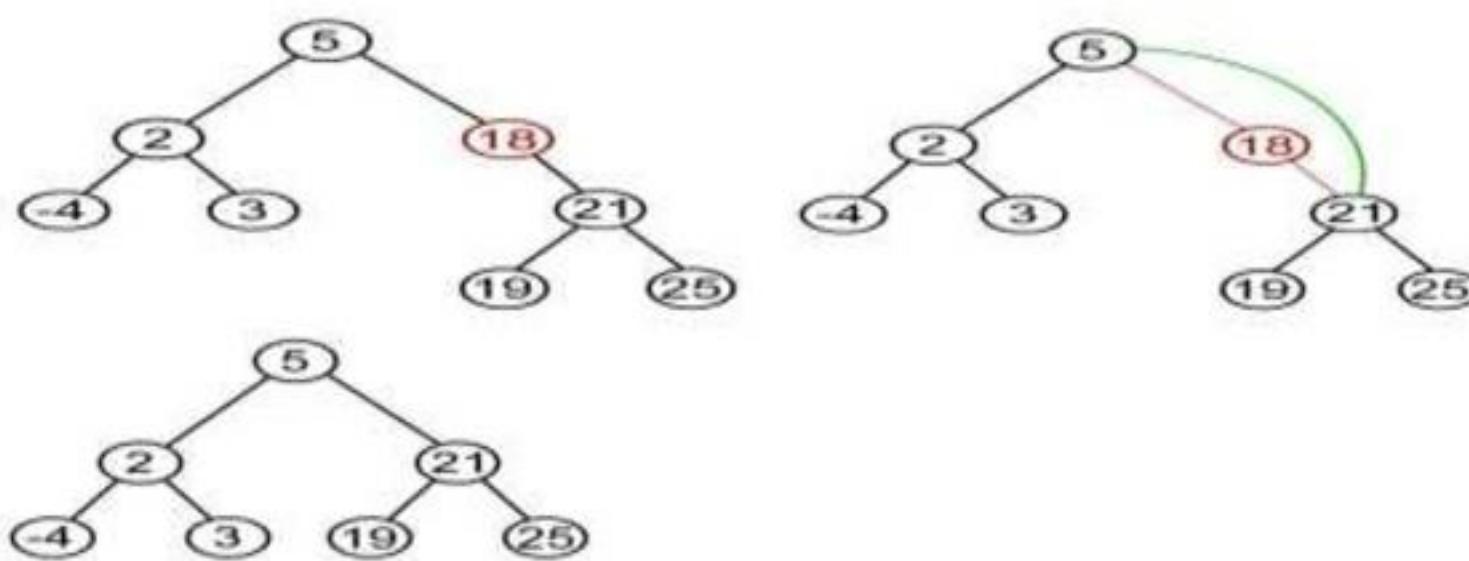


Suppose node to be deleted is -4

2. The node to be deleted has one child

In this case the child of the node to be deleted is appended to its parent node. Suppose node to be deleted is 18



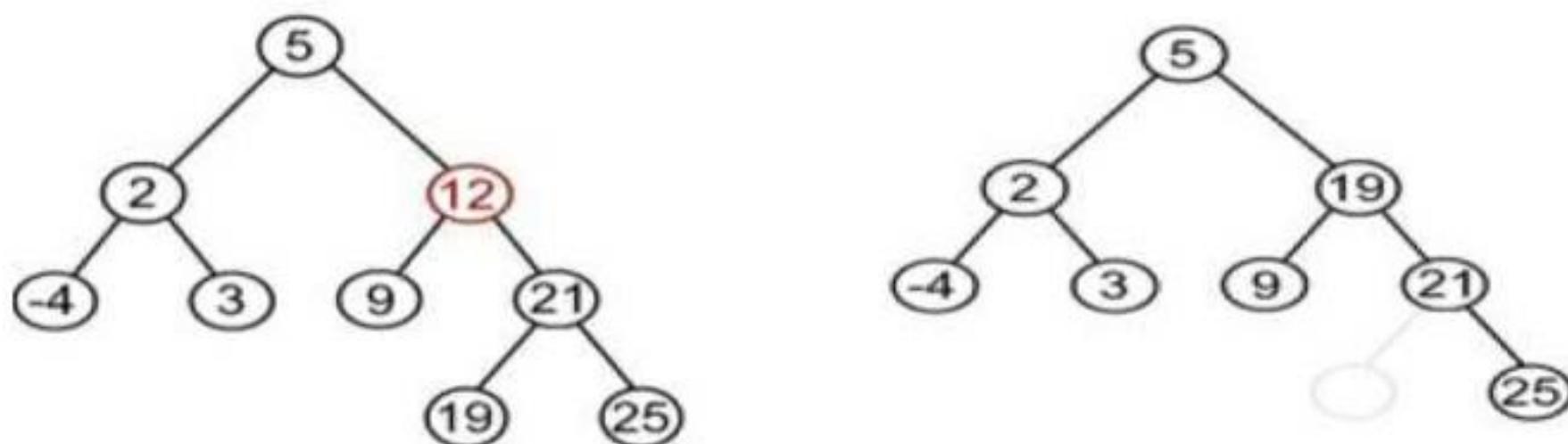


3. the node to be deleted has two children

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.



Suppose node to be deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

General algorithm to delete a node from a BST

1. Start
2. if a node to be deleted is a leaf node at left side then simply delete and set null pointer to its parent's left pointer.
3. if a node to be deleted is a leaf node at right side then simply delete and set null pointer to its parent's right pointer
4. if a node to be deleted has one child then connect its child pointer with its parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
 - a. right most node of its left sub-tree or
 - b. left most node of its right sub-tree.
6. End

C function to delete a node from a BST

```
struct bnode *Delete(struct bnode *root, int item)
{
    struct bnode *temp;
    if(root==NULL)
    {
        printf("Empty tree");
        return;
    } else if(item<root->info)
        root->left=Delete(root->left, item);
    else if(item>root->info)
        root->right=Delete(root->right, item);
    else if(root->left!=NULL &&root->right!=NULL) //node has two children
    {
        temp=find_min(root->right);
```

```

        root->info=temp->info;
        root->right=Delete(root->right, root->info);
    } else
    {
        temp=root;
        if(root->left==NULL)
            root=root->right;
        else if(root->right==NULL)
            root=root->left;
        free(temp);
    }
    return(temp);
}

```

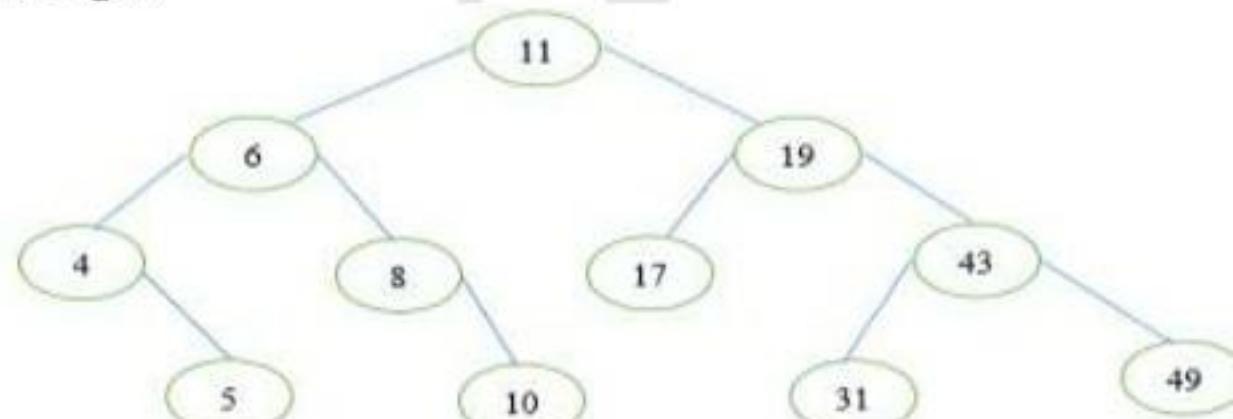
❖ **C function for find minimum element**

```

struct bnode *find_min(struct bnode *root)
{
    if(root==NULL)
        return 0;
    else if(root->left==NULL)
        return root;
    else
        return(find_min(root->left));
}

```

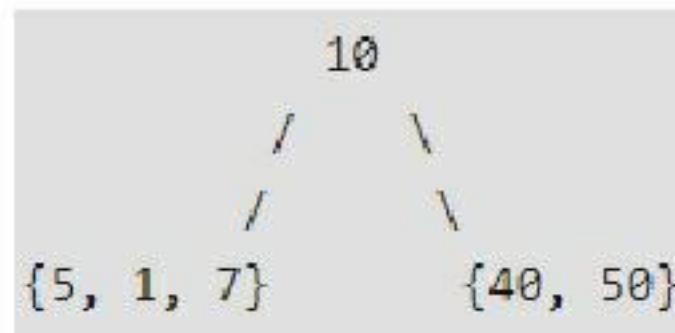
- Given a sequence of numbers: 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31, Draw a binary search tree by inserting the given numbers from left to right.



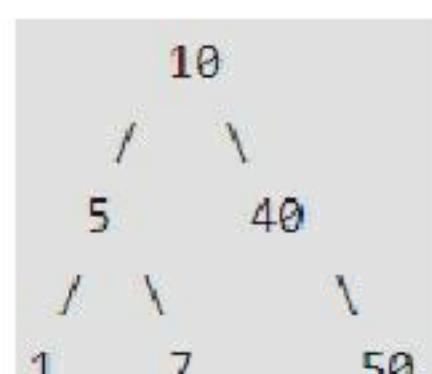
❖ **Construct BST from given preorder traversal**

10, 5, 1, 7, 40, 50

In given data, 10 is the first element (The first element of preorder traversal is always root), so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as following.



We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.



6.8 BALANCED BINARY TREE

A balanced binary tree is one in which the largest path through the left sub tree is the same length as the largest path of the right sub tree, i.e., from root to leaf. Searching time is very less in balanced binary trees compared to unbalanced binary tree. i.e., balanced trees are used to maximize the efficiency of the operations on the tree. There are two types of balanced trees:

1. Height Balanced Trees
2. Weight Balanced Trees

6.8.1 Height Balanced Tree

In height balanced trees balancing the height is the important factor. There are two main approaches, which may be used to balance (or decrease) the depth of a binary tree:

- a) Insert a number of elements into a binary tree in the usual way, using the algorithm of binary search Tree insertion. After inserting the elements, copy the tree into another binary tree in such a way that the tree is balanced. This method is efficient if the data(s) are continually added to the tree.
- b) Another popular algorithm for constructing a height balanced binary tree is the AVL tree

6.8.2 Weight Balanced Tree

A weight-balanced tree is a balanced binary tree in which additional weight field is also there. The nodes of a weight-balanced tree contain four fields:

- a) Data Element
- b) Left Pointer
- c) Right Pointer
- d) A probability or weight field
- The data element, left and right pointer fields are same as that in any other tree node.
- The probability field is a specially added field for a weight-balanced tree. This field holds the probability of the node being accessed again, that is the number of times the node has been previously searched for.
- When the tree is created, the nodes with the highest probability of access are placed at the top. That is the nodes that are most likely to be accessed have the lowest search time.
- And the tree is balanced if the weights in the right and left sub trees are as equal as possible.
- The average length of search in a weighted tree is equal to the sum of the probability and the depth for every node in the tree.
- The root node contain highest weighted node of the tree or sub tree.
- The left sub tree contains nodes where data values are less than the current root node, and the right sub tree contain the nodes that have data values greater than the current root node.

6.9 AVL TREES

Two Russian Mathematicians, G.M. Adel'son Vel'sky and E.M. Landis developed algorithm in 1962; here the tree is called AVL Tree.

An AVL tree is a binary tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees. Each node in the AVL Tree possesses any one of the following properties:

- a. A node is called **left heavy**, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree.
- b. A node is called **right heavy**, if the largest path in its right sub tree is one level larger than the largest path of its left sub tree.
- c. The node is called **balanced**, if the largest paths in both the right and left sub trees are equal. Figure shows some example for AVL trees.

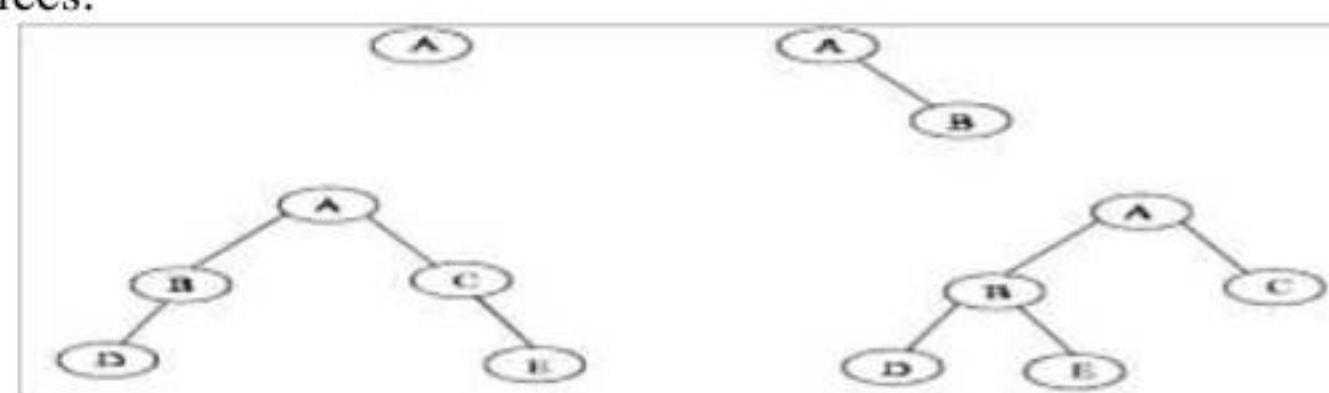


Figure: AVL Trees

The construction of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated. If the new node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done.

Algorithm for inserting a node in an AVL Tree

1. Insert the node in the same way as in an ordinary binary tree.
2. Trace a path from the new nodes, back towards the root for checking the height difference of the two subtrees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layers immediately below.
4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.
5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

The above algorithm will be illustrated with an example shown in Figure A, which is an unbalance tree. We have to apply the rotation to the nodes 40, 50 and 60 so that a balance tree is generated.

Since the three nodes are lying in a straight line, single rotation is applied to restore the balance.

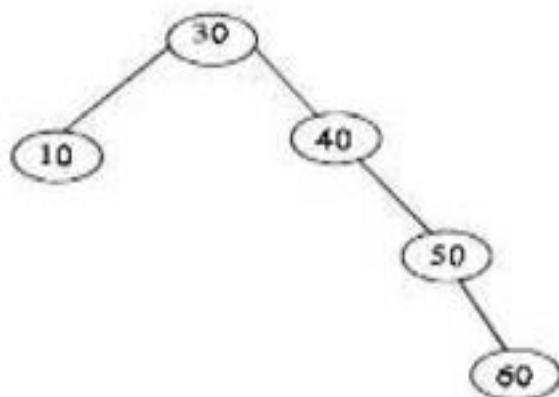


Figure A

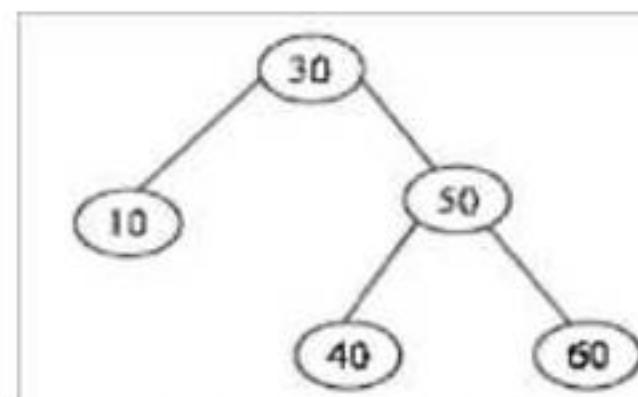


Figure B

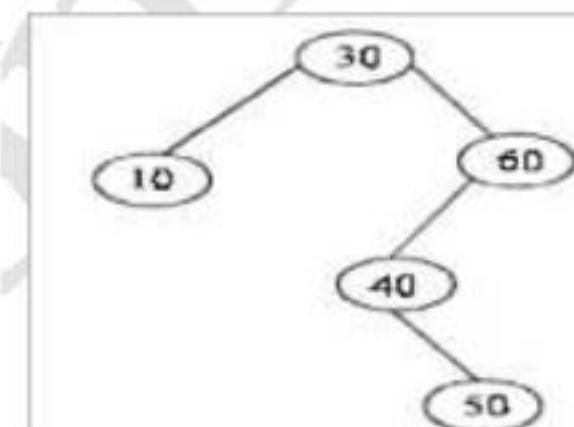


Figure C

Figure B is a balance tree of the unbalanced tree in Figure A. Consider a tree in Figure C to explain the double rotation. While tracing the path, first imbalance is detected at node 60. We restrict our attention to this node and the two nodes immediately below it (40 and 50). These three nodes form a dogleg pattern. That is there is bend in the path. Therefore we apply double rotation to correct the balance. A double rotation, as its name implies, consists of two single rotations, which are in opposite direction. The first rotation occurs on the two layers (or levels) below the node where the imbalance is found (i.e., 40 and 50). Rotate the node 50 up by replacing 40, and now 40 become the child of 50 as shown in Figure D.

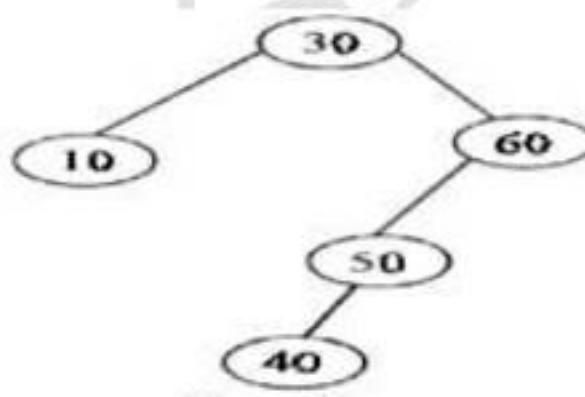


Figure D

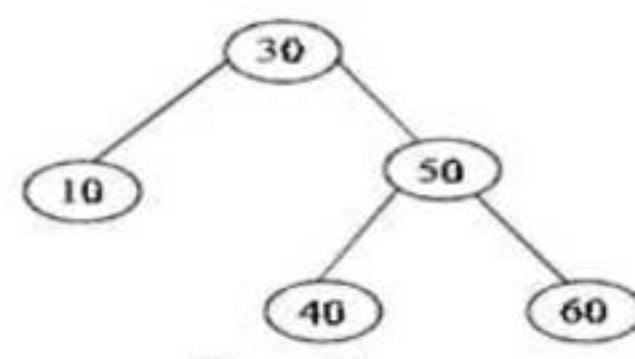
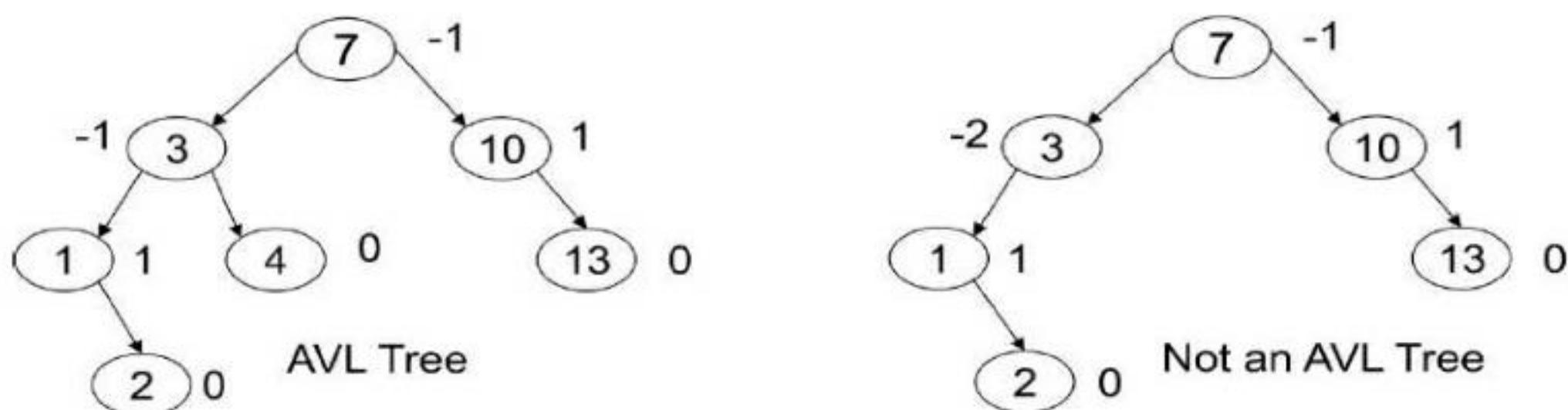


Figure E

Apply the second rotation, which involves the nodes 60, 50 and 40. Since these three nodes are lying in a straight line, apply the single rotation to restore the balance, by replacing 60 by 50 and placing 60 as the right child of 50 as shown in Figure E. Balanced binary tree is a very useful data structure for searching the element with less time.

Need for AVL tree

- The disadvantage of a binary search tree is that its height can be as large as $N-1$
- This means that the time needed to perform insertion and deletion and many other operations can be $O(N)$ in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Omega(\log N)$
- Thus, our goal is to keep the height of a binary search tree $O(\log N)$
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.



HEIGHTS OF AVL TREE

An AVL tree is a special type of binary tree that is always "partially" balanced. The criteria that is used to determine the "level" of "balanced-ness" which is the difference between the heights of subtrees of a root in the tree. The "height" of tree is the "number of levels" in the tree. The height of a tree is defined as follows:

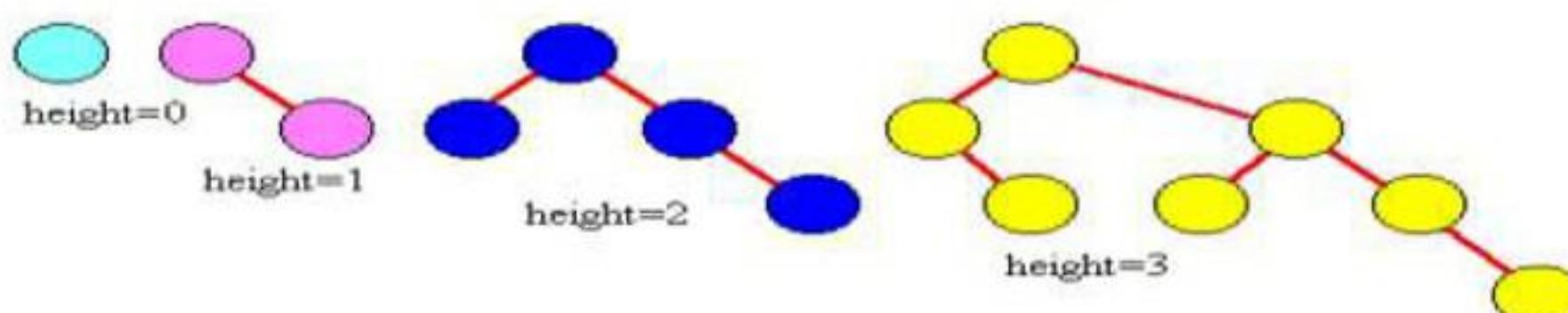
1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.
4. The height of a leaf is 1. The height of a null pointer is zero.

The height of an internal node is the maximum height of its children plus 1.

FINDING THE HEIGHT OF AVL TREE

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1. AVL trees are HB-k trees (height balanced trees of order k) of order HB-1. The following is the height differential formula: $|Height(T_l) - Height(T_r)| \leq k$

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same.



An AVL tree is a binary search tree with a balanced condition.

Balance Factor(BF) = H_l --- H_r.

H_l => Height of the left subtree.

H_r => Height of the right subtree.

If BF = { -1, 0, 1 } is satisfied, only then the tree is balanced.

AVL tree is a Height Balanced Tree.

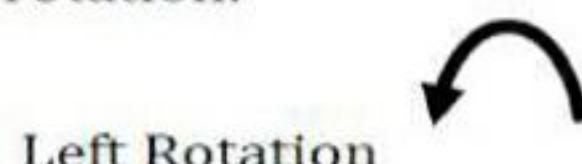
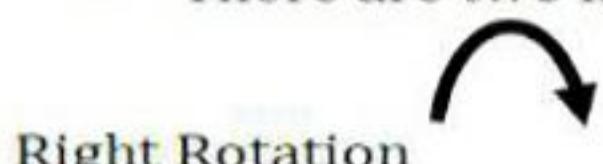
If the calculated value of BF goes out of the range, then balancing has to be done.

Rotation :

Modification to the tree. i.e., If the AVL tree is Imbalanced, proper rotations has to be done.

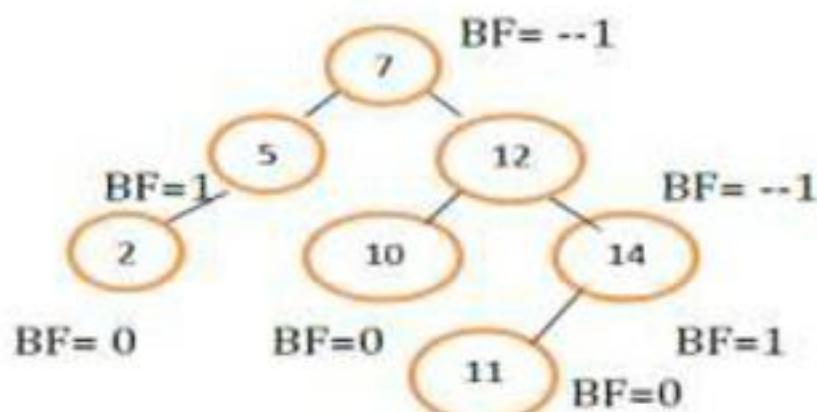
A rotation is a process of switching children and parents among two or three adjacent nodes to restore balance to a tree.

- There are two kinds of single rotation:



An insertion or deletion may cause an imbalance in an AVL tree.

The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.

Balance Factor :

This Tree is an AVL Tree and a height balanced tree.

An AVL tree causes imbalance when any of following condition occurs:

- An insertion into Right child's right subtree.
- An insertion into Left child's left subtree.
- An insertion into Right child's left subtree.
- An insertion into Left child's right subtree.

These imbalances can be overcome by,

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.

1. Single Rotation

- LL (Left -- Left rotation)
- RR (Right -- Right rotation)

2. Double Rotation

- RL (Right -- Left rotation) --- Do single Right, then single Left.
- LR (Left -- Right rotation) --- Do single Left, then single Right.

General Representation of Single Rotation

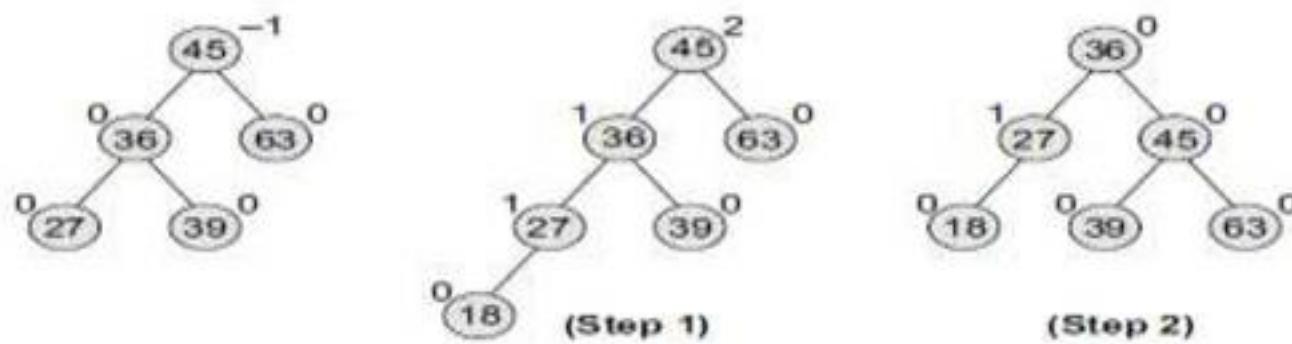
1. LL Rotation:

- The left child x of a node y becomes y's parent.
- y becomes the right child of x.
- The right child T₂ of x, if any, becomes the left child of y.



Note: The pivot of the rotation is the deepest unbalanced node

Example: Consider the AVL tree given in Figure and insert 18 into it.



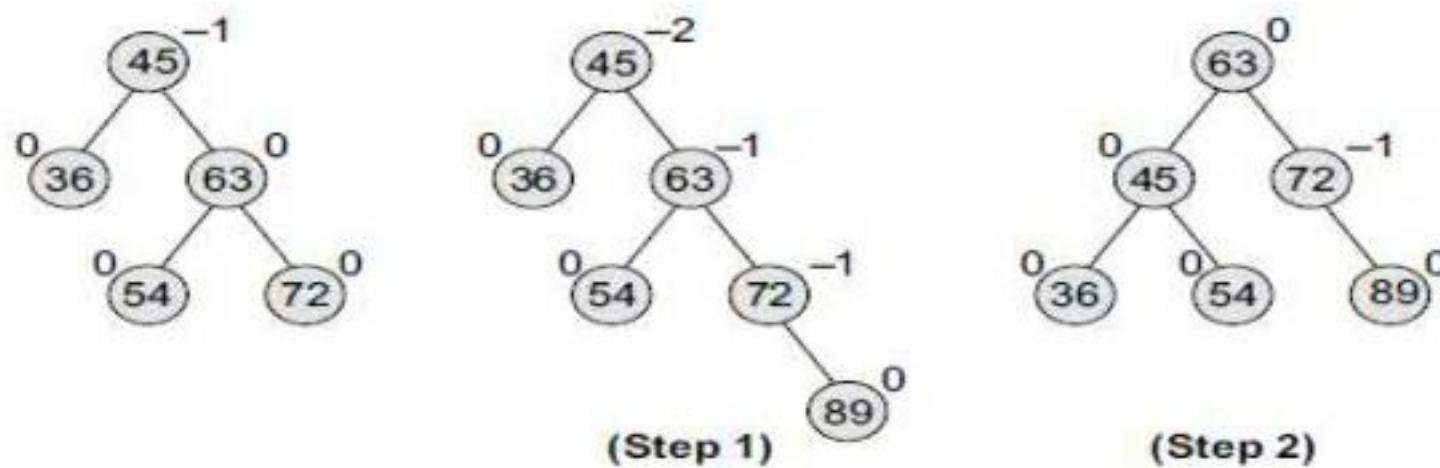
2. RR Rotation :

- The right child y of a node x becomes x 's parent.
- x becomes the left child of y .
- The left child T_2 of y , if any, becomes the right child of x .



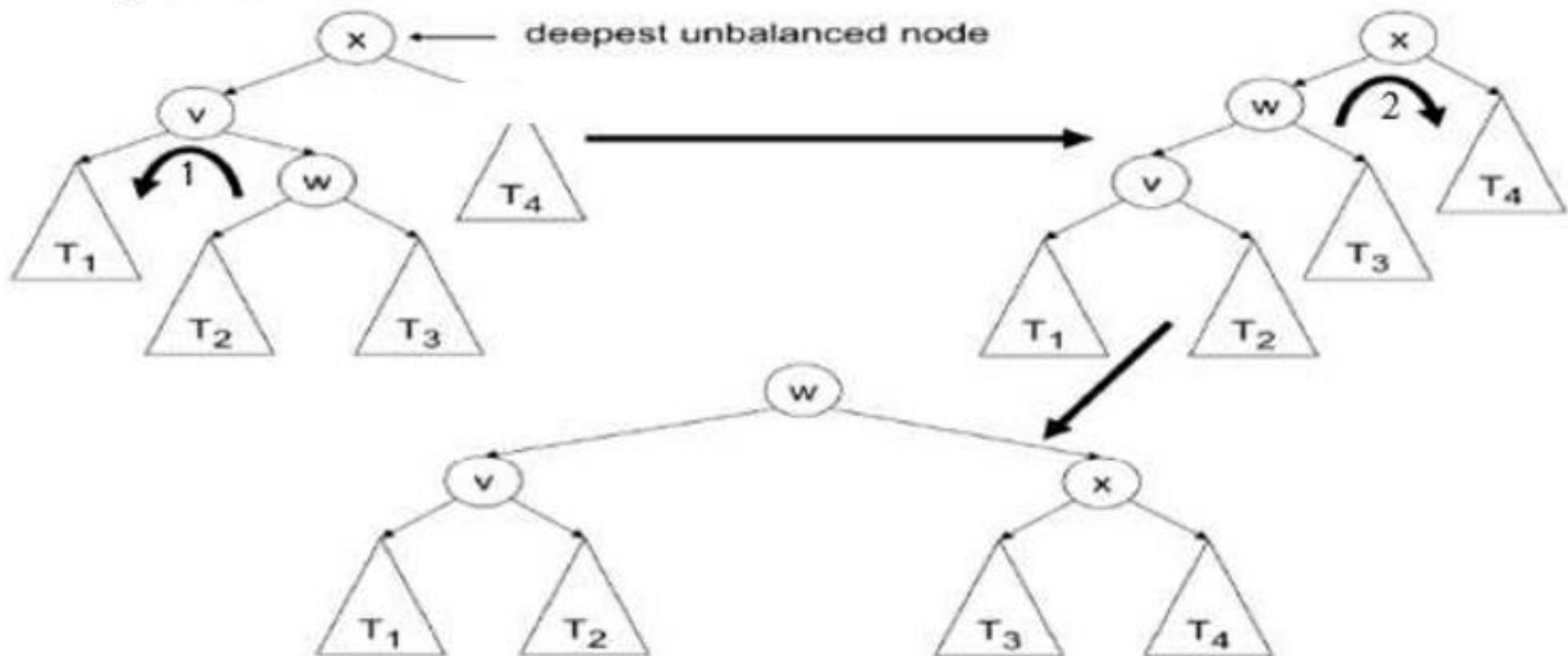
Note: The pivot of the rotation is the deepest unbalanced node

Example: Consider the AVL tree given in Figure and insert 89 into it.



General Representation of Double Rotation

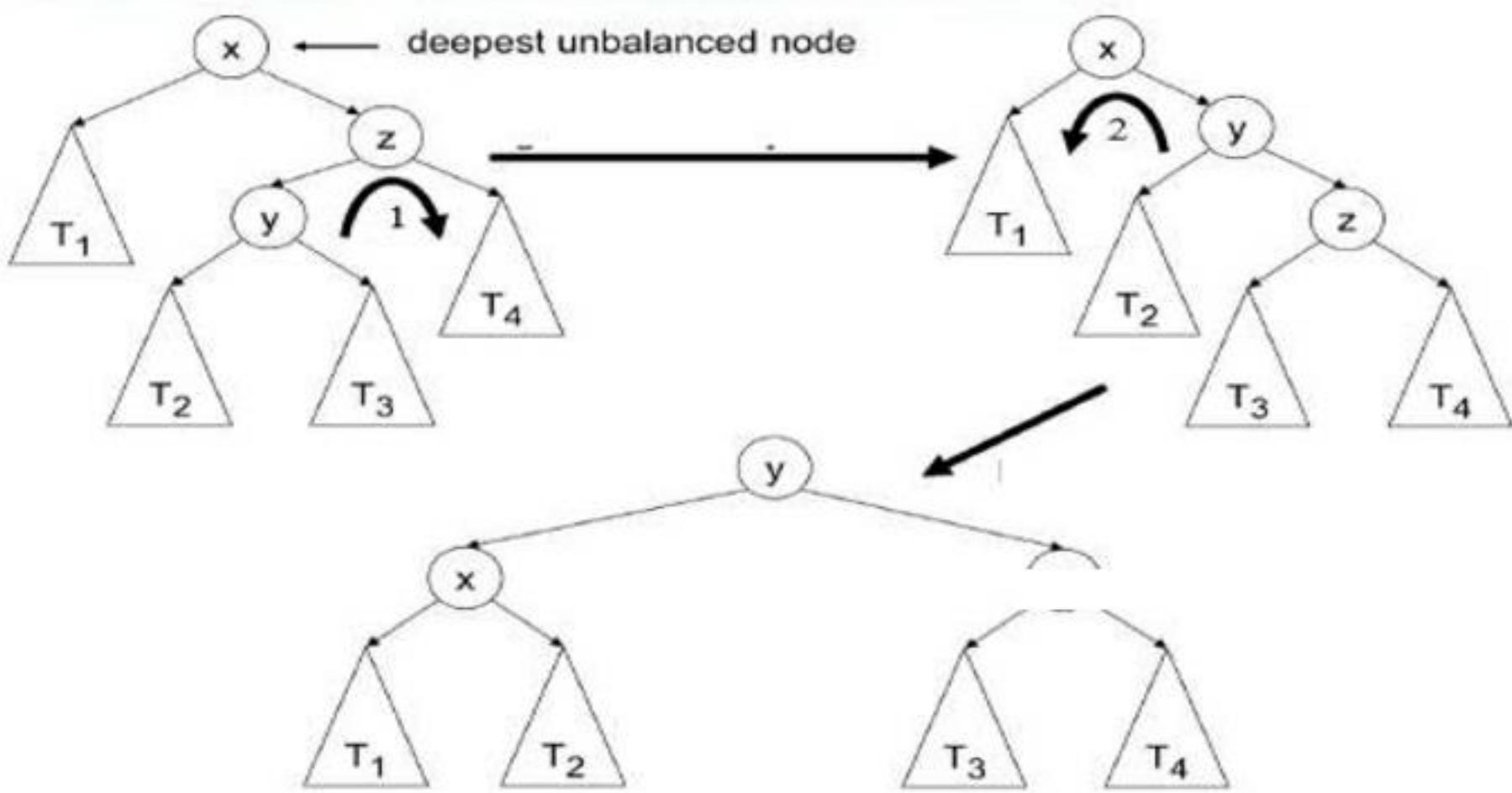
1. Left –Right Rotation



Note: First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node

2. Right-Left Rotation

Note: First pivot is the right child of the deepest unbalanced node; second pivot is the deepest unbalanced node



EXAMPLE:

LET US CONSIDER INSERTING OF NODES 20,10,40,50,90,30,60,70 in an AVL TREE

Step 1:(Insert the value 20)

(20) BF = 0

Step 2: (Insert the value 10)

(20) BF = 1
(10) BF = 0

Step 3: (Insert the value 40)

(20) BF = 1
BF = 0 (10) (40) BF = 0

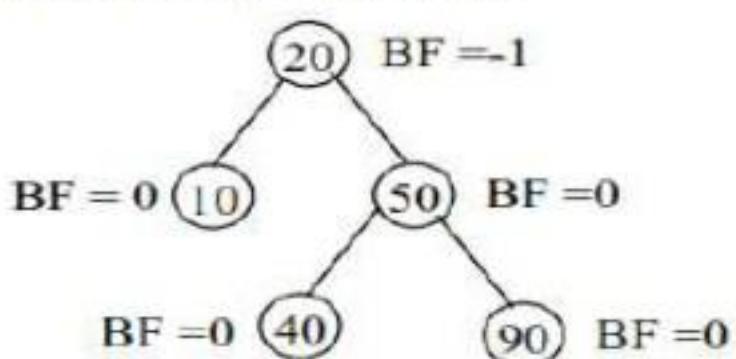
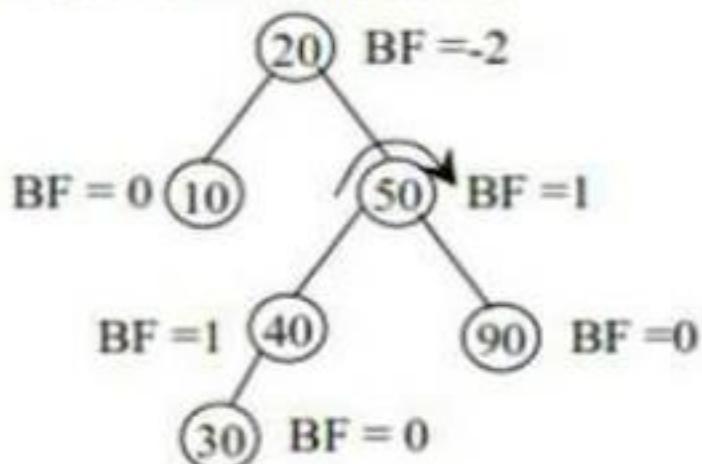
Step 4:(Insert the value 50)

(20) BF = -1
BF = 0 (10) (40) BF = -1
 (50) BF = 0

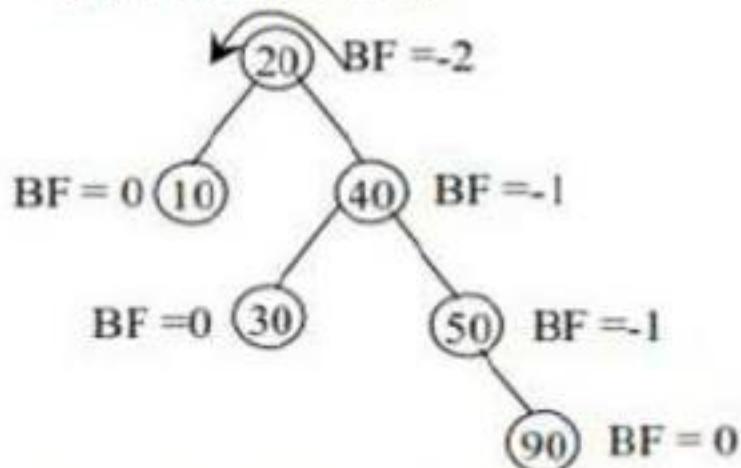
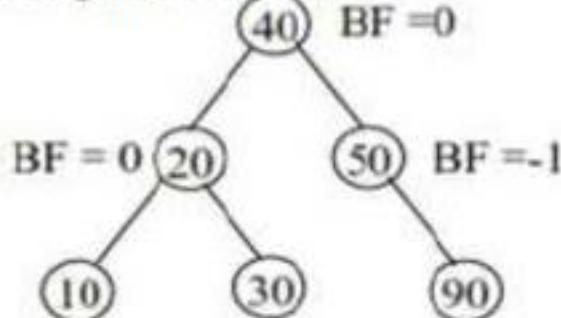
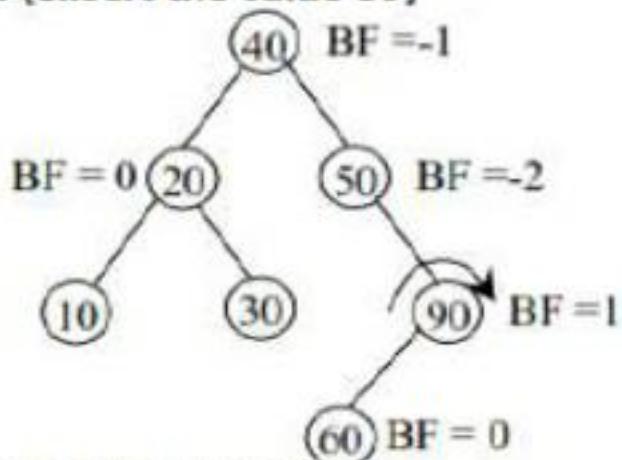
Step 5: (Insert the value 90)

(20) BF = -2
BF = 0 (10) (40) BF = -2
 (50) BF = -1
 (90) BF = 0

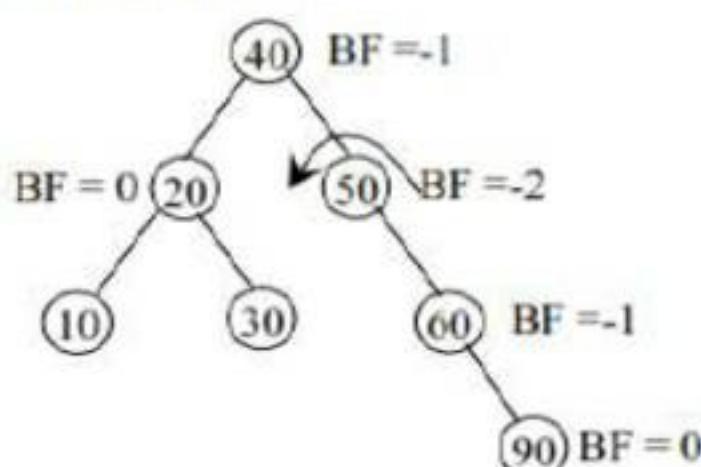
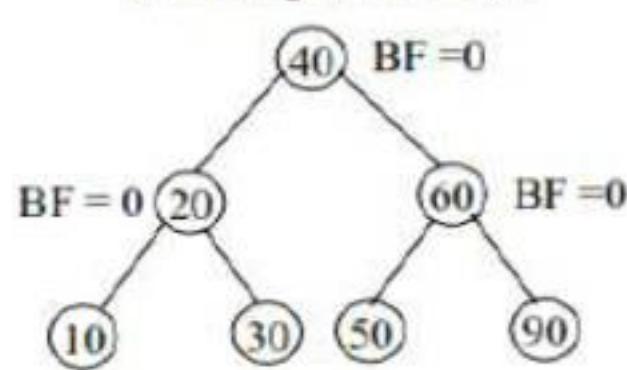
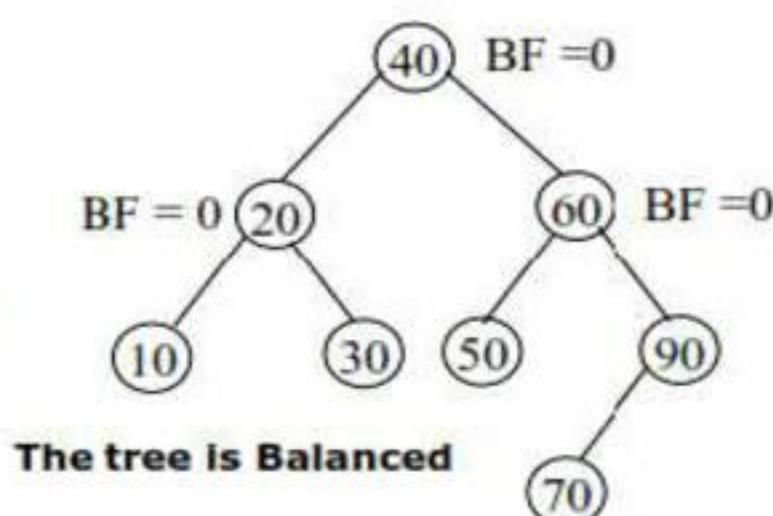
Now the tree is unbalanced due to the insertion of node in the Right subtree of the Right subtree. So we have to make single Right rotation in the node '40'.

After the Right Rotation:**Now the tree is Balanced****Step 6: (Insert the value 30)**

Now the tree is unbalanced due to the insertion of node '30' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '50' and then with Right on the node '20'.

After Left Rotation:**After Right Rotation:****Now the tree is Balanced****Step 7: (Insert the value 60)**

Now the tree at the node '50' is unbalanced due to the insertion of node '60' in the Left subtree of the Right subtree. So we have to make Double rotation first with Left on the node '90' and then with Right on the node '50'.

After Left Rotation:**After Right Rotation:****Now the tree is Balanced****Step 8: (Insert the value 70)****The tree is Balanced**

APPLICATIONS

AVL trees play an important role in most computer related applications. The need and use of AVL trees are increasing day by day. Their efficiency and less complexity add value to their reputation. Some of the applications are

- Contour extraction algorithm
- Parallel dictionaries
- Compression of computer files
- Translation from source language to target language
- Spell checker

ADVANTAGES OF AVL TREE

- AVL trees guarantee that the difference in height of any two subtrees rooted at the same node will be at most one. This guarantees an asymptotic running time of $O(\log(n))$ as opposed to $O(n)$ in the case of a standard BST.
- Height of an AVL tree with n nodes is always very close to the theoretical minimum.
- Since the AVL tree is height balanced the operation like insertion and deletion have low time complexity.
- Since tree is always height balanced. Recursive implementation is possible.
- The height of left and the right sub-trees should differ by atmost 1. Rotations are possible.

DISADVANTAGES OF AVL TREE

- One limitation is that the tree might be spread across memory
- As you need to travel down the tree, you take a performance hit at every level down
- One solution: store more information on the path
- Difficult to program & debug; more space for balance factor.
- Asymptotically faster but rebalancing costs time.
- Most larger searches are done in database systems on disk and use other structures

6.10 Multi-way Search Trees

- We have discussed that every node in a binary search tree contains one value and two pointers, left and right, which point to the node's left and right sub-trees, respectively. The structure of a binary search tree node is shown in Figure A.

Pointer to left sub-tree	Value or Key of the node	Pointer to right sub-tree
--------------------------	--------------------------	---------------------------

Figure A: Structure of a binary search tree node

- The same concept is used in an M-way search tree which has $M - 1$ values per node and M subtrees. In such a tree, M is called the degree of the tree. Note that in a binary search tree $M = 2$, so it has one value and two sub-trees. In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains $M - 1$ keys, where $M > 2$. The structure of an M-way search tree node is shown in Figure B.

P_0	K_0	P_1	K_1	P_2	K_2	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-------	-------	-------	-------	-----------	-----------	-------

Figure B: Structure of an M-way search tree node

In the structure shown, $P_0, P_1, P_2, \dots, P_n$ are pointers to the node's sub-trees and $K_0, K_1, K_2, \dots, K_{n-1}$ are the key values of the node. All the key values are stored in ascending order. That is, $K_i < K_{i+1}$ for $0 \leq i \leq n-2$.

- In an M-way search tree, it is not compulsory that every node has exactly $M-1$ values and M subtrees. Rather, the node can have anywhere from 1 to $M-1$ values, and the number of sub-trees can vary from 0 (for a leaf node) to $i + 1$, where i is the number of key values in the node. M is thus a fixed upper limit that defines how many key values can be stored in the node.

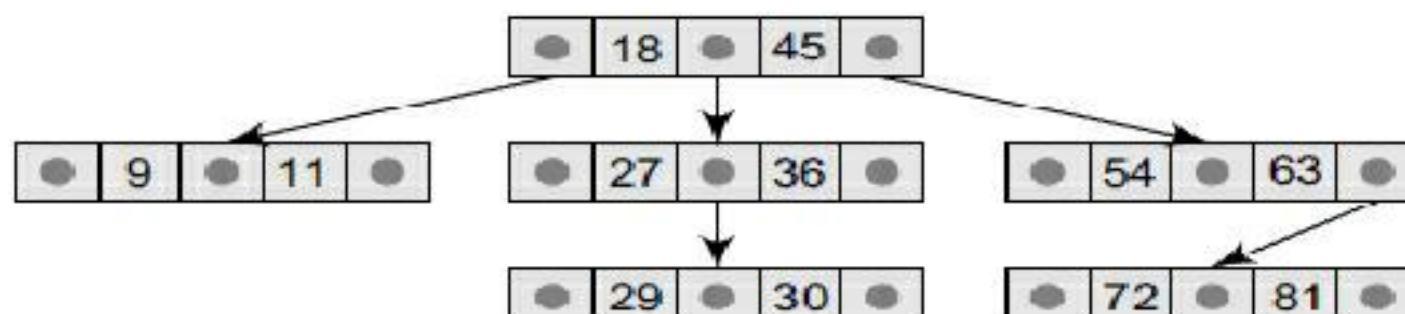


Figure C: M-way search tree of order 3

- Consider the M-way search tree shown in Figure C. Here $M = 3$. So a node can store a maximum of two key values and can contain pointers to three sub-trees.
- In our example, we have taken a very small value of M so that the concept becomes easier, but in practice, M is usually very large. Using a 3-way search tree, let us lay down some of the basic properties of an M-way search tree.
 - Note that the key values in the sub-tree pointed by p_0 are less than the key value κ_0 . Similarly, all the key values in the sub-tree pointed by p_1 are less than κ_1 , so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by p_i are less than κ_i , where $0 \leq i \leq n-1$.
 - Note that the key values in the sub-tree pointed by p_1 are greater than the key value κ_0 . Similarly, all the key values in the sub-tree pointed by p_2 are greater than κ_1 , so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by p_i are greater than κ_{i-1} , where $0 \leq i \leq n-1$.

In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

6.11 B – Tree

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access. A B tree of order m can have a maximum of $m-1$ keys and m pointers to its sub-trees. A B tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time. A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

1. Every node in the B tree has at most (maximum) m children.
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) $m/2$ children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.
3. The root node has at least two children if it is not a terminal (leaf) node.
4. All leaf nodes are at the same level.

An internal node in the B tree can have n number of children, where $0 \leq n \leq m$. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least $m/2$ children. As B tree of order 4 is given in Fig. 11.4.

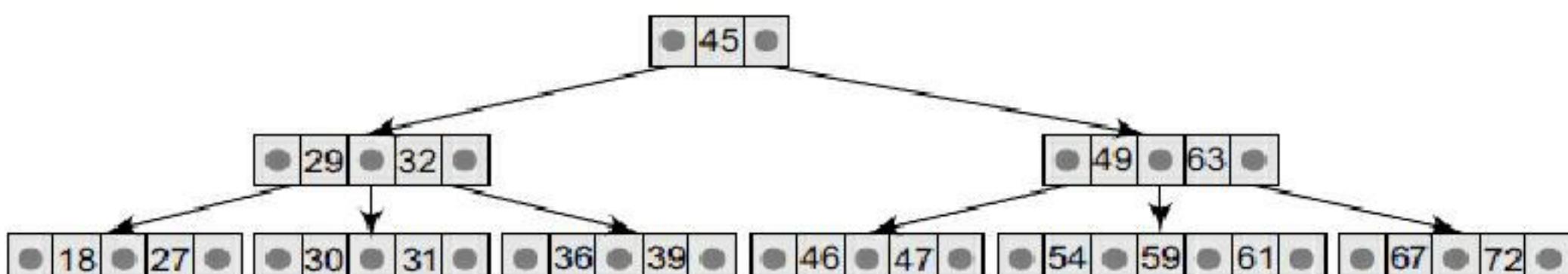


Figure 11.4 B tree of order 4

While performing insertion and deletion operations in a B tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split.

6.11.1 Searching for an Element in a B Tree

Searching for an element in a B tree is similar to that in binary search trees. Consider the B tree given in Fig. 11.4. To search for 59, we begin at the root node. The root node has a value 45 which is less than 59. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since $49 \leq 59 \leq 63$, we traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful.

Take another example. If you want to search for 9, then we traverse the left sub-tree of the root node. The left sub-tree has two key values, 29 and 32. Again, we traverse the left sub-tree of 29. We find that it has two key values, 18 and 27. There is no left sub-tree of 18, hence the value 9 is not stored in the tree.

Since the running time of the search operation depends upon the height of the tree, the algorithm to search for an element in a B tree takes $O(\log_2 n)$ time to execute.

6.11.2 Inserting a New Element in a B Tree

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

1. Search the B tree to find the leaf node where the new key value should be inserted.
2. If the leaf node is not full, that is, it contains less than $m-1$ key values, then insert the new element in the node keeping the node's elements ordered.
3. If the leaf node is full, that is, the leaf node already contains $m-1$ key values, then
 - (a) insert the new value in order into the existing set of keys,
 - (b) split the node at its median into two nodes (note that the split nodes are half full), and
 - (c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Example 11.1 Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.

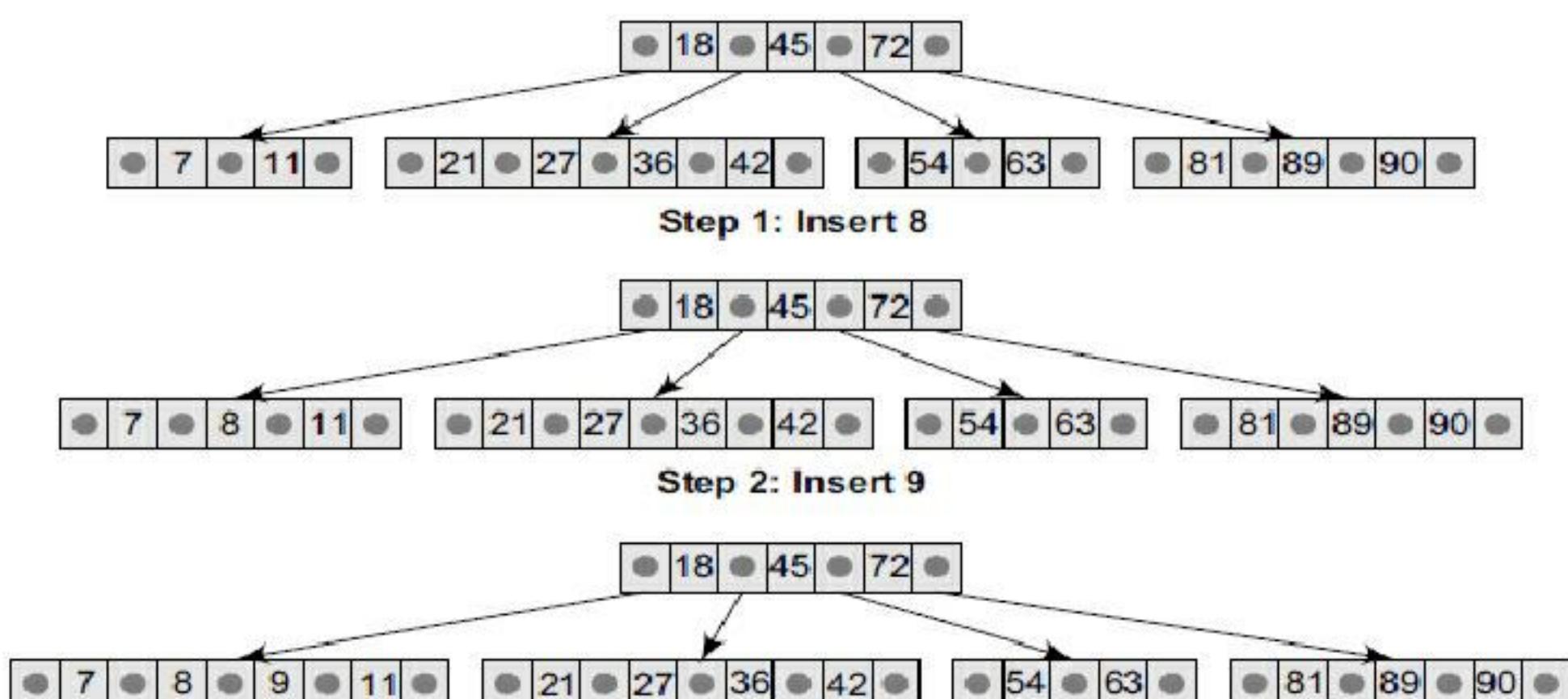


Figure 11.5(a)

Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

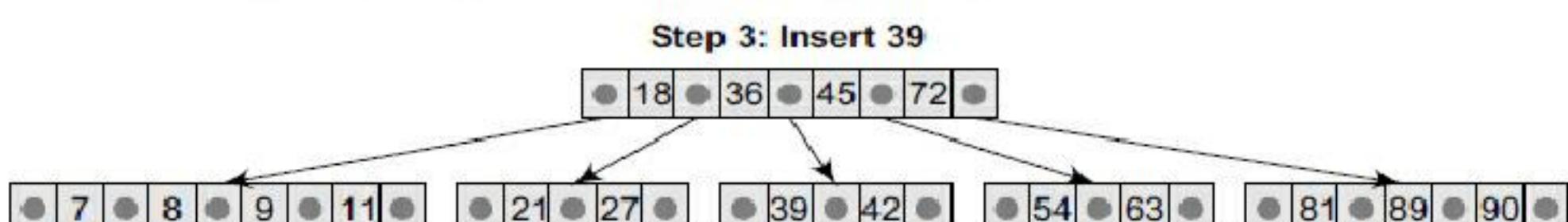


Figure 11.5(b)

Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.

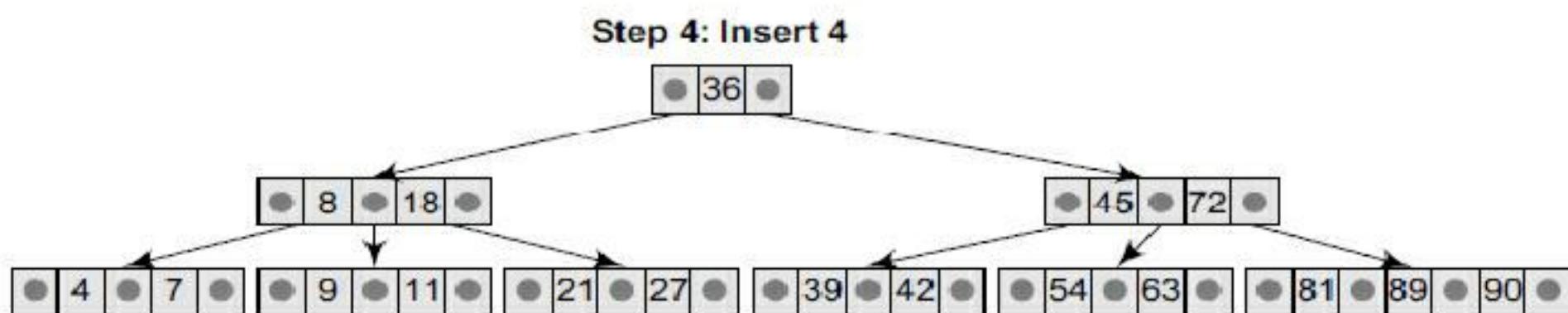


Figure 11.5(c) B tree

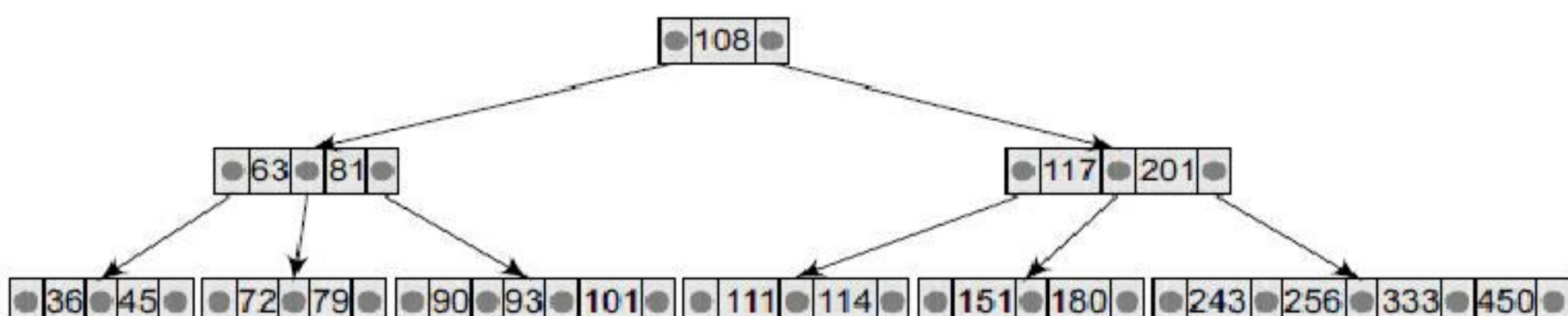
6.11.3 Deleting an Element from a B Tree

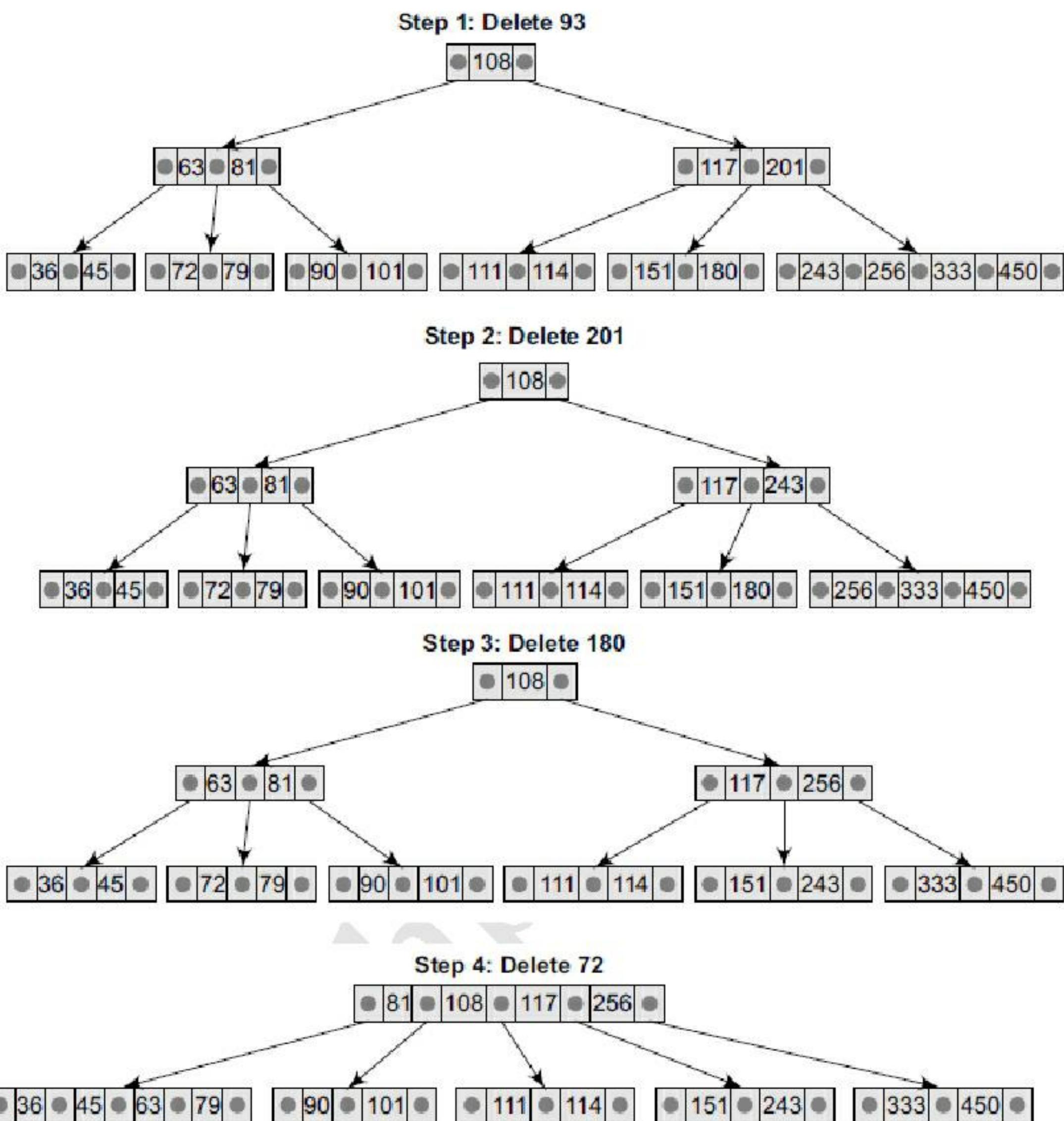
Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.
2. If the leaf node contains more than the minimum number of key values (more than $m/2$ elements), then delete the value.
3. Else if the leaf node does not contain $m/2$ elements, then fill the node by taking an element either from the left or from the right sibling.
 - (a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
 - (b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, m). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

Example 11.2 Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it (Fig. 11.6(a)).

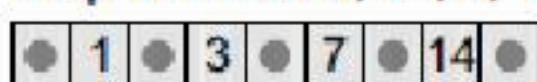


**Figure 11.6** B tree

Example 11.4 Create a B tree of order 5 by inserting the following elements:

3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.

Step 1: Insert 3, 14, 7, 1

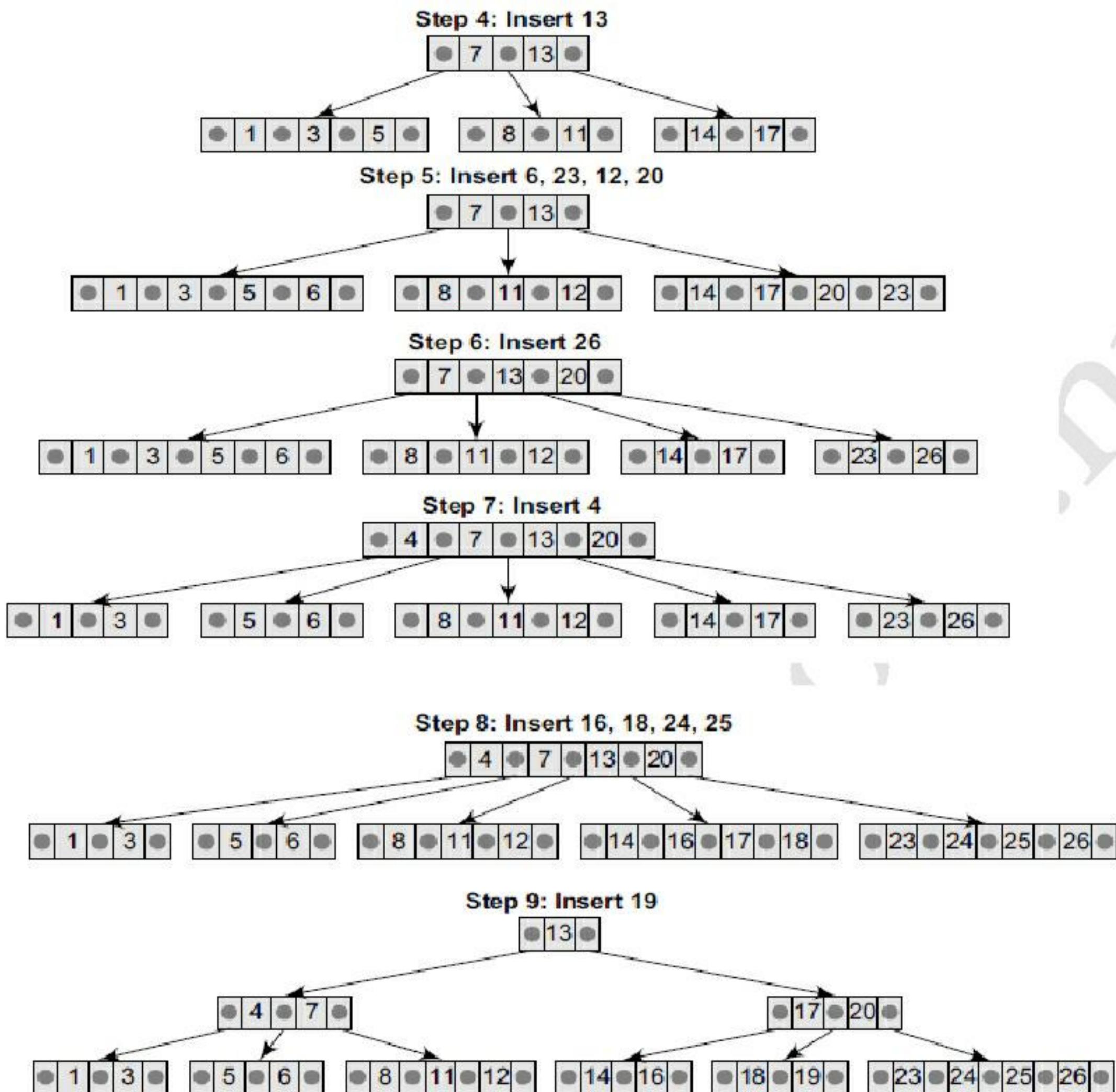


Step 2: Insert 8



Step 3: Insert 5, 11, 17



**Figure 11.8** B tree

6.11.4 Applications of B Trees

A database is a collection of related data. The prime reason for using a database is that it stores organized data to facilitate its users to update, retrieve, and manage the data. The data stored in the database may include names, addresses, pictures, and numbers. For example, a teacher may wish to maintain a database of all the students that includes the names, roll numbers, date of birth, and marks obtained by every student.

Nowadays, databases are used in every industry to store hundreds of millions of records. In the real world, it is not uncommon for a database to store gigabytes and terabytes of data. For example, a telecommunication company maintains a customer billing database with more than 50 billion rows that contains terabytes of data. We know that primary memory is very expensive and is capable of storing very little data as compared to secondary memory devices like magnetic disks. Also, RAM is volatile in nature and we cannot store all the data in primary memory. We have no other option but to store data on secondary storage devices. But accessing data from magnetic disks is 10,000 to 1,000,000 times slower than accessing it from the main memory. So, B trees are often used to index the data and provide fast access.

Consider a situation in which we have to search an un-indexed and unsorted database that contains n key values. The worst case running time to perform this operation would be $O(n)$. In contrast, if the data in the database is indexed with a B tree, the same search operation will run in $O(\log n)$. For example, searching for a single key on a set of one million keys will at most require 1,000,000 comparisons. But if the same data is indexed with a B tree of order 10, then only 114 comparisons will be

required in the worst case. Hence, we see that indexing large amounts of data can provide significant boost to the performance of search operations. When we use B trees or generalized M-way search trees, the value of m or the order of B trees is often very large. Typically, it varies from 128–512. This means that a single node in the tree can contain 127–511 keys and 128–512 pointers to child nodes.

We take a large value of m mainly because of three reasons:

- Disk access is very slow. We should be able to fetch a large amount of data in one disk access.
- Disk is a block-oriented device. That is, data is organized and retrieved in terms of blocks. So while using a B tree (generalized M-way search tree), a large value of m is used so that one single node of the tree can occupy the entire block. In other words, m represents the maximum number of data items that can be stored in a single block. m is maximized to speed up processing. More the data stored in a block, lesser the time needed to move it into the main memory.
- A large value minimizes the height of the tree. So, search operation becomes really fast.

6.12 B+ TREES

A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key. While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes. The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient. Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory. B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are called index nodes or i-nodes and store index values. This allows us to traverse the tree from the root down to the leaf node that stores the desired data item. Figure 11.9 shows a B+ tree of order 3.

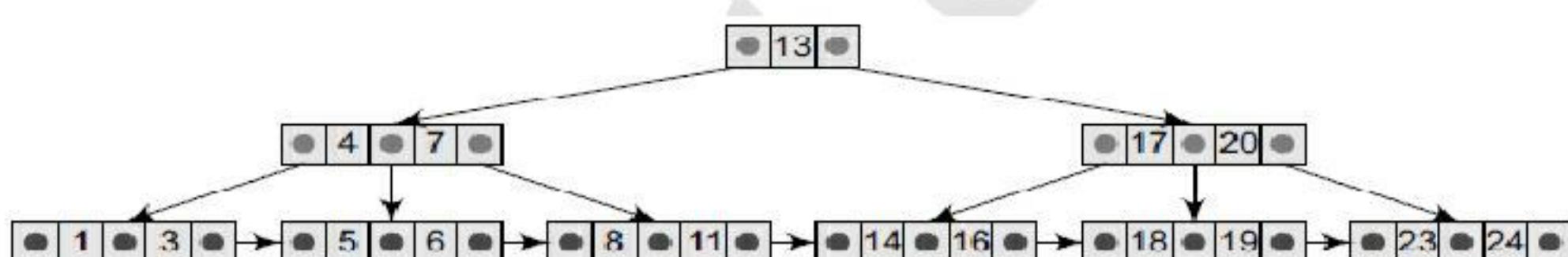


Figure 11.9 B+ tree of order 3

Many database systems are implemented using B+ tree structure because of its simplicity. Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.

A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index. The advantages of B+ trees can be given as follows:

1. Records can be fetched in equal number of disk accesses
2. It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level
3. Height of the tree is less and balanced
4. Supports both random and sequential access to records
5. Keys are used for indexing

Table 11.1 Comparison between B trees and to B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

6.12.1 Inserting a New Element in a B+ Tree

A new element is simply added in the leaf node if there is space for it. But if the data node in the tree where insertion has to be done is full, then that node is split into two nodes. This calls for adding a new index value in the parent index node so that future queries can arbitrate between the two new nodes.

However, adding the new index value in the parent node may cause it, in turn, to split. In fact, all the nodes on the path from a leaf to the root may split when a new value is added to a leaf node. If the root node splits, a new leaf node is created and the tree grows by one level.

The steps to insert a new node in a B+ Tree are summarized in Fig. 11.10.

- Step 1: Insert the new node as the leaf node.
- Step 2: If the leaf node overflows, split the node and copy the middle element to next index node.
- Step 3: If the index node overflows, split that node and move the middle element to next index page.

Figure 11.10 Algorithm for inserting a new node in a B+ tree

Example 11.5 Consider the B+ tree of order 4 given and insert 33 in it.

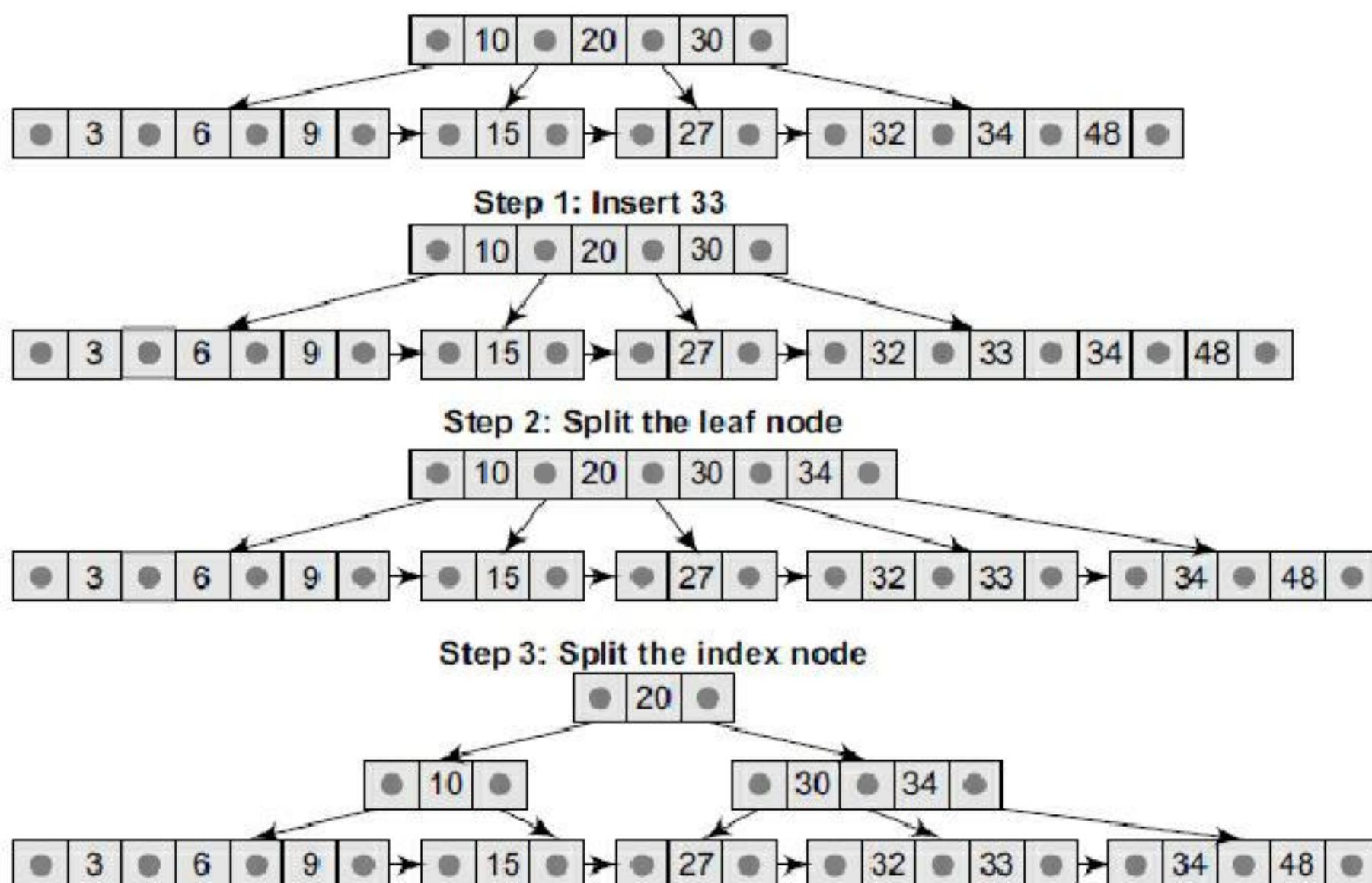


Figure 11.11 Inserting node 33 in the given B+ Tree

6.12.2 Deleting an Element from a B+ Tree

As in B trees, deletion is always done from a leaf node. If deleting a data element leaves that node empty, then the neighbouring nodes are examined and merged with the *underfull* node.

This process calls for the deletion of an index value from the parent index node which, in turn, may cause it to become empty. Similar to the insertion process, deletion may cause a merge-delete wave to run from a leaf node all the way up to the root. This leads to shrinking of the tree by one level. The steps to delete a node from a B+ tree are summarized in Fig. 11.12.

- Step 1: Delete the key and data from the leaves.
- Step 2: If the leaf node underflows, merge that node with the sibling and delete the key in between them.
- Step 3: If the index node underflows, merge that node with the sibling and move down the key in between them.

Figure 11.12 Algorithm for deleting a node from a B+ Tree

Example 11.6 Consider the B+ tree of order 4 given below and delete node 15 from it.

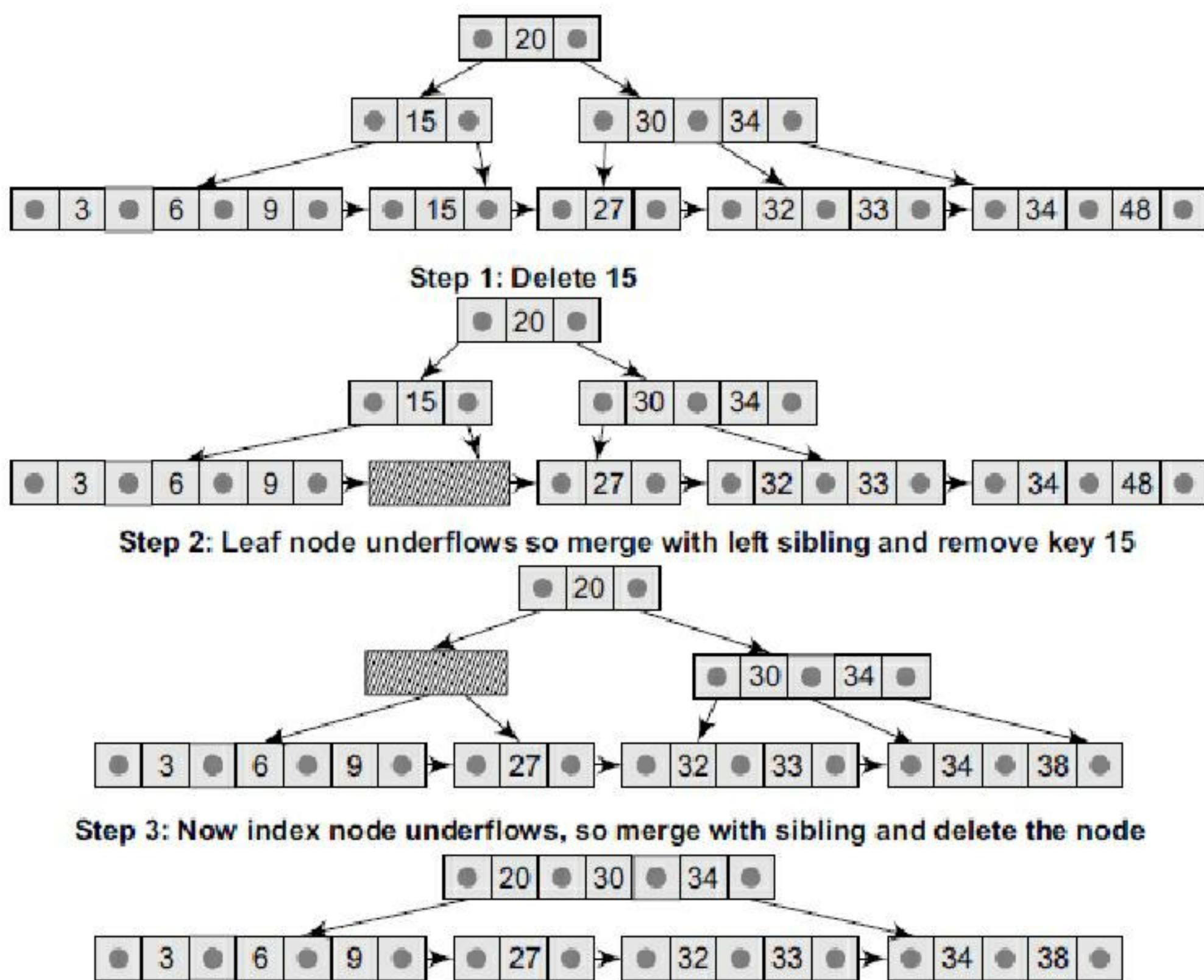


Figure 11.13 Deleting node 15 from the given B+ Tree

6.13 HUFFMAN ALGORITHM

- Huffman coding is an entropy encoding algorithm developed by David A. Huffman that is widely used as a lossless data compression technique. The Huffman coding algorithm uses a variable length code table to encode a source character where the variable-length code table is derived on the basis of the estimated probability of occurrence of the source character.
- The key idea behind Huffman algorithm is that it encodes the most common characters using shorter strings of bits than those used for less common source characters.
- The algorithm works by creating a binary tree of nodes that are stored in an array. A node can be either a leaf node or an internal node. Initially, all the nodes in the tree are at the leaf level and store the source character and its frequency of occurrence (also known as weight).
 - Suppose there are 'n' weights w_1, w_2, \dots, w_n .
 - Take 2 minimum weights among the 'n' given inputs. Suppose w_1, w_2 are first two minimum, then the subtree will be
 - Now the remaining weights will be $w_3, w_1 + w_2, \dots, w_n$.
 - Create all subtree till the last weight. Thus Huffman algorithms construct the tree from the bottom up approach rather than top down approach.

➤ **Technique:**

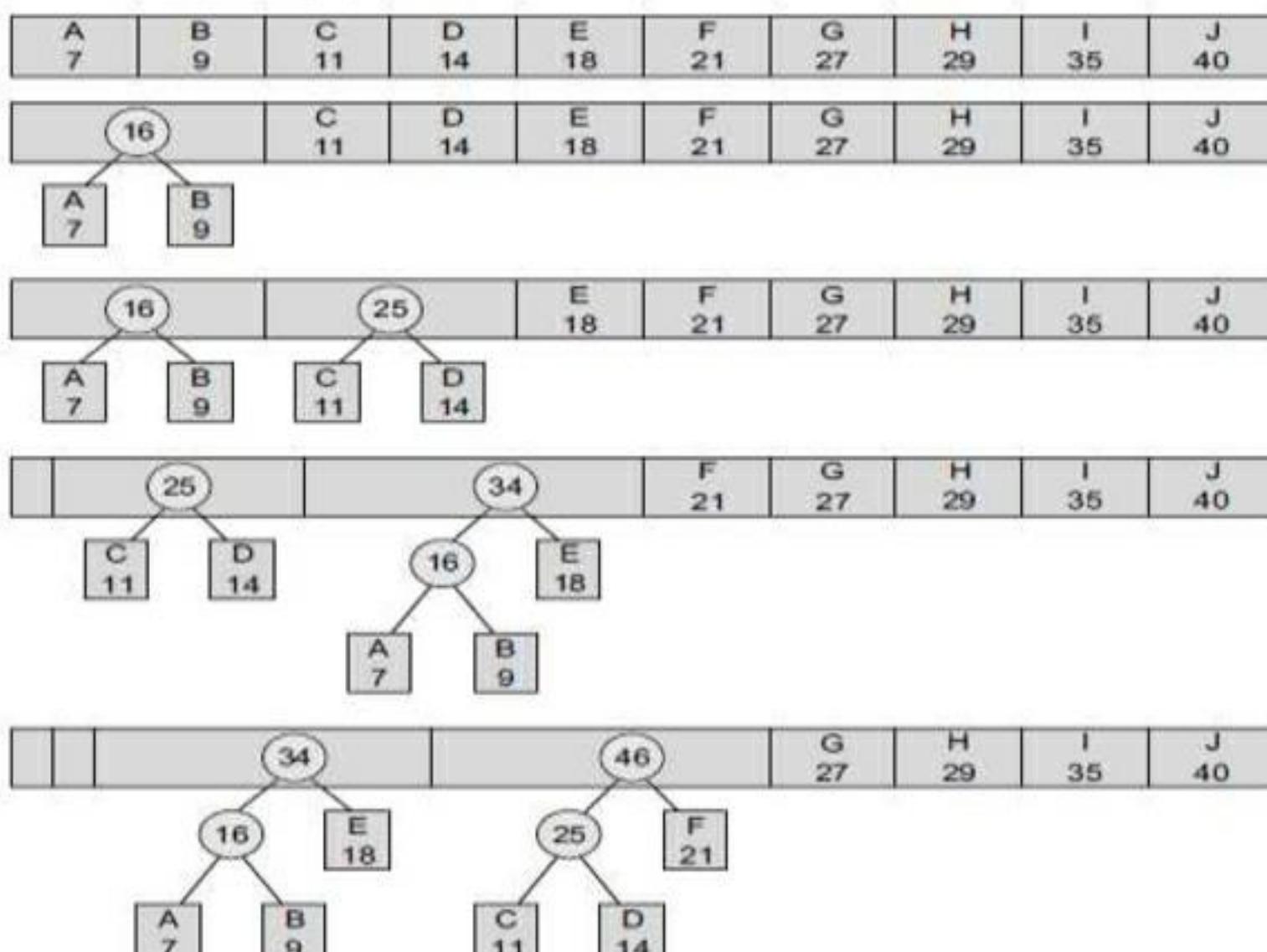
Given n nodes and their weights, the Huffman algorithm is used to find a tree with a minimum weighted path length. The process essentially begins by creating a new node whose children are the two nodes with the smallest weight, such that the new node's weight is equal to the sum of the children's weight. That is, the two nodes are merged into one node. This process is repeated until the tree has only one node. Such a tree with only one node is known as the Huffman tree.

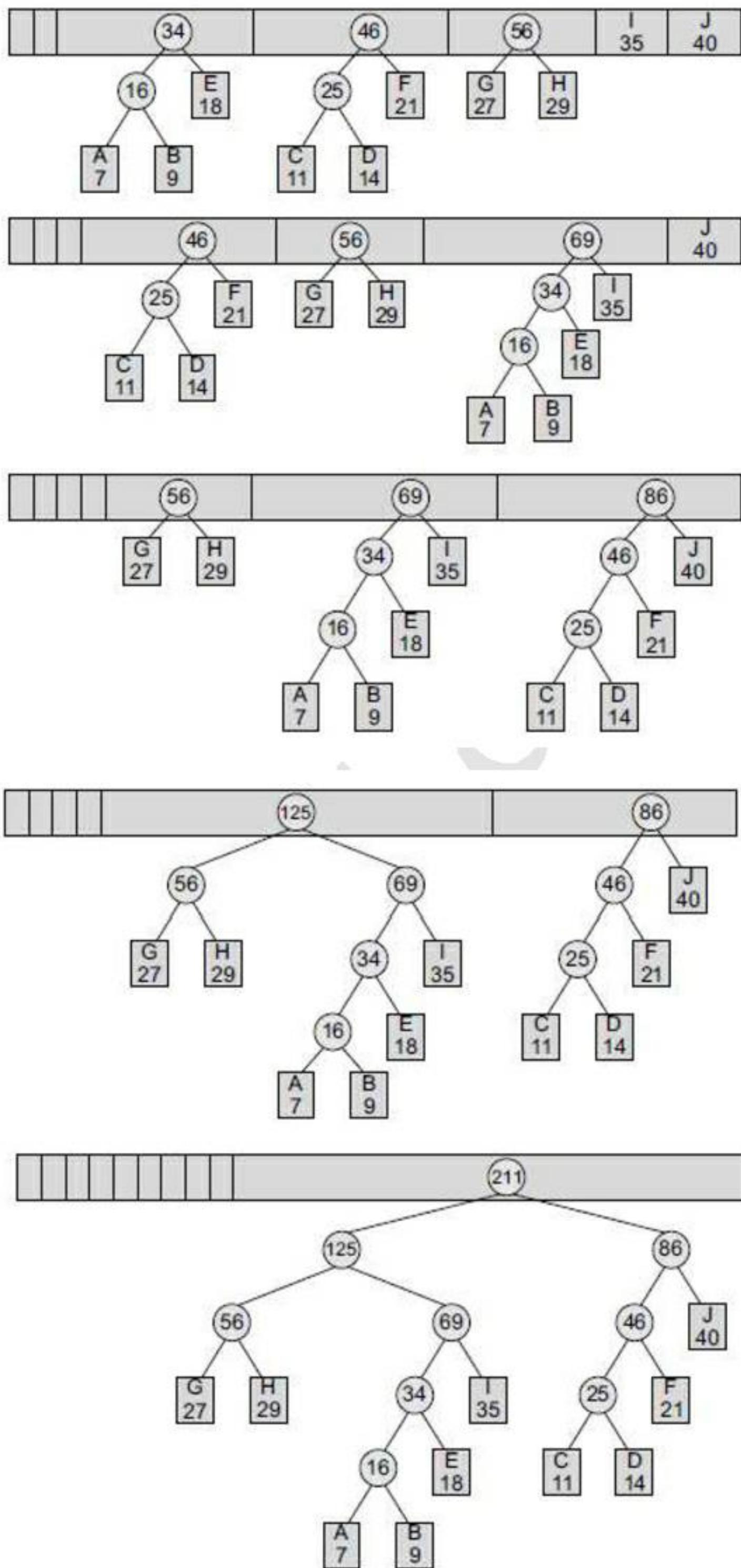
The Huffman algorithm can be implemented using a priority queue in which all the nodes are placed in such a way that the node with the lowest weight is given the highest priority. The algorithm is shown in Figure A.

- Step 1: Create a leaf node for each character. Add the character and its weight or frequency of occurrence to the priority queue.
- Step 2: Repeat Steps 3 to 5 while the total number of nodes in the queue is greater than 1.
- Step 3: Remove two nodes that have the lowest weight (or highest priority).
- Step 4: Create a new internal node by merging these two nodes as children and with weight equal to the sum of the two nodes' weights.
- Step 5: Add the newly created node to the queue.

Figure A: Huffman algorithm

Example: Create a Huffman tree with the following nodes arranged in a priority queue.





4 Huffman Coding

In the Huffman tree, circles contain the cumulative weights of their child nodes. Every left branch is coded with 0 and every right branch is coded with 1. So, the characters A, E, R, W, X, Y, and Z are coded as shown in Table

Table Characters with their codes

Character	Code
A	00
E	01
R	11
W	1010
X	1000
Y	1001
Z	1011

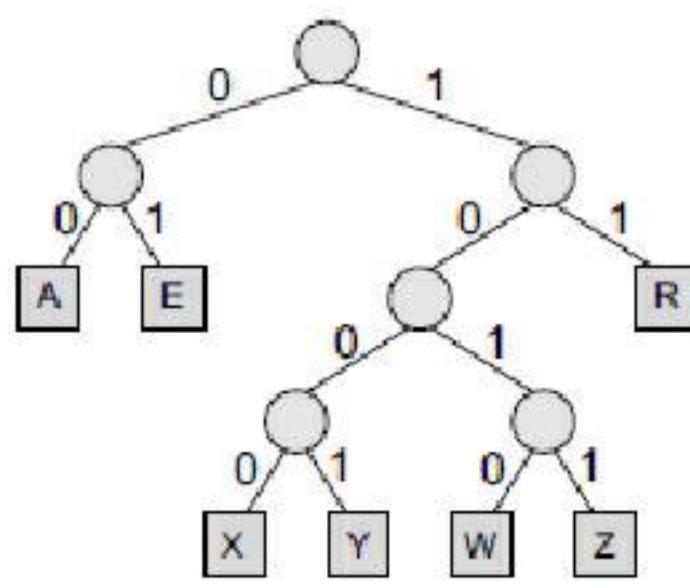


Figure Huffman tree

Thus, we see that frequent characters have a shorter code and infrequent characters have a longer code.

Example:

Huffman's Algorithm with Example

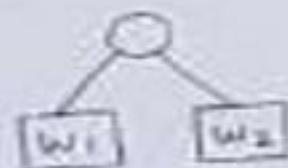
Algorithm: Suppose w_1 and w_2 are two minimum weights among the n given weights w_1, w_2, \dots, w_n . Find a tree T' which gives a solution for the $(n-1)$ weights

$w_1 + w_2, w_3, w_4, \dots, w_n$

Then, in the tree T' , replace the external node

$w_1 + w_2$

by the subtree

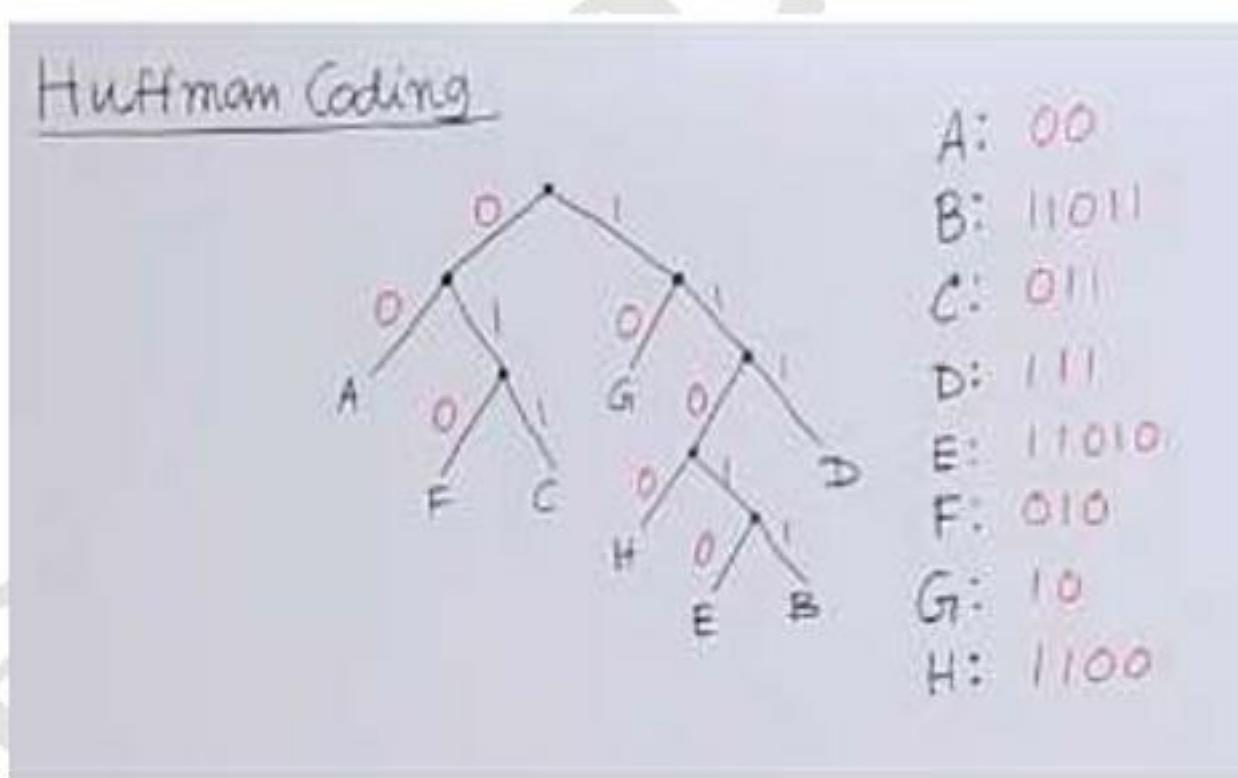
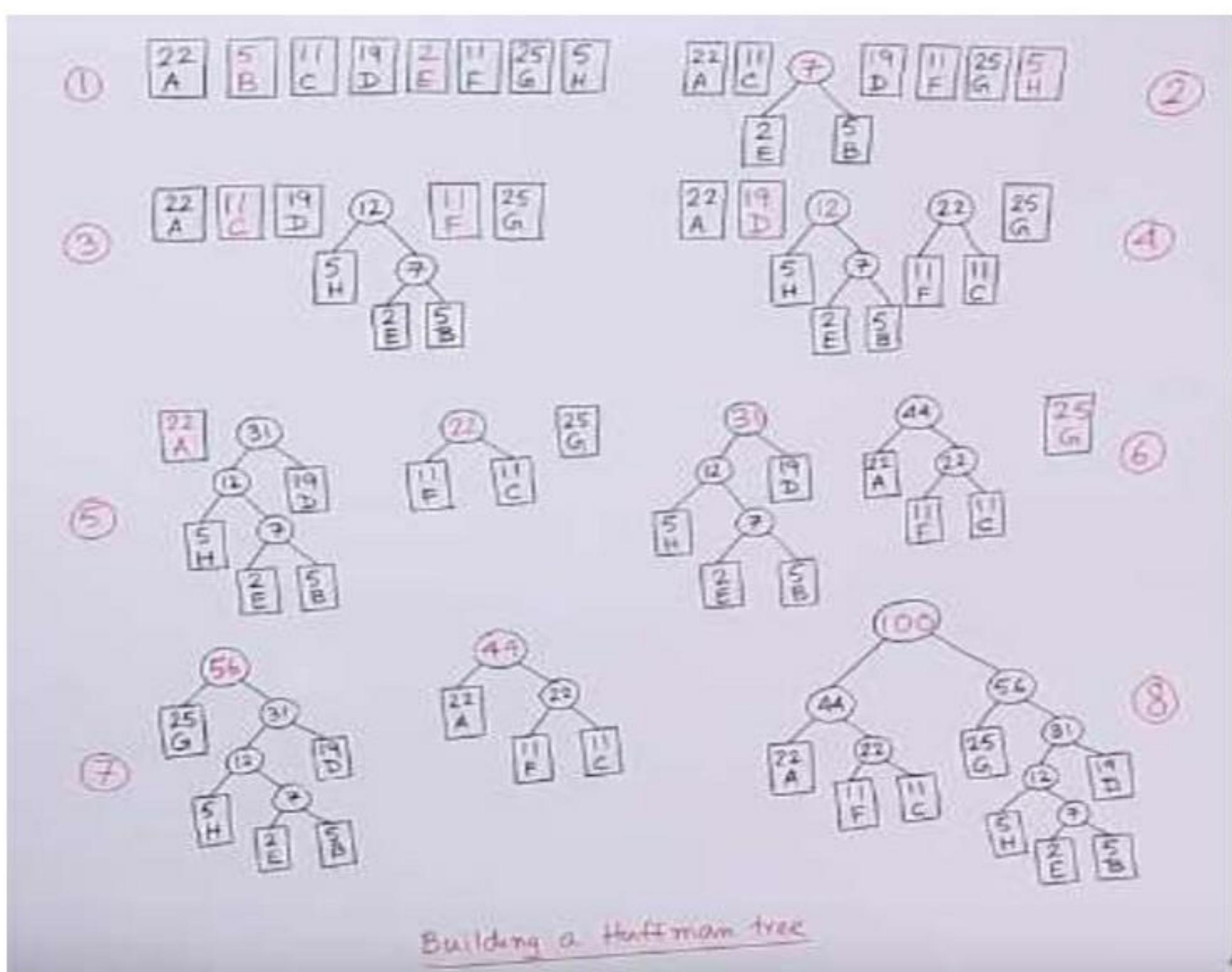


The new 2-tree T is the desired solution.

Example: Suppose A, B, C, D, E, F, G and H are 8 data items, and they are assigned weights as follows:

Data items: A B C D E F G H
Weight : 22 5 11 19 2 11 25 5

Construct tree T with minimum-weighted path length using the above data and Huffman's algorithm.



6.14 Game Tree

A game tree isn't a special new data structure—it's a name for any regular tree that maps how a discrete game is played.

Lets take an example of game called Rocks. In this game, we have different piles of rocks, with one or more rocks in each pile. The game has two players, who take turns taking one or more rocks from a single pile until one pile is left. When one pile is left, our goal is to force our opponent to remove the last rock. The person who removes the last rock loses.



Figure A: Simple game of rock with 2 piles

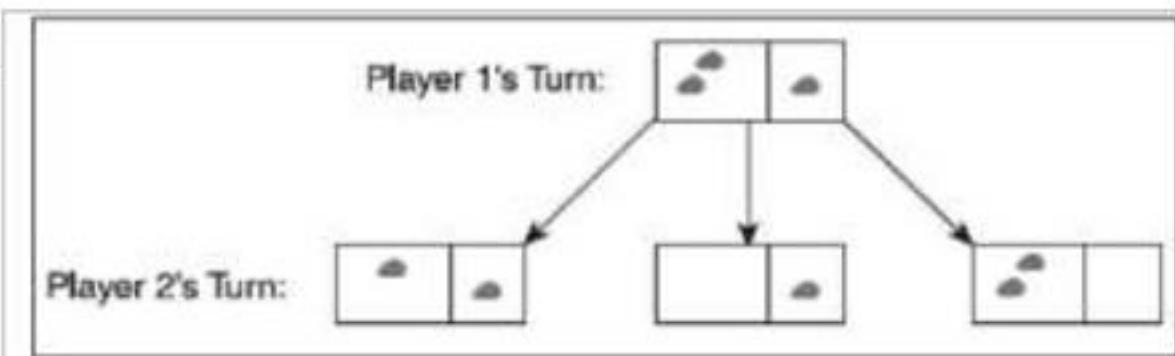


Figure B: First 2 levels of game tree

In the figure, there are two piles. The first pile has two rocks, and the second pile has only one rock. The first player has three choices:

- Remove one rock from pile 1.
- Remove two rocks from pile 1.
- Remove one rock from pile 2.

We can start the game off with one of those three moves. We can create a simple game tree to represent these moves, as shown in Figure B. After Player 1 has moved, it is now Player 2's turn. Player 2's choice of a move is limited to the current state of the game, however. In the leftmost state of Figure A, Player 2 has two choices: He can remove one rock from pile 1 or one rock from pile 2. His choice for the middle state is even less useful He can only remove one rock from pile 2. Of course, because this is the last rock, Player 2 has lost the game. On the right state, Player 2 has two options again: He can remove one or two rocks from pile 2.

Figure C shows the game tree for all five of these moves and goes down one more level to show the complete game tree.

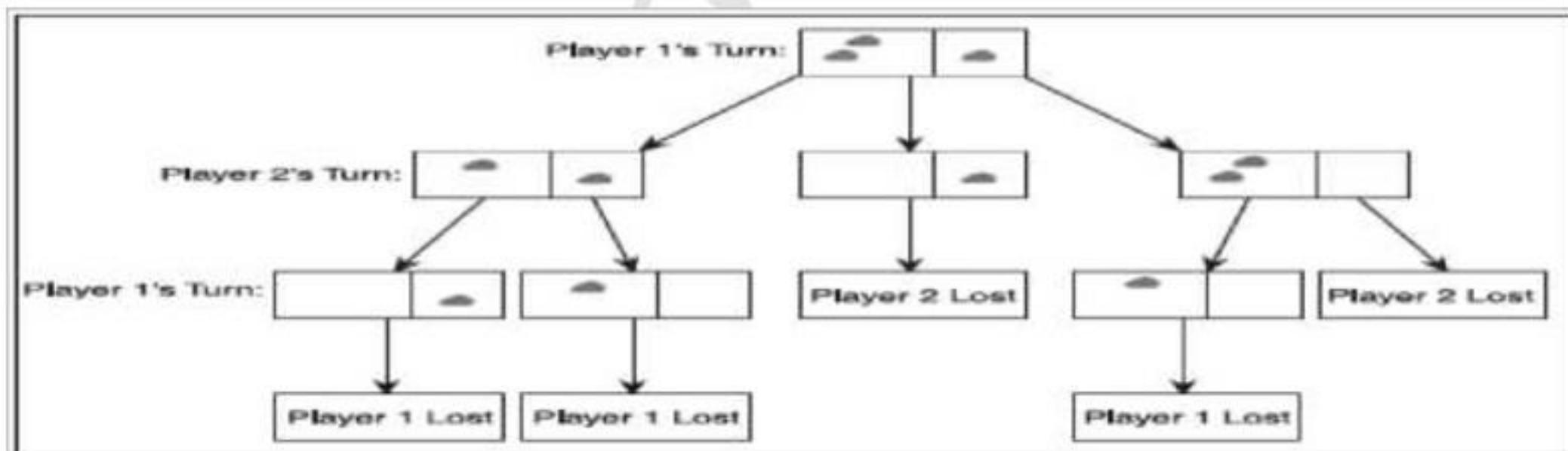


Figure C: A Complete Game tree

The game is entirely complete by the time the fourth level is reached. The game can have up to three moves because there were only three rocks. We can also tell from the tree that there are five total outcomes from the game because there are five leaf nodes. The game always ends on a leaf node because there are no more moves that can be made. So what can we tell about the game tree that we couldn't easily tell about the initial game setup?

For Player 1, the obvious first move is the second one, removing the two stones from pile 1. By doing that, Player 2 forced to be lost the game, because he has no other option and cannot possibly win. Another thing that we would notice if Player 1 plays the leftmost move, removing one rock from pile 1, is a death sentence. If Player 1 makes that move, then he have given Player 2 a free win, because no matter what move he makes, there is no chance for Player 1 to win in that branch. If Player 1 takes the third route on the opening move, then Player 2 decides the outcome of the game. If Player 2 removes both rocks in pile 1 (a very stupid move), he loses. If he only removes one rock, then he forces Player 1 to remove the last one, and he loses.

6.15 Applications of Tree Data Structure

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because we want to store information that naturally forms a hierarchy. For example, the file system on a computer:
2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Selfbalancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log n)$ for search.
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log n)$ for insertion/deletion.
4. **Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes** as nodes are linked using pointers.

In Short, following are the common uses of tree data structure.

1. Manipulate hierarchical data.
2. Make information easy to search.
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

Other Applications

1. *Binary Search Tree*: Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
2. *Hash Trees* - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
3. *Heaps* - Used in heap-sort; fast implementations of Dijkstra's algorithm; and in implementing efficient priority-queues, which are used in scheduling processes in many operating systems, Quality-of-Service in routers, and A* (path-finding algorithm used in AI applications, including video games).
4. *Huffman Coding Tree* - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
5. *Syntax Tree* - Constructed by compilers and (implicitly) calculators to parse expressions.
6. *Treap* - Randomized data structure used in wireless networking and memory allocation.
7. *T-tree* - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so