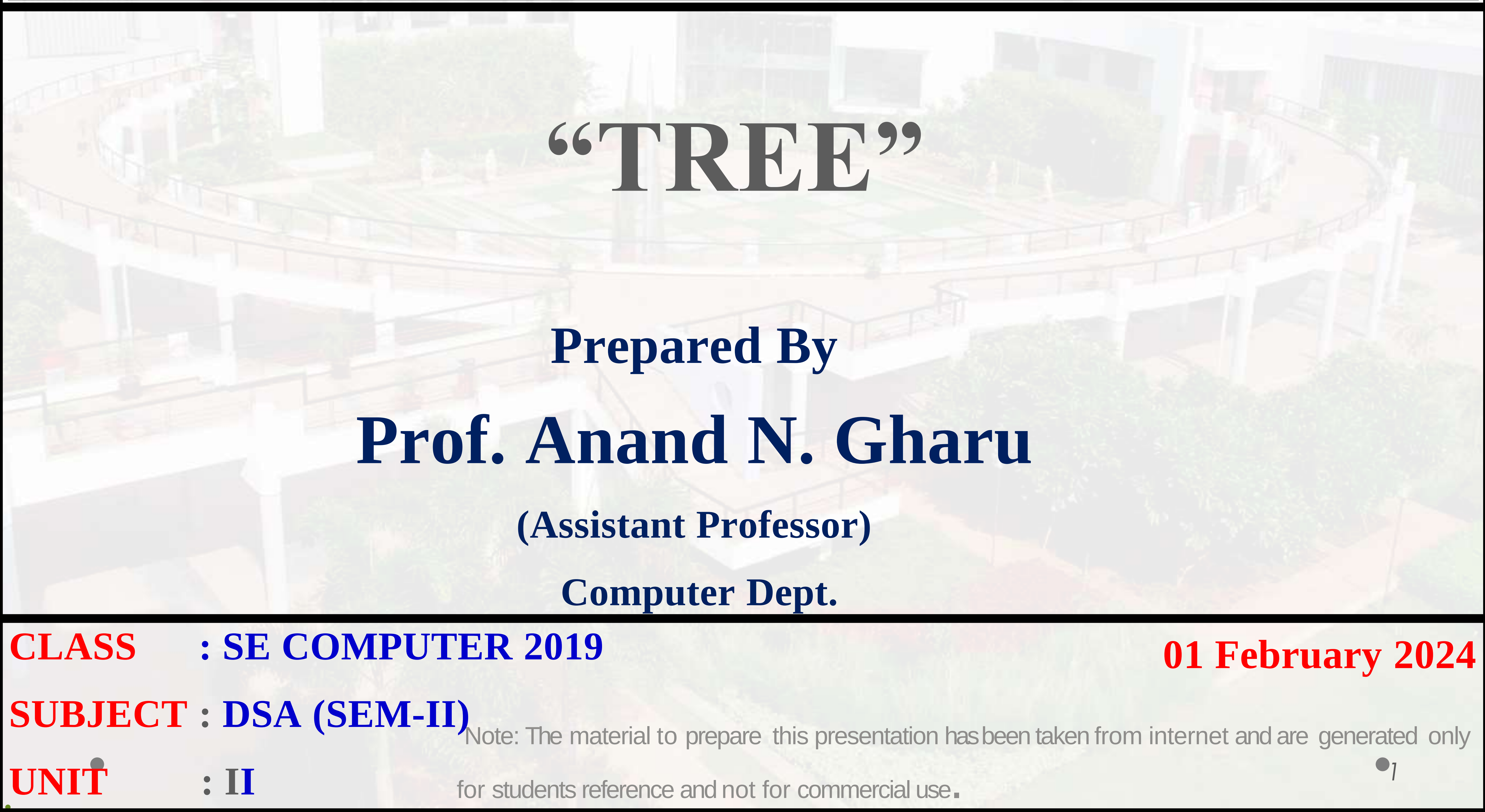
MET's Institute of Engineering

Bhujbal Knowledge City, Adgaon, Nashik.

Department of Computer Engineering



"TREE"

Prepared By

Prof. Anand N. Gharu

(Assistant Professor)

Computer Dept.

CLASS : SE COMPUTER 2019

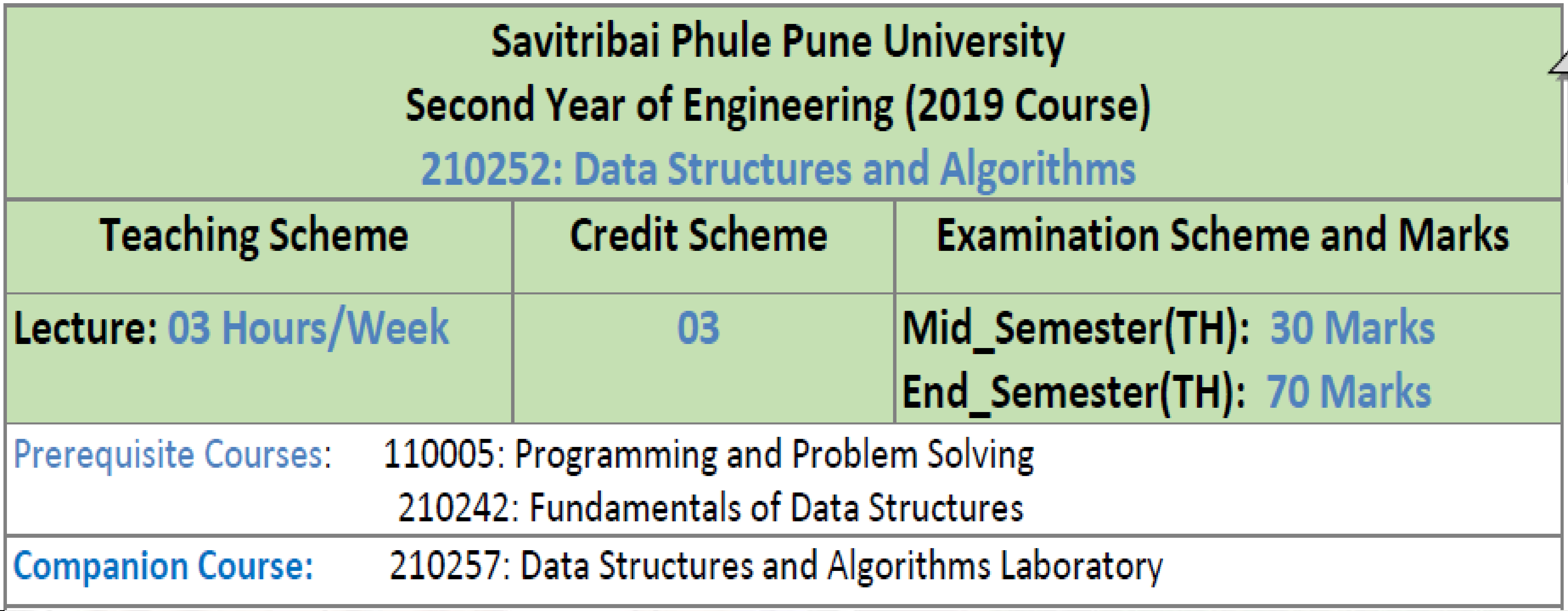
SUBJECT : DSA (SEM-II)

UNIT : II

01 February 2024

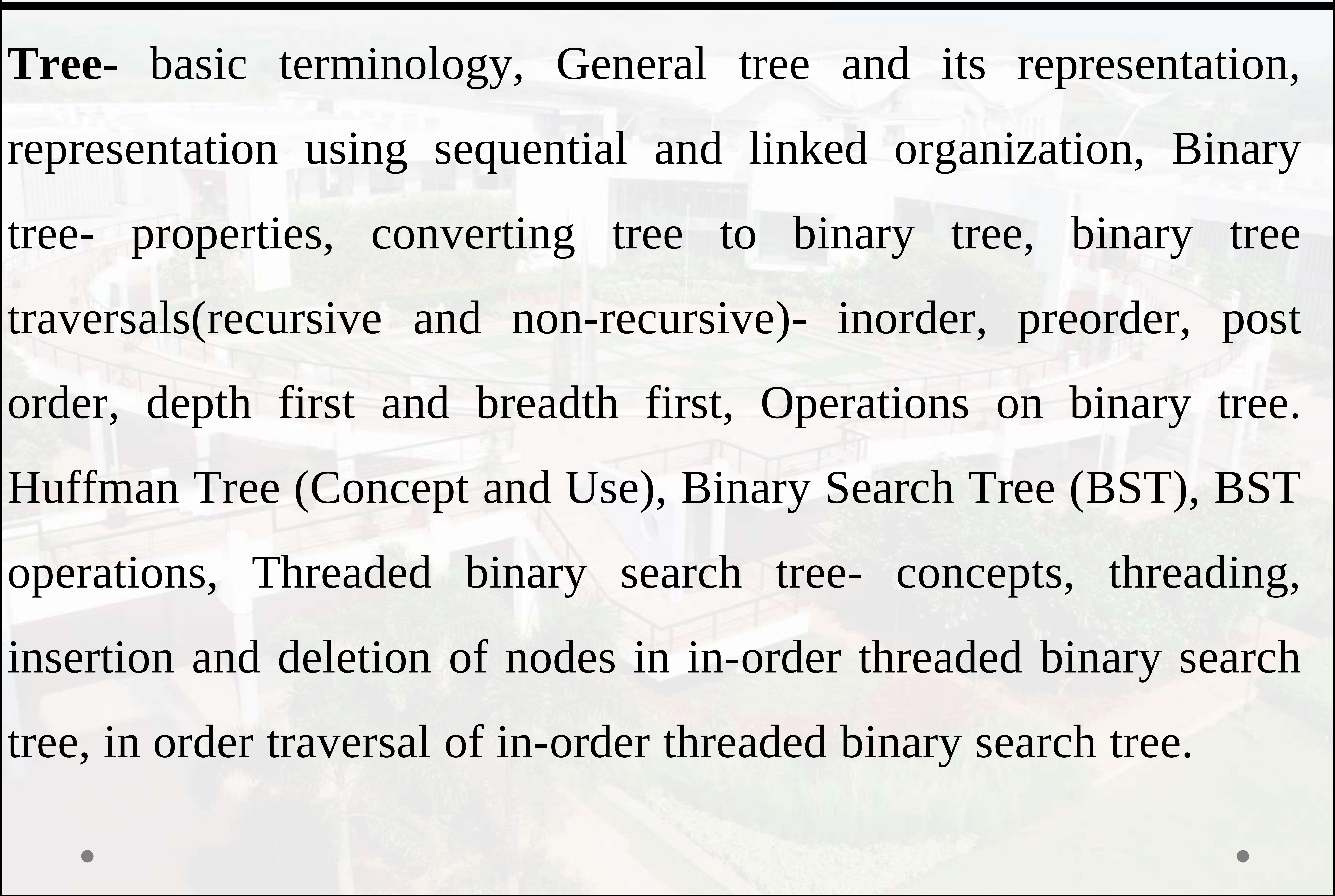
Note: The material to prepare this presentation has been taken from internet and are generated only for students reference and not for commercial use.

SYLLABUS



Savitribai Phule Pune University Second Year of Engineering (2019 Course) 210252: Data Structures and Algorithms		
Teaching Scheme	Credit Scheme	Examination Scheme and Marks
Lecture: 03 Hours/Week	03	Mid_Semester(TH): 30 Marks End_Semester(TH): 70 Marks
Prerequisite Courses:	110005: Programming and Problem Solving 210242: Fundamentals of Data Structures	
Companion Course:	210257: Data Structures and Algorithms Laboratory	

SYLLABUS



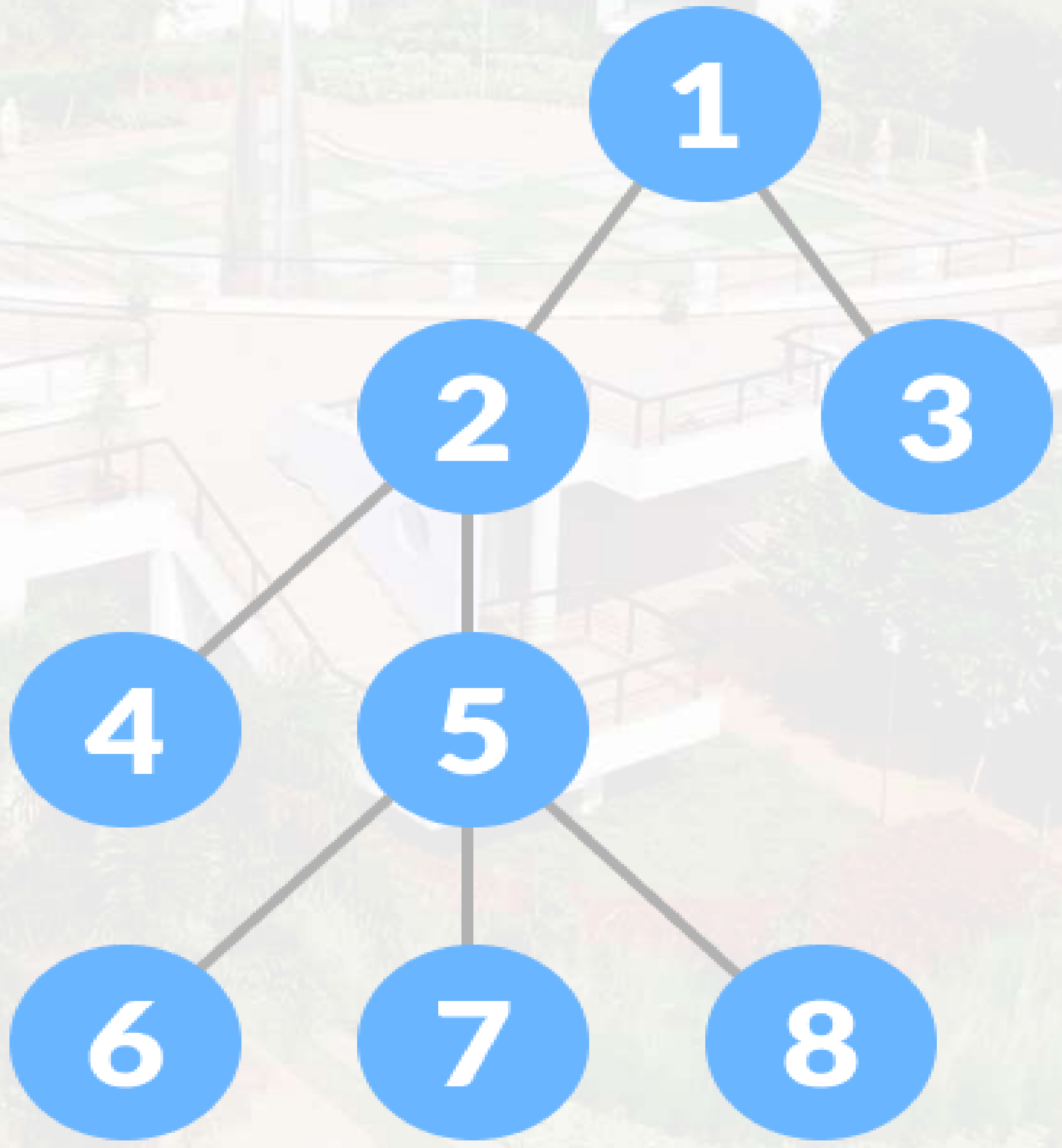
Tree- basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, binary tree traversals(recursive and non-recursive)- inorder, preorder, post order, depth first and breadth first, Operations on binary tree. Huffman Tree (Concept and Use), Binary Search Tree (BST), BST operations, Threaded binary search tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary search tree, in order traversal of in-order threaded binary search tree.

UNIT-I I

TREE

TERMINOLOGIES IN TREE

- "A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges."



Why Tree Data Structure?

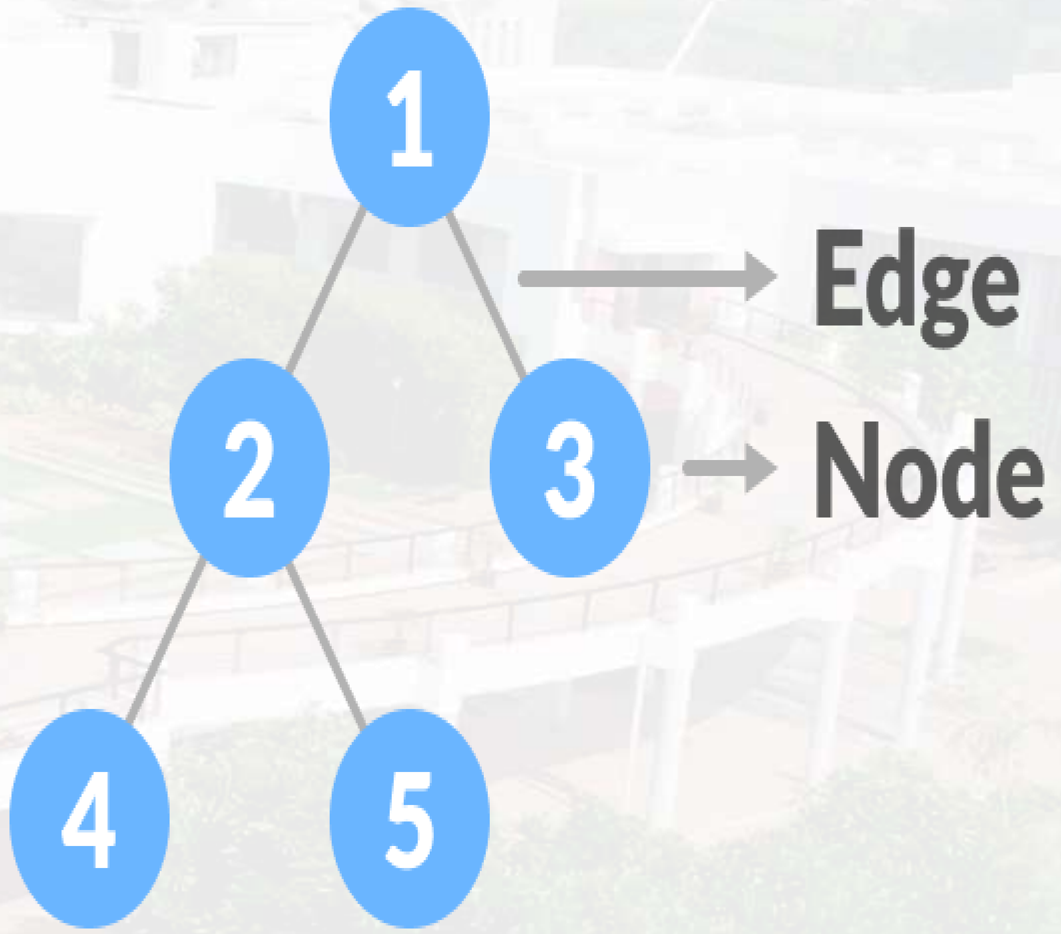
Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Terminologies of Tree

- **Root :**

It is the topmost node of a tree.



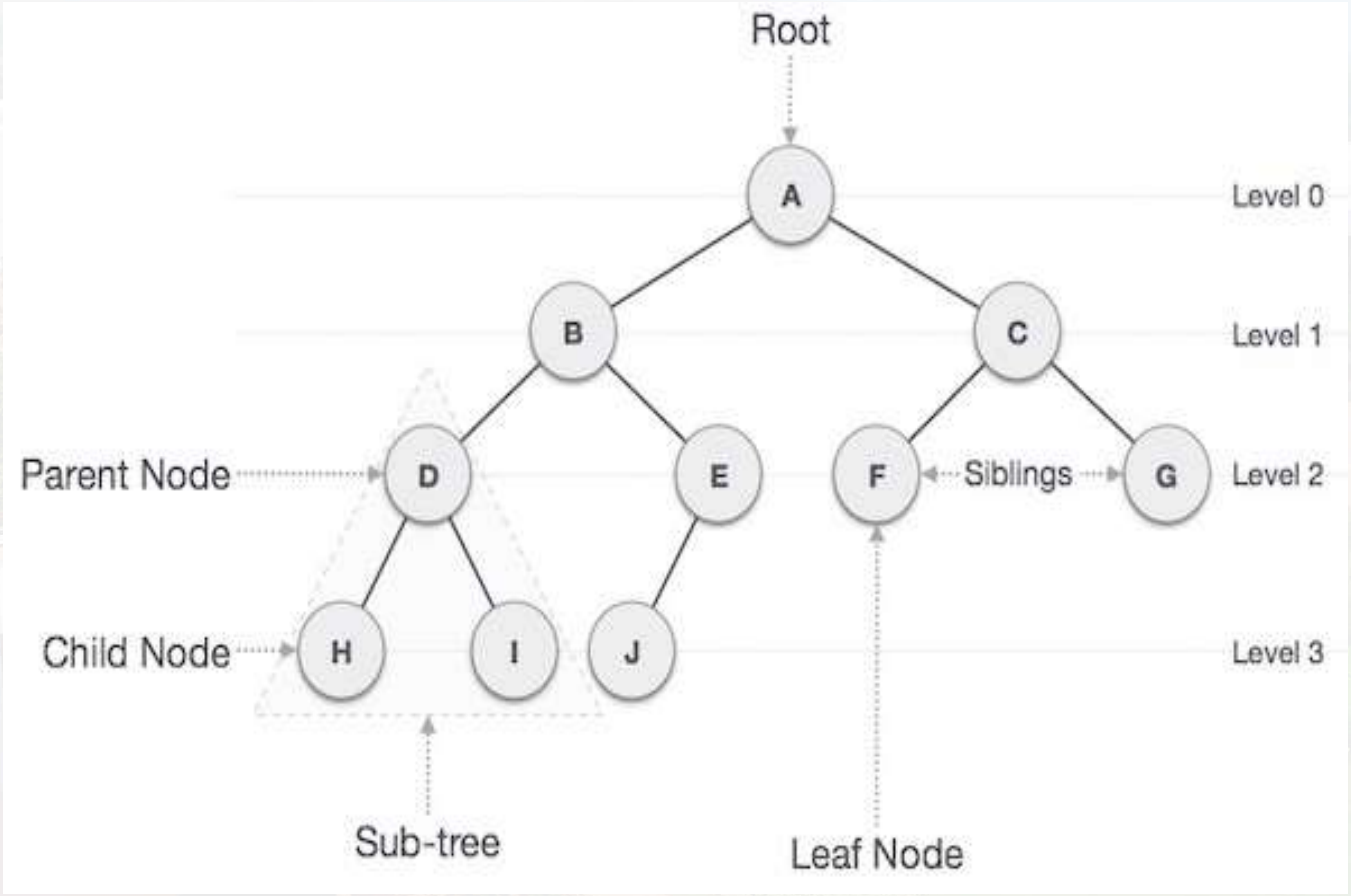
- **Edge :**

It is the link between any two nodes.

- **Node :**

A node is an entity that contains a key or value and pointers to its child nodes.

Terminologies of Tree



Terminologies of Tree

- **Child node:** If the node is a descendant of any node, then the node is known as a child node.

- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.

- **Sibling:** The nodes that have the same parent are known as siblings.

Terminologies of Tree

- **Leaf** – The node which does not have any child node is called the leaf node.

- **Subtree** – Subtree represents the descendants of a node.

- **Path** – Path refers to the sequence of nodes along the edges of a tree

Terminologies of Tree

- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Terminologies of Tree

Leaf/External node: Node with no children.

Internal node:

The node having at least a child node is called an **internal node**

- **Height of a Node :**

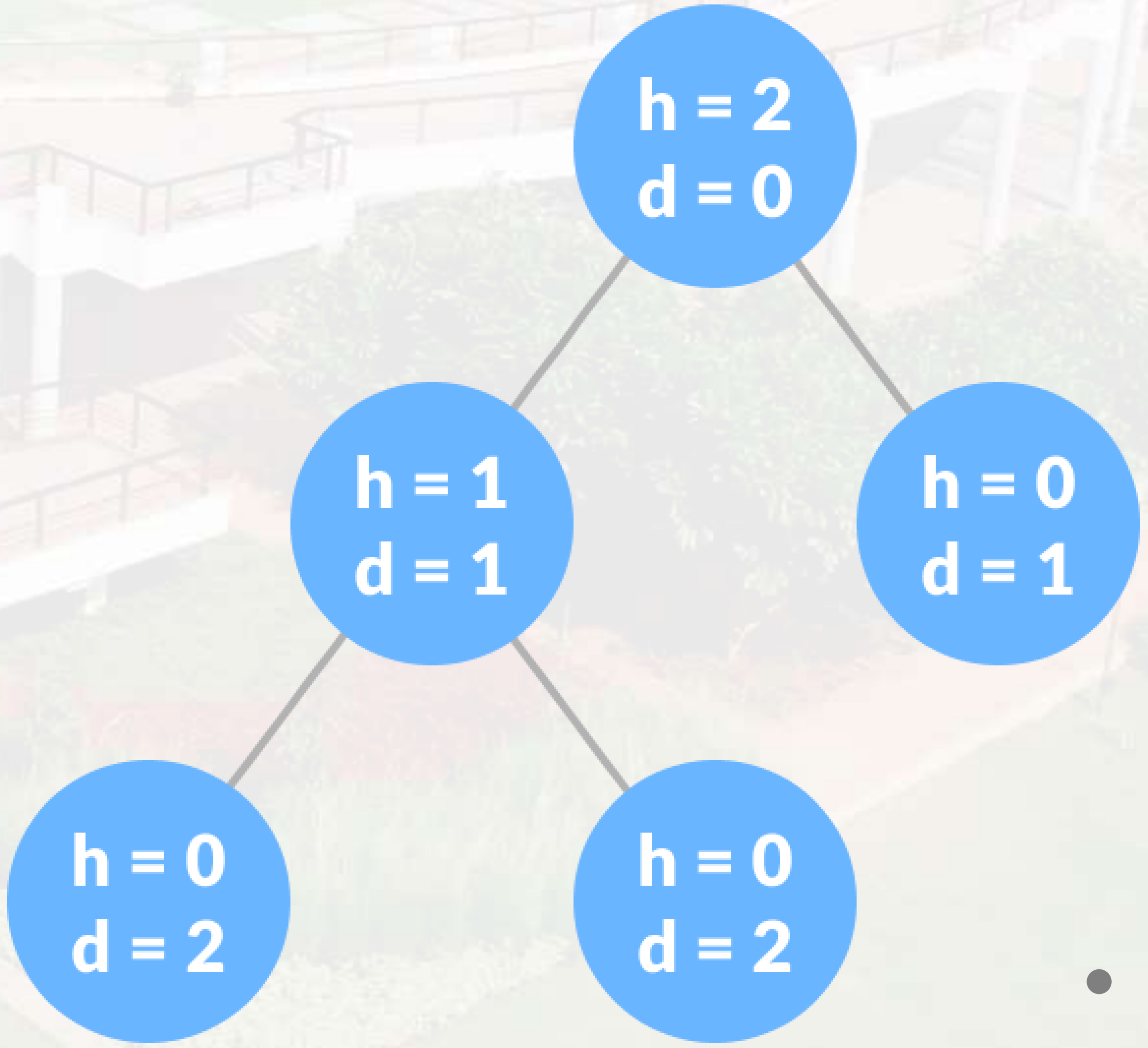
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Terminologies of Tree

The depth of a node is the number of edges from the root to the node.

- **Height of a Tree :**

The height of a Tree is the height of the root node or the depth of the deepest node.



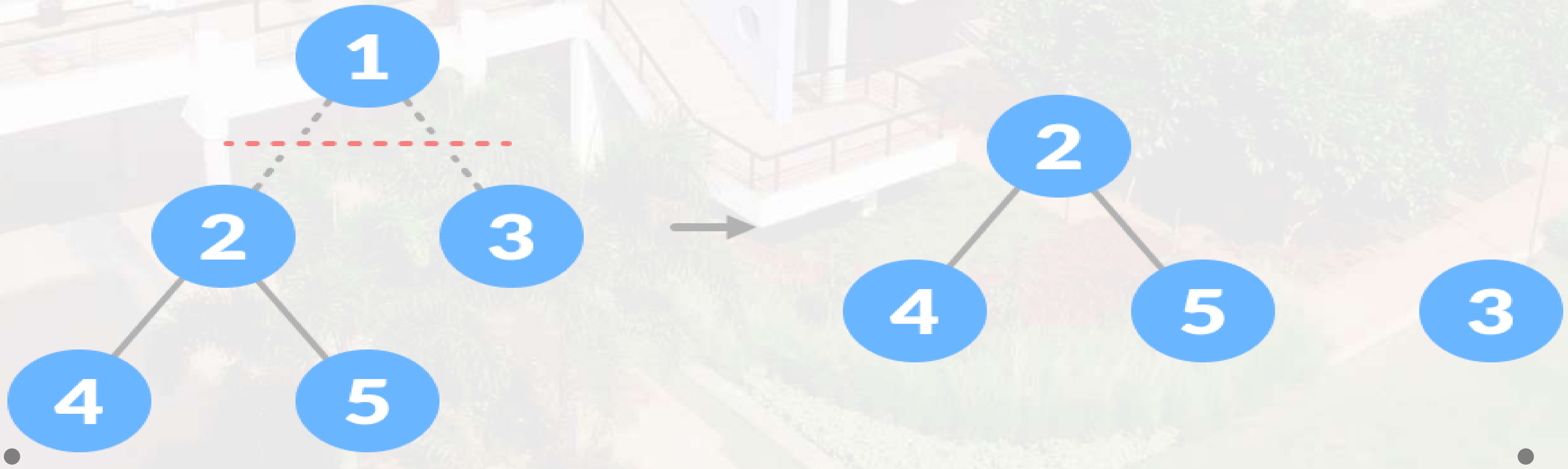
Terminologies of Tree

- **Degree of a Node :**

The degree of a node is the total number of branches of that node.

- **Forest :**

A collection of disjoint trees is called a forest.
You can create a forest by cutting the root of a tree.



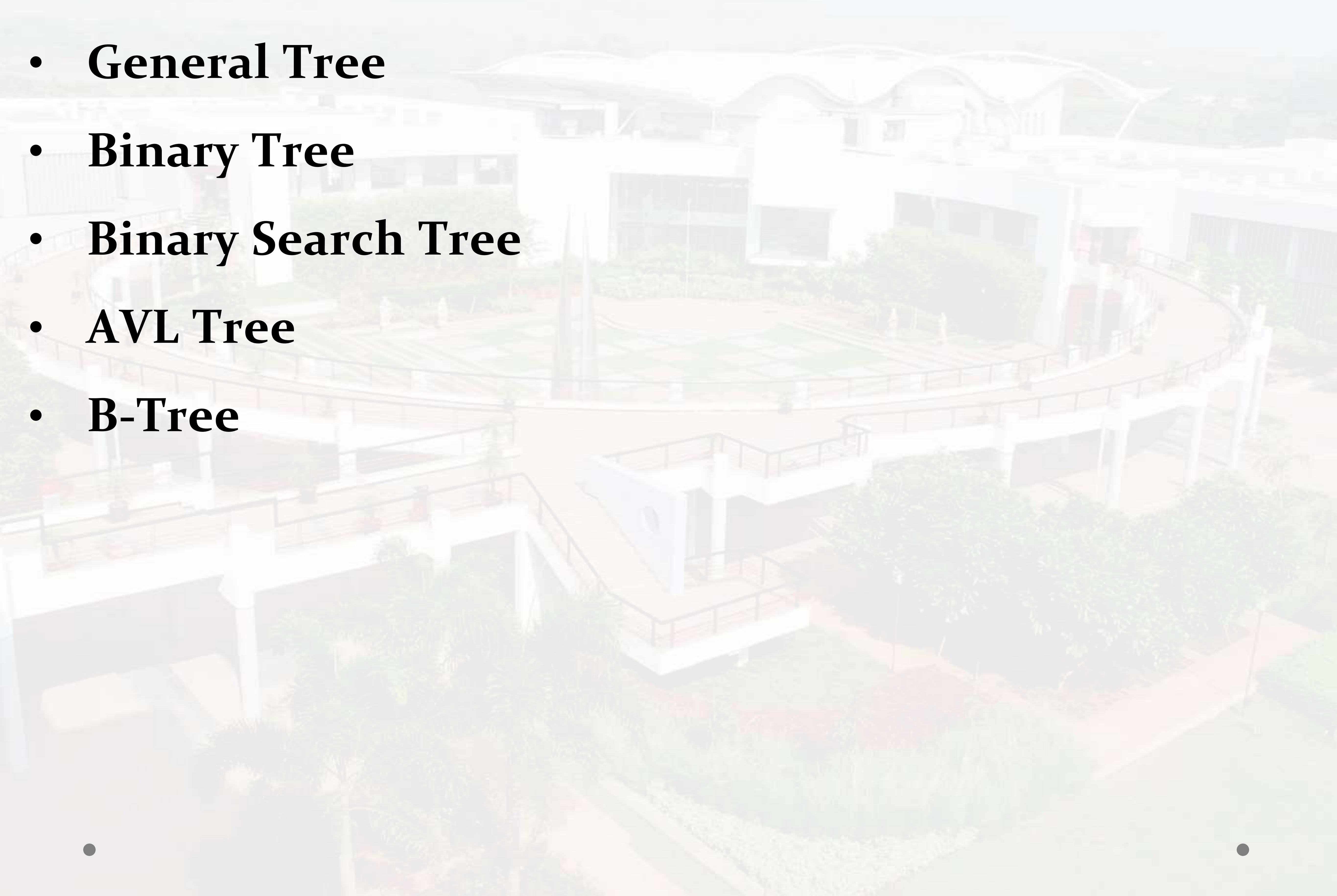
Applications of Tree

- Binary Search Tree (BST) is used to check whether elements present or not.
- Heap is a type of tree that is used to heap sort.
- Tries are the modified version of the tree used in modem routing to information of the router.
- The widespread database uses B-tree.
- Compilers use syntax trees to check every syntax of a program.

Advantages of Tree

1. Trees reflect structural relationships in the data.
2. Trees are used to represent hierarchies.
3. Trees provide an efficient insertion and searching.
4. Trees are very flexible data, allowing to move subtrees around with minimum effort.

Types of Tree

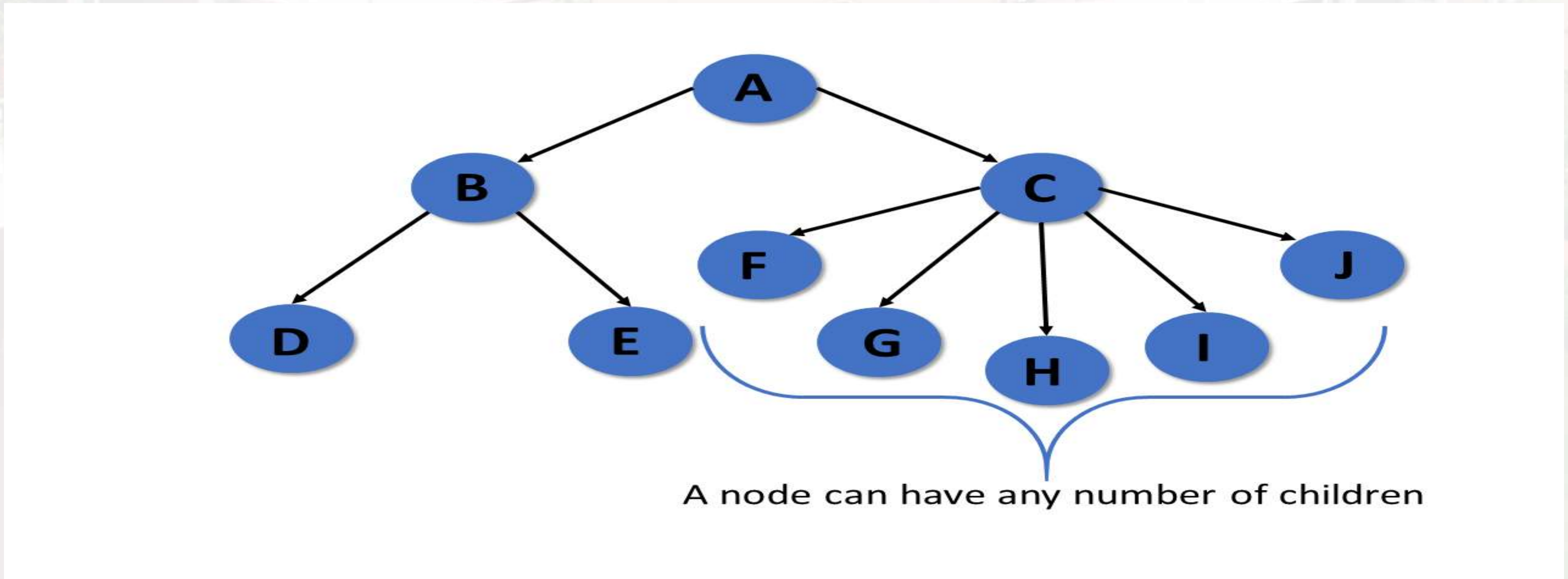


- General Tree
- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree

General Tree

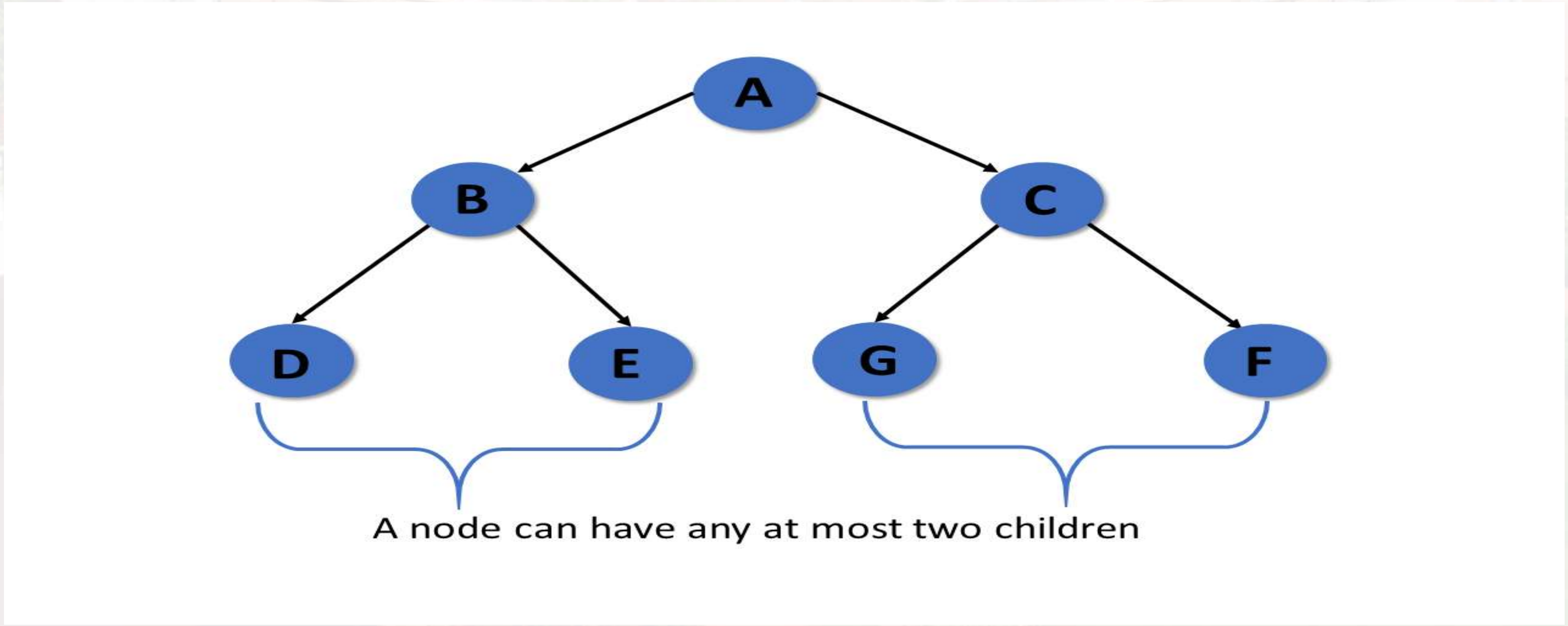
- **General Tree**
- The general tree is the type of tree where there are no constraints on the hierarchical structure.

- **Properties**
- The general tree follows all properties of the tree data structure.
- A node can have any number of nodes.



Binary Tree

- **A binary tree has the following properties:**
- **Properties**
- Follows all properties of the tree data structure.
- Binary trees can have at most two child nodes.
- These two children are called the left child and the right child.



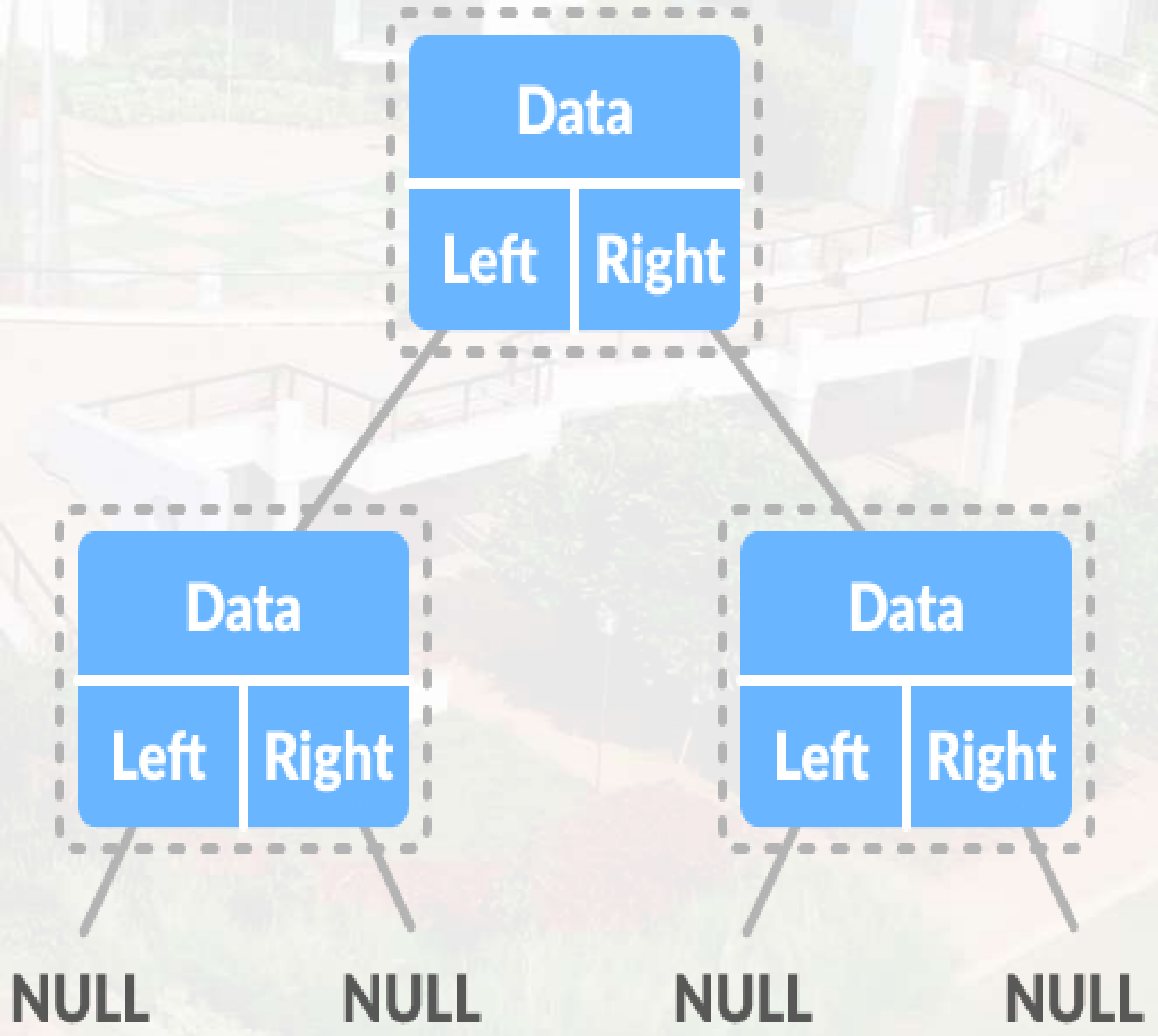
Binary Tree

- **A binary tree has the following properties:**
- At each level of i, the maximum number of nodes is 2i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is (20 + 21 + 22+….2h) = 2h+1 -1.
- The minimum number of nodes possible at height h is equal to h+1.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum

Binary Tree Representation

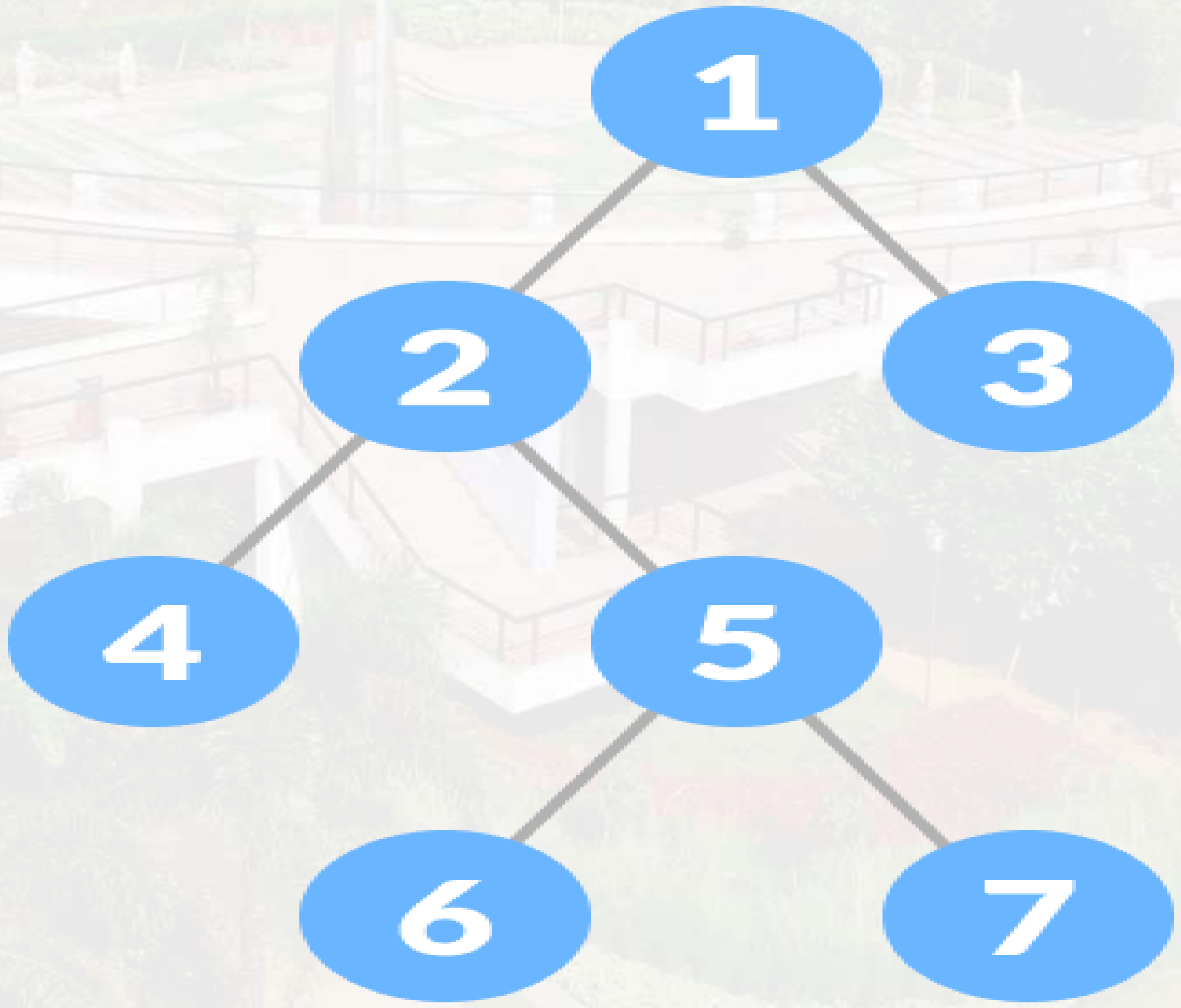
A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

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



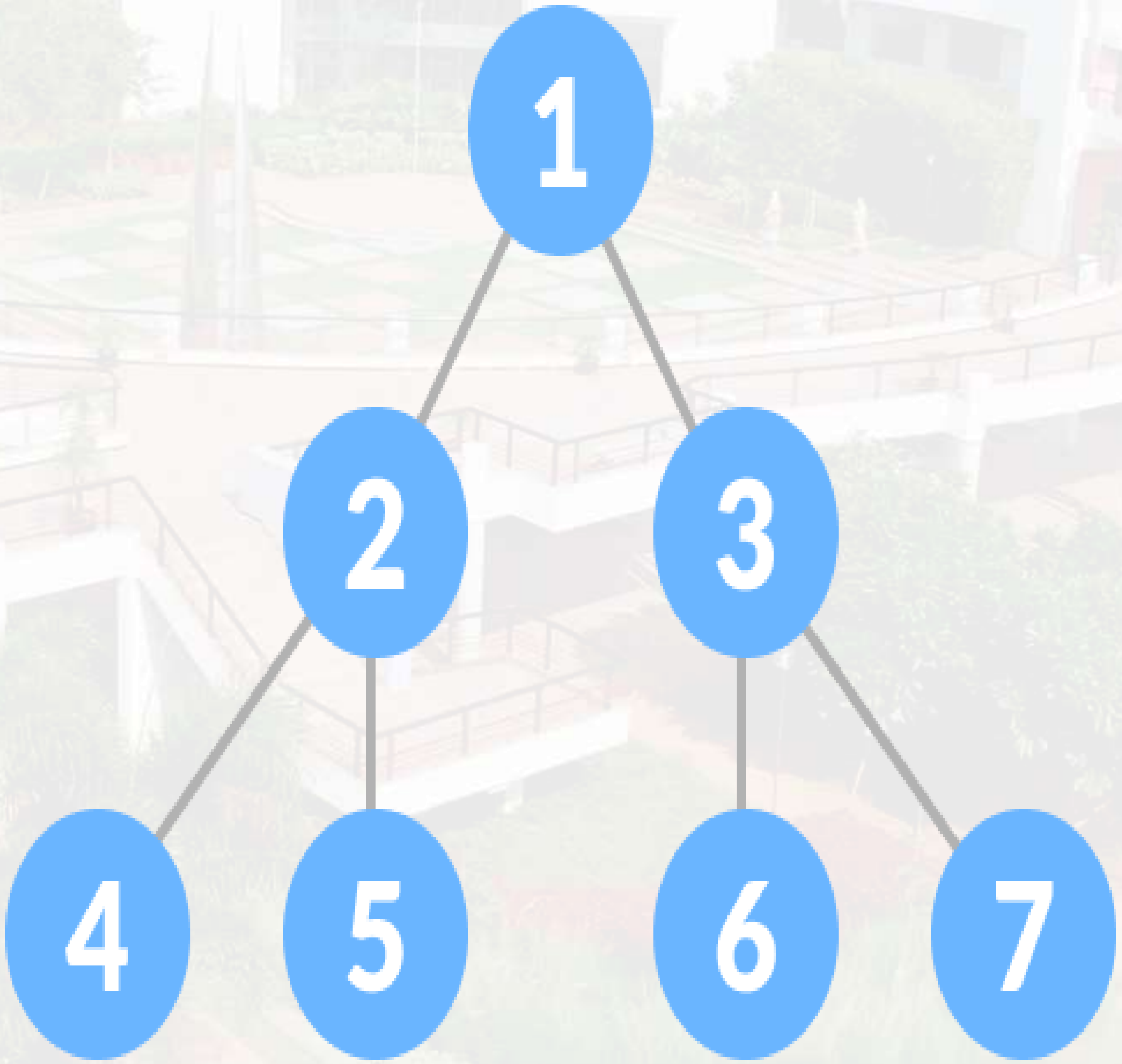
Types of Binary Tree

- **Full Binary tree:** It is a special type of binary tree. In this tree data structure, every parent node or an internal node has either two children or no child nodes.
- .



Types of Binary Tree

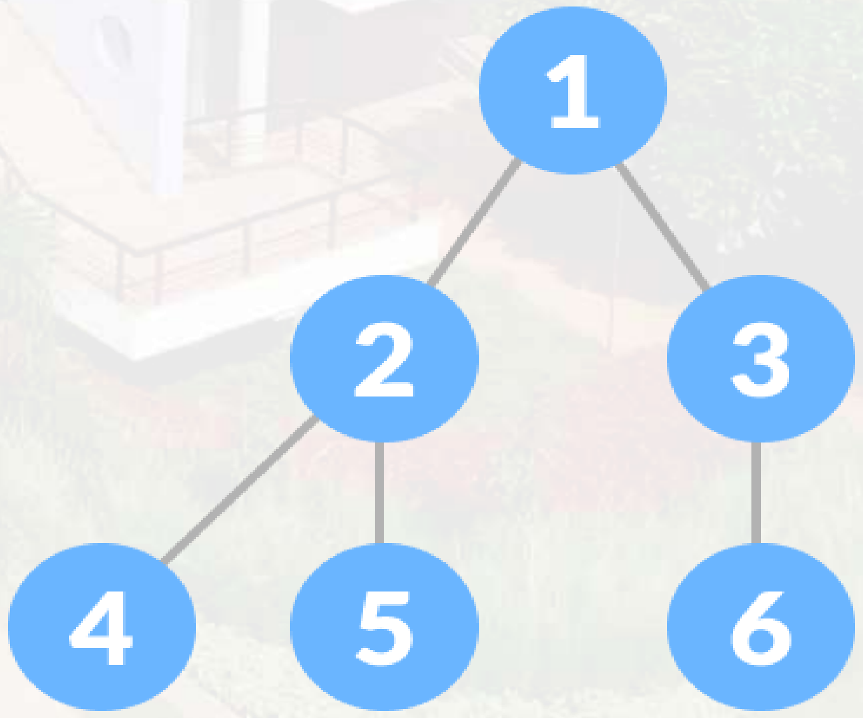
- **Perfect binary tree:** In this type of tree data structure, every internal node has exactly two child nodes and all the leaf nodes are at the same level.
- .



Binary Tree

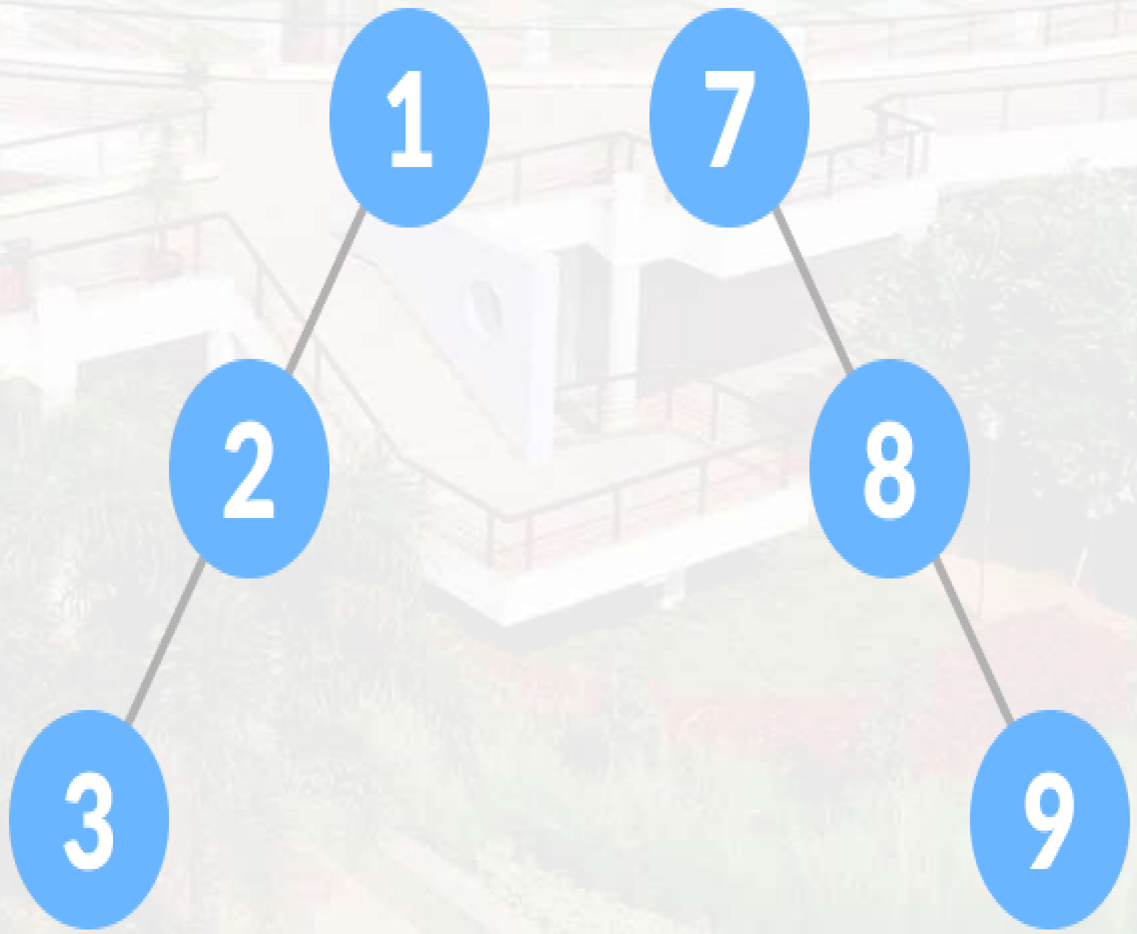
- **Complete binary tree:** It resembles that of the full binary tree with a few differences.

1. Every level is completely filled.
2. The leaf nodes lean towards the left of the tree.
3. It is not a requirement for the last leaf node to have the right sibling, i.e. a complete binary tree doesn't have to be a full binary tree.



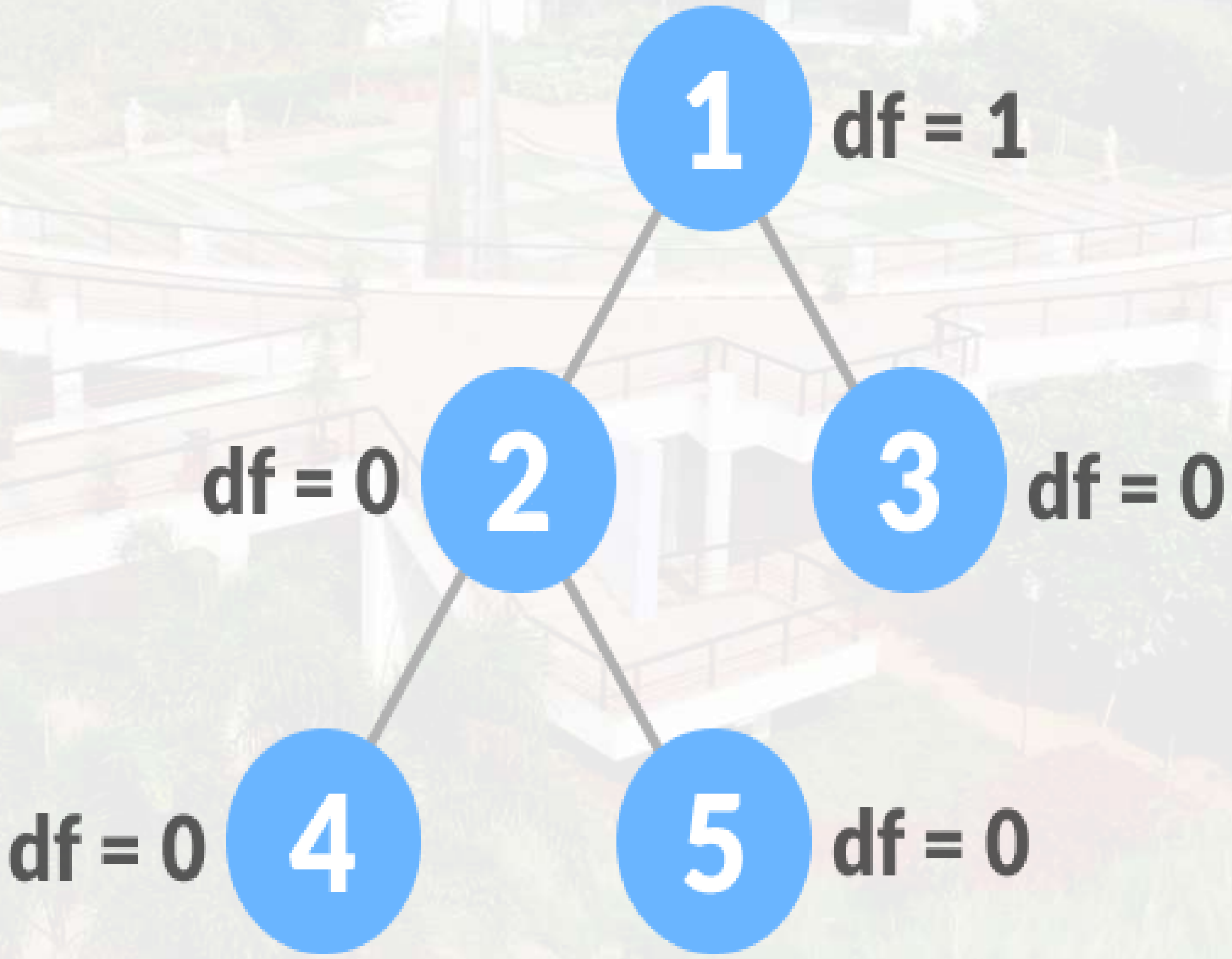
Binary Tree

- **Skewed binary tree:** It is a pathological or degenerate tree where the tree is dominated by either the left nodes or the right nodes. Therefore, there are two types of skewed binary trees, i.e. left-skewed or the right-skewed binary tree.



Binary Tree

- **Balanced binary tree:** The difference between the height of the left and right sub tree for each node is either 0 or 1.



Static Binary Tree Representation :

- **Array Representation :**
- Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

- Array index is a value in tree nodes and array value gives to the parent node of that particular index or node. Value of the root node index is always -1 as there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

- Location number of an array is used to store the size of the tree. The first index of an array that is 'o', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index i is put into the array as its i th element.

Binary Tree Representation using Array

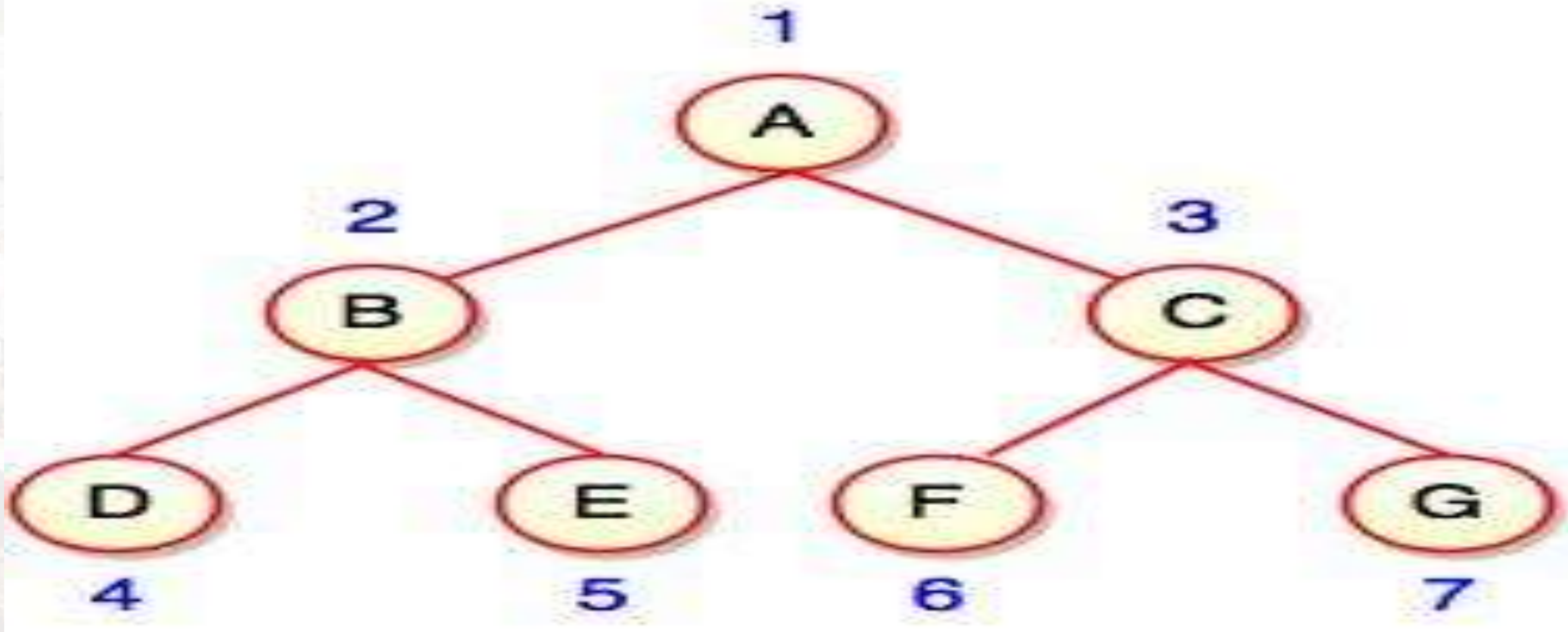
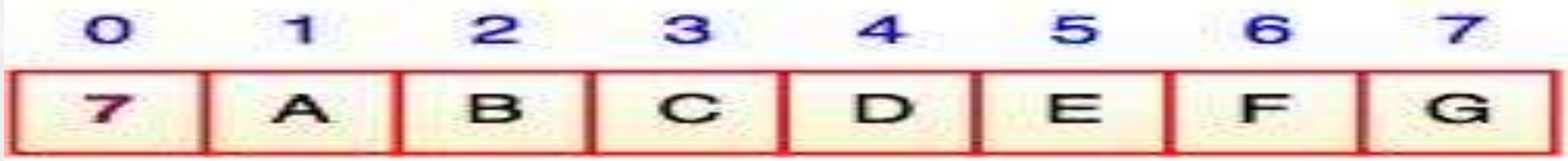


Fig. Binary Tree using Array



0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

Fig. Location Number of an Array in a Tree

Dynamic -Binary Tree Representation

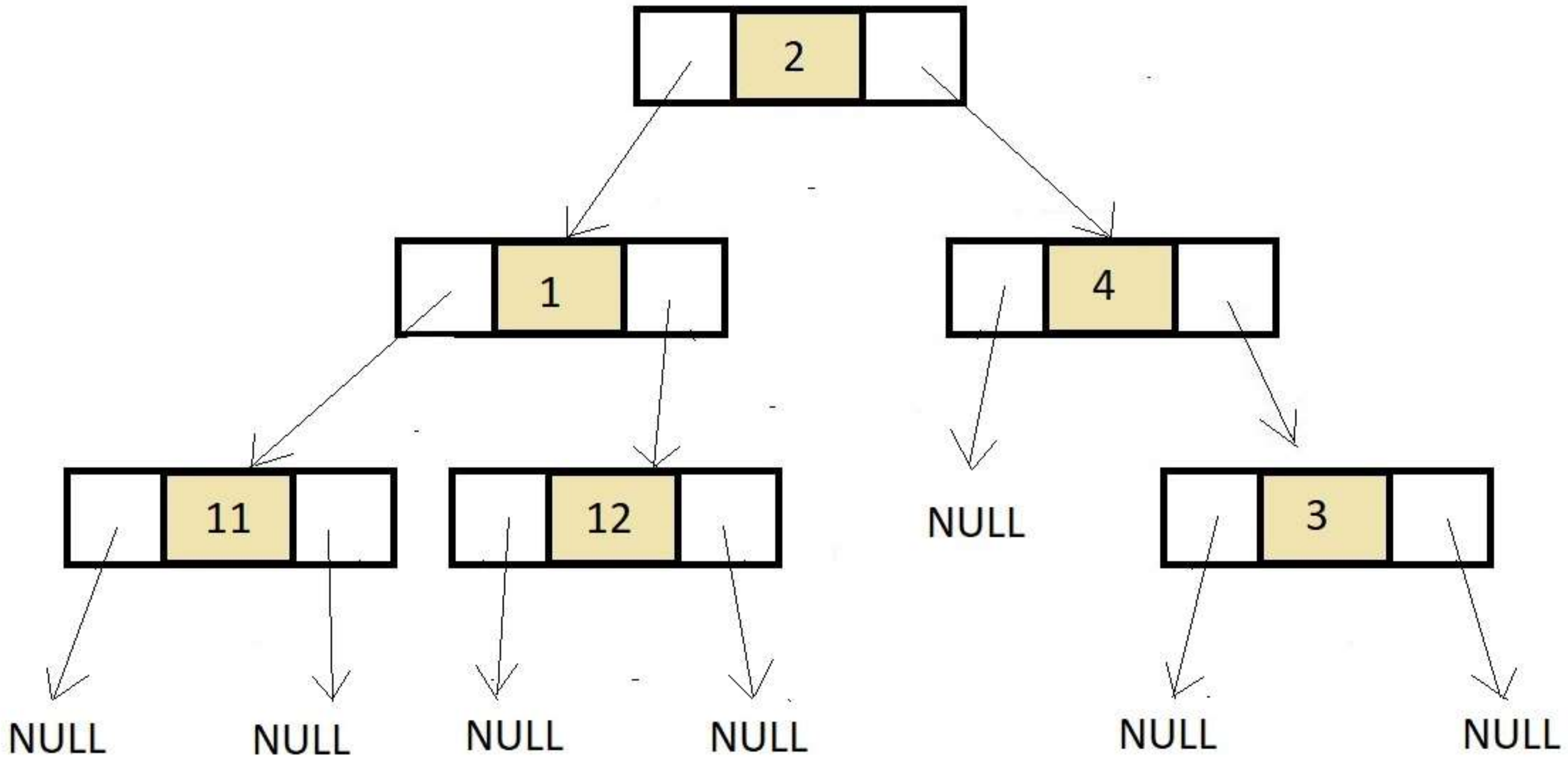
Linked representation :

Binary trees in linked representation are stored in the memory as linked lists. These lists have nodes that aren't stored at adjacent or neighboring memory locations and are linked to each other through the parent-child relationship associated with trees.

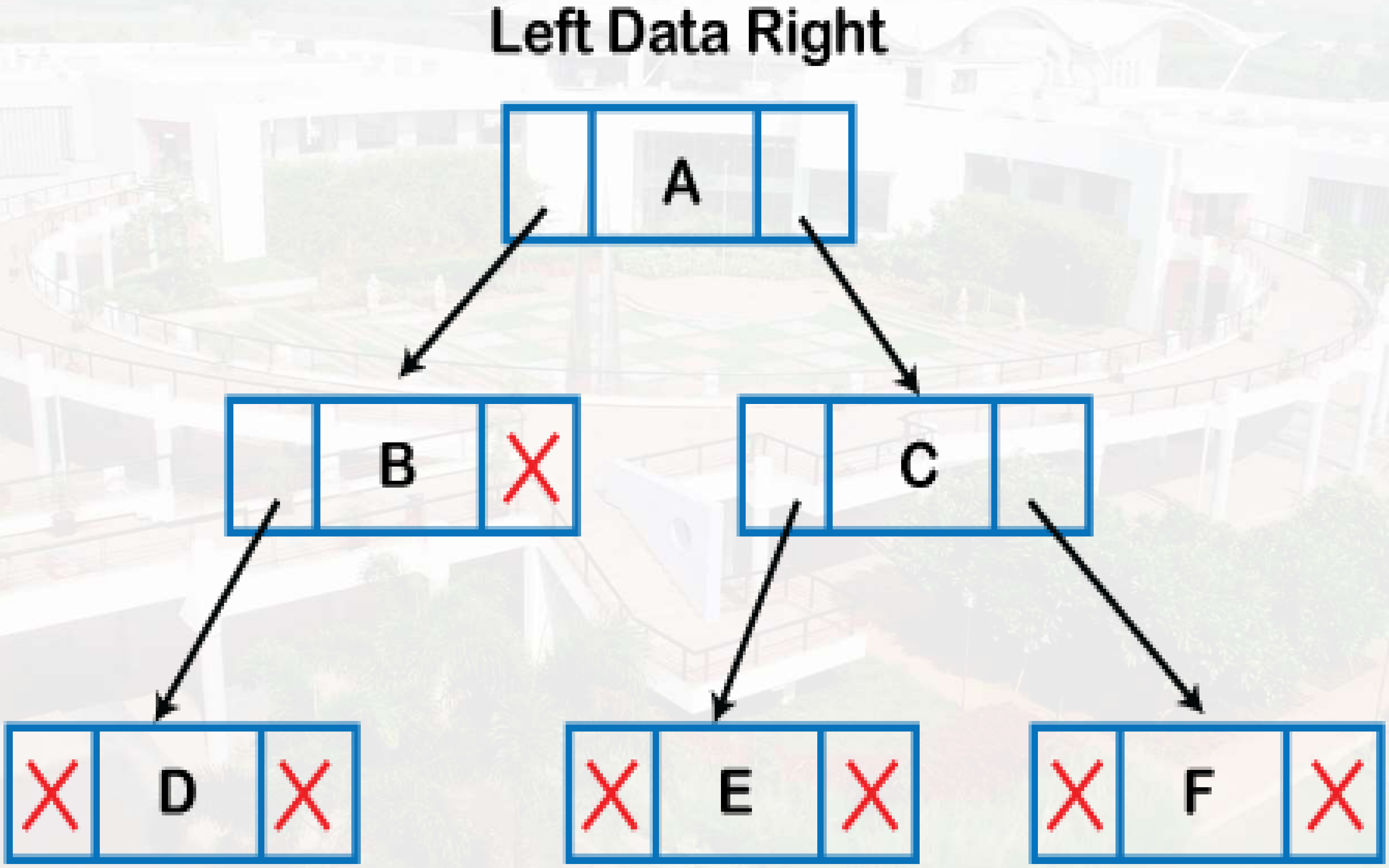
In this representation, each node has three different parts –

- pointer that points towards the right node,
- pointer that points towards the left node,
- data element.

Binary Tree Representation



Binary Tree Representation



Binary Tree Representation

Linked representation :

This is the more common representation. All binary trees consist of a root pointer that points in the direction of the root node. When you see a root node pointing towards null or 0, you should know that you are dealing with an empty binary tree. The right and left pointers store the address of the right and left children of the tree

Operations of Binary Tree

Searching: For searching element 2, we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of O(n).

Insertion: For inserting element as left child of 2, we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of O(n).

Deletion: For deletion of element 2, we have to traverse all elements to find 2 (assuming we do breadth first traversal). Therefore, deletion in binary tree has worst case complexity of O(n).

Binary Tree Examples

Example 1: Consider the example of Perfect binary tree

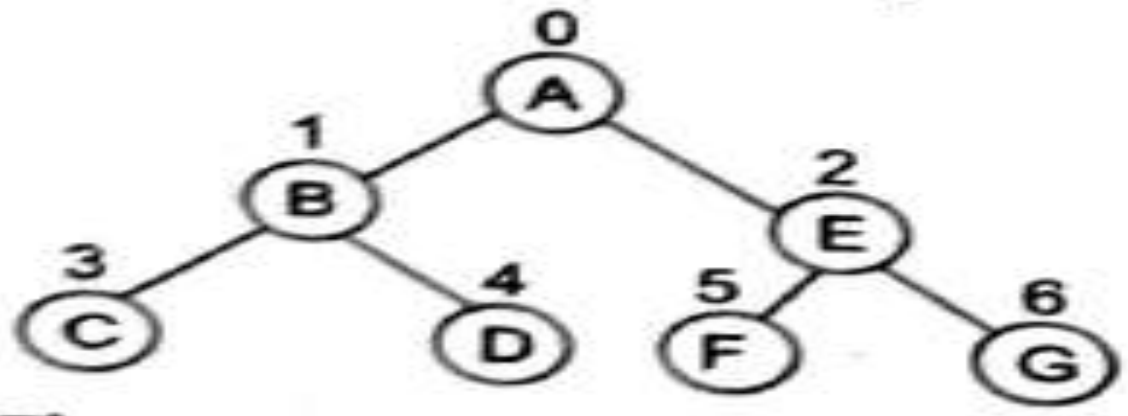


Fig. 6.13: Perfect binary tree

In Fig. 6.13,

Number of levels = 3 (0 to 2) and height = 2

Therefore, we need the array of size $2^{2+1} - 1$ is $2^3 - 1 = 7$.

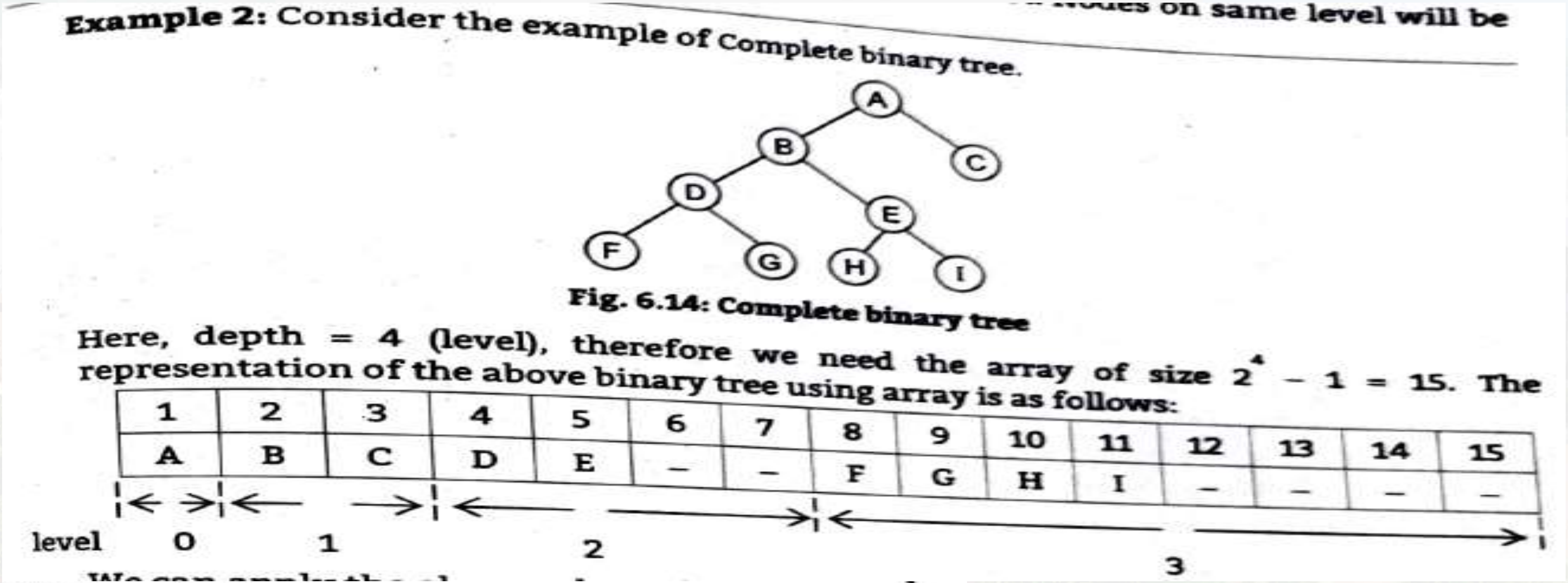
The representation of the above binary tree using array is as followed:

1	2	3	4	5	6	7
A	B	C	D	E	F	G

We will number each node of tree starting from the root. Nodes on same level will be numbered left to right.

Example 2: Consider the example of

Binary Tree Examples



Nodes on same level will be

Example 2: Consider the example of Complete binary tree.

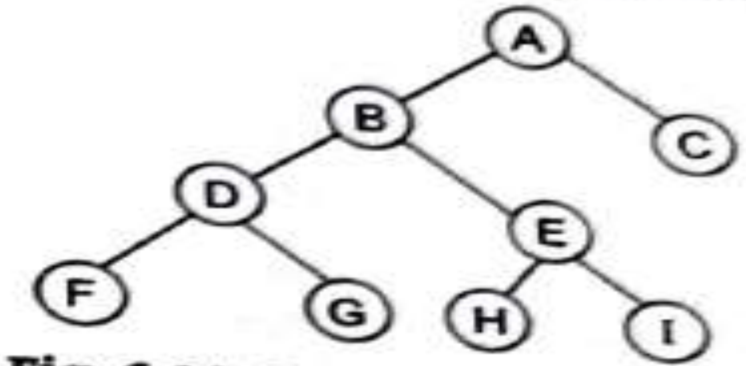
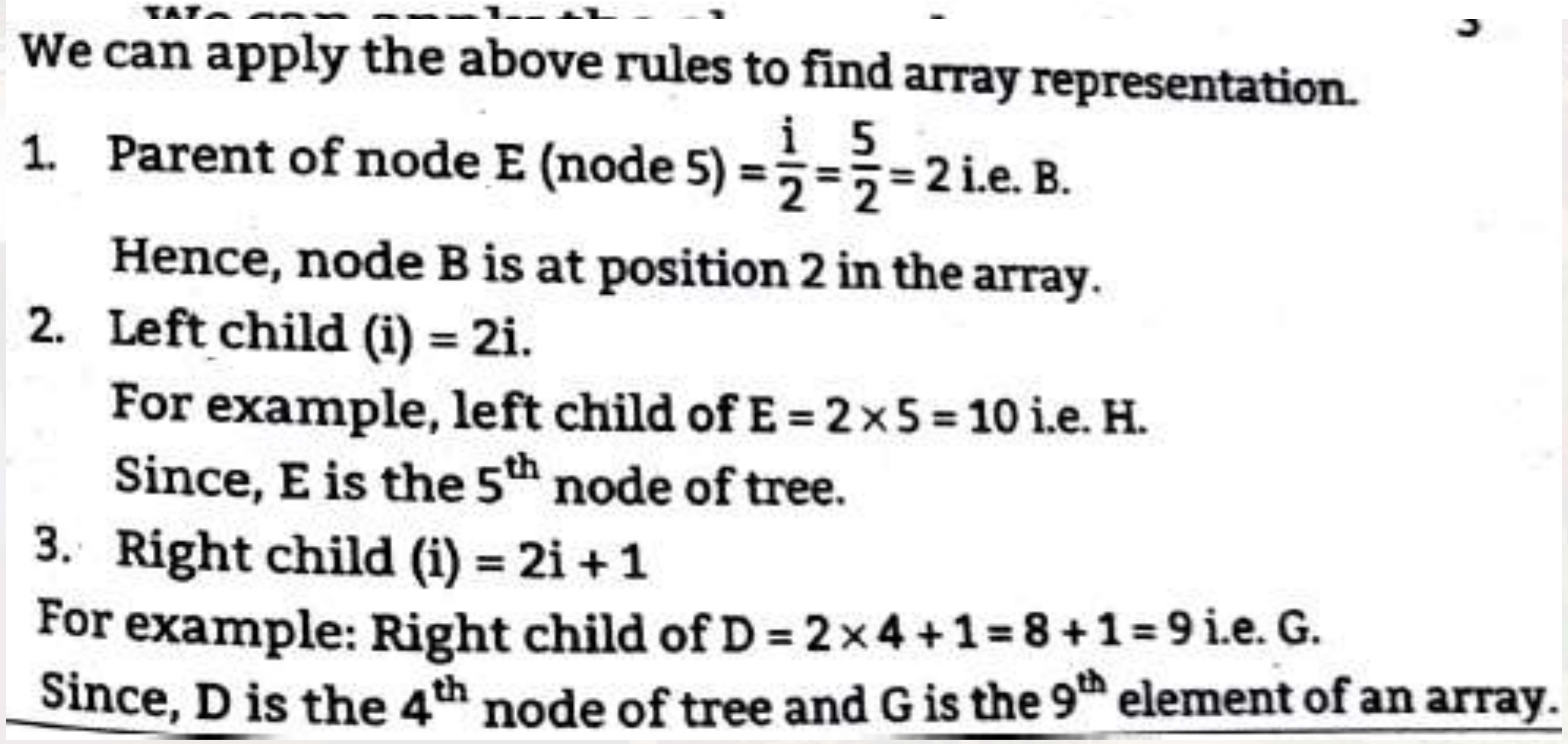


Fig. 6.14: Complete binary tree

Here, depth = 4 (level), therefore we need the array of size $2^4 - 1 = 15$. The representation of the above binary tree using array is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	–	–	F	G	H	I	–	–	–	–

level 0 1 2 3



We can apply the above rules to find array representation.

1. Parent of node E (node 5) $= \frac{i}{2} = \frac{5}{2} = 2$ i.e. B.

   Hence, node B is at position 2 in the array.

2. Left child (i) = 2i.

   For example, left child of E $= 2 \times 5 = 10$ i.e. H.

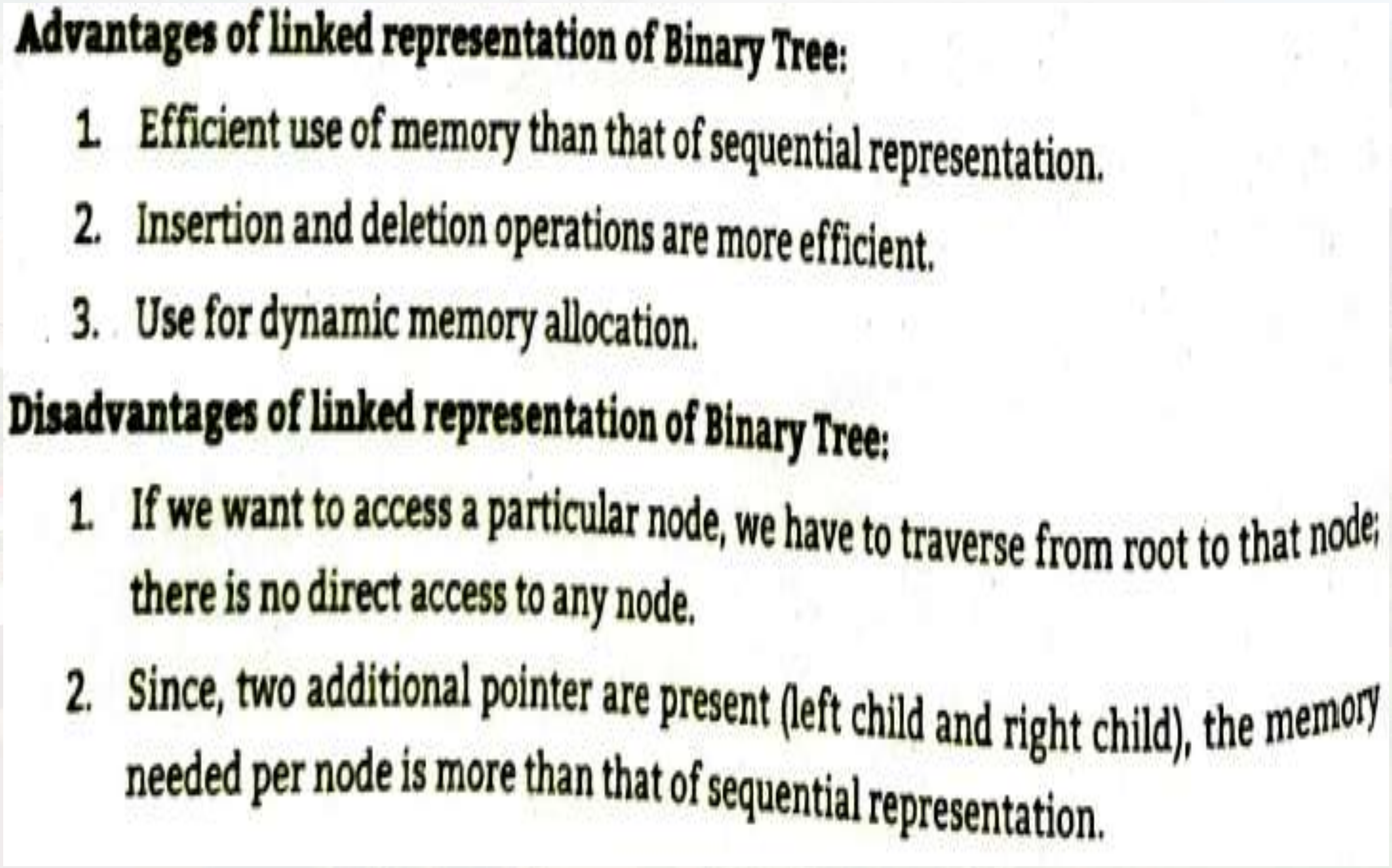
   Since, E is the 5th node of tree.

3. Right child (i) = 2i + 1

For example: Right child of D $= 2 \times 4 + 1 = 8 + 1 = 9$ i.e. G.

Since, D is the 4th node of tree and G is the 9th element of an array.

Advantages and Disadvantages



Advantages of linked representation of Binary Tree:

1. Efficient use of memory than that of sequential representation.
2. Insertion and deletion operations are more efficient.
3. Use for dynamic memory allocation.

Disadvantages of linked representation of Binary Tree:

1. If we want to access a particular node, we have to traverse from root to that node; there is no direct access to any node.
2. Since, two additional pointer are present (left child and right child), the memory needed per node is more than that of sequential representation.

Binary Search Tree

- **Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.**

1. It is called a binary tree because each tree node has a maximum of two children.
2. It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time

Binary Search Tree

- **Properties**
- Follows all properties of the tree data structure.

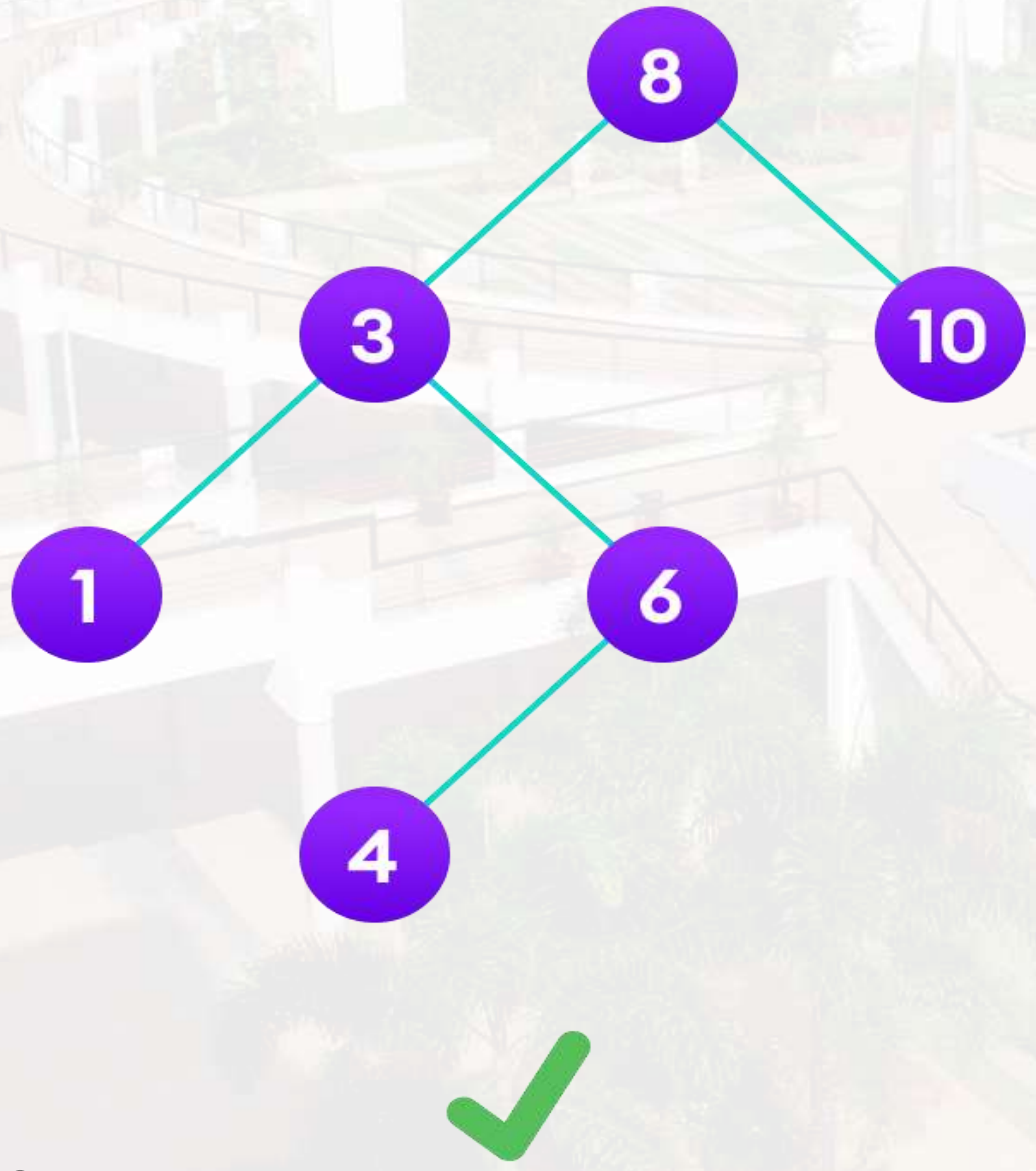
1. The value of the key of the left sub-tree is less than the value of its parent (root) node's key.

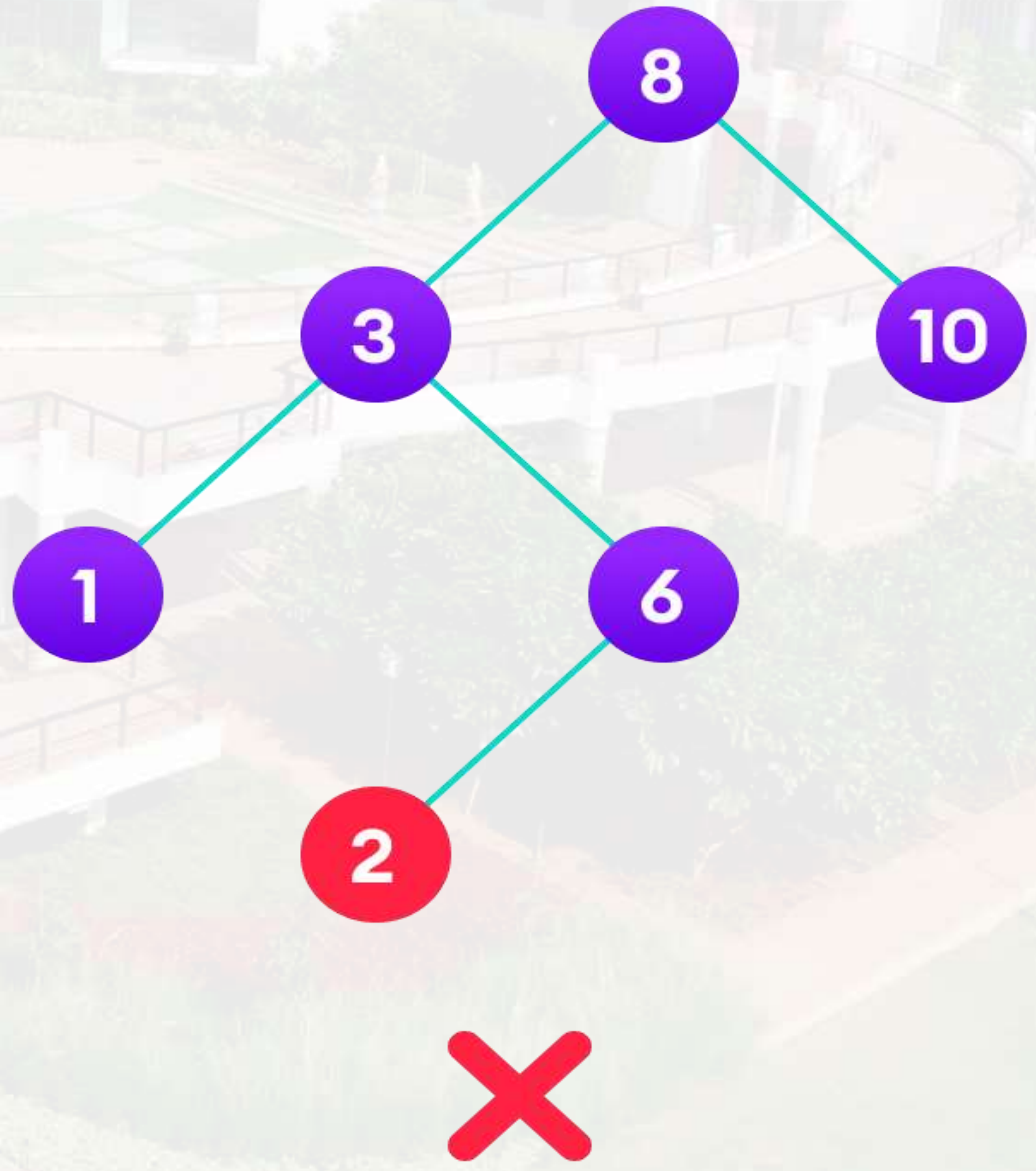
2. The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

3. Both subtrees of each node are also BSTs i.e. they have the above two properties

Binary Search Tree

- The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

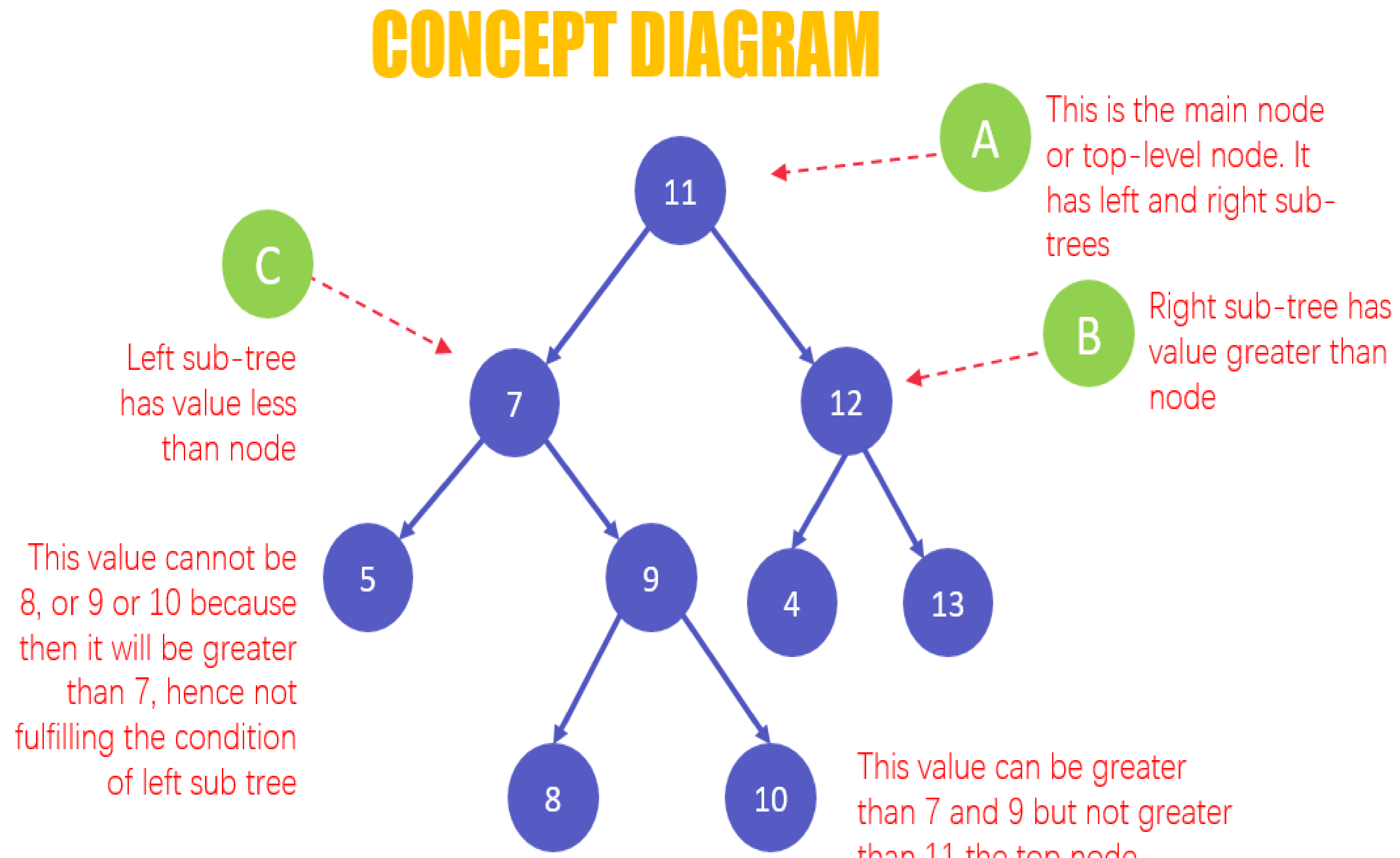




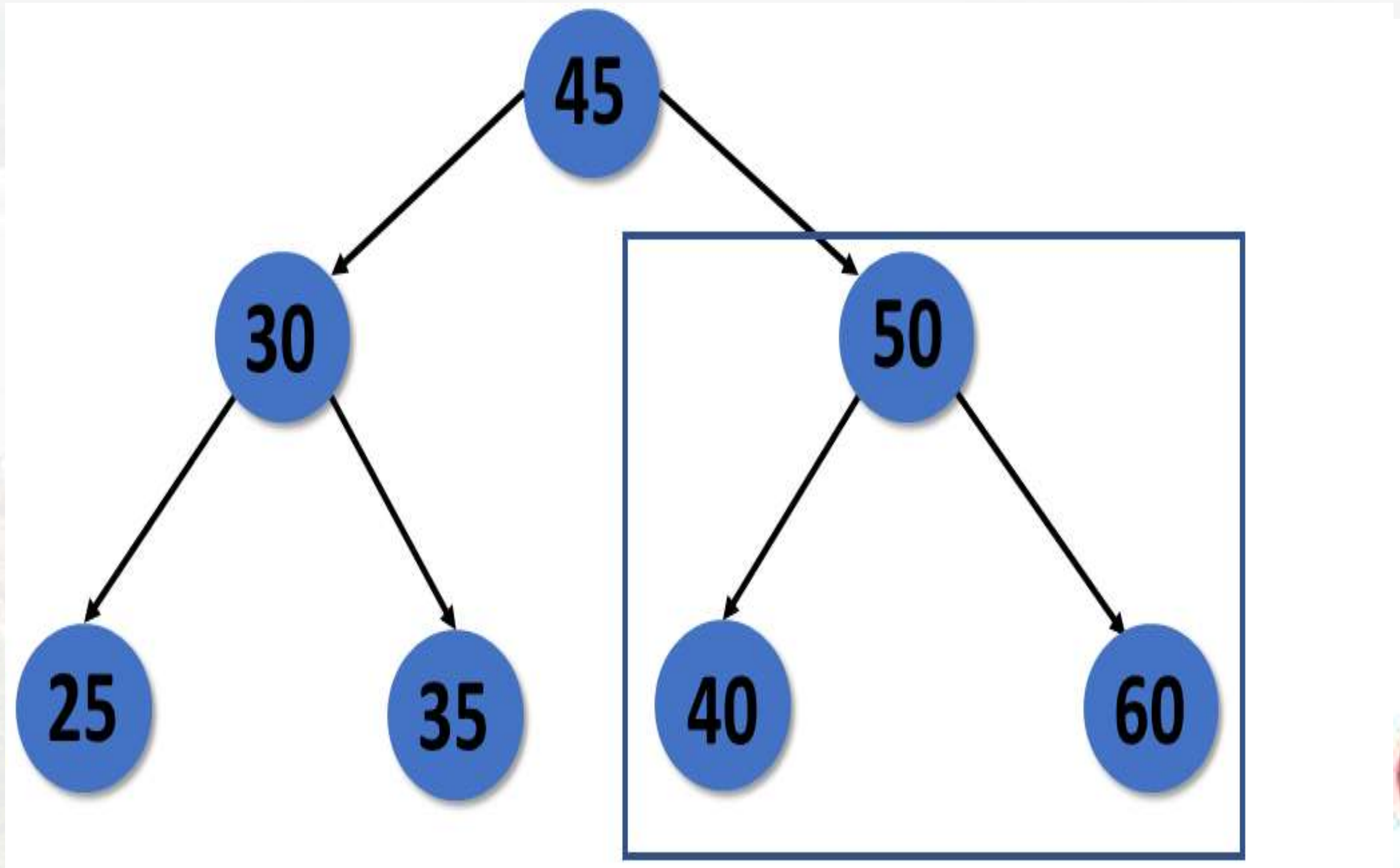
Operation of Binary Search Tree

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.
- **Postorder Traversal** – Traverses a tree in a post-order manner.

Binary Search Tree


CONCEPT DIAGRAM
A
This is the main node or top-level node. It has left and right sub-trees
11
C
B
Right sub-tree has value greater than node
Left sub-tree has value less than node
7
12
This value cannot be 8, or 9 or 10 because then it will be greater than 7, hence not fulfilling the condition of left sub tree
5
9
4
13
8
10
This value can be greater than 7 and 9 but not greater than 11 the top node

Binary Search Tree



Left node value<= root node <= right node value

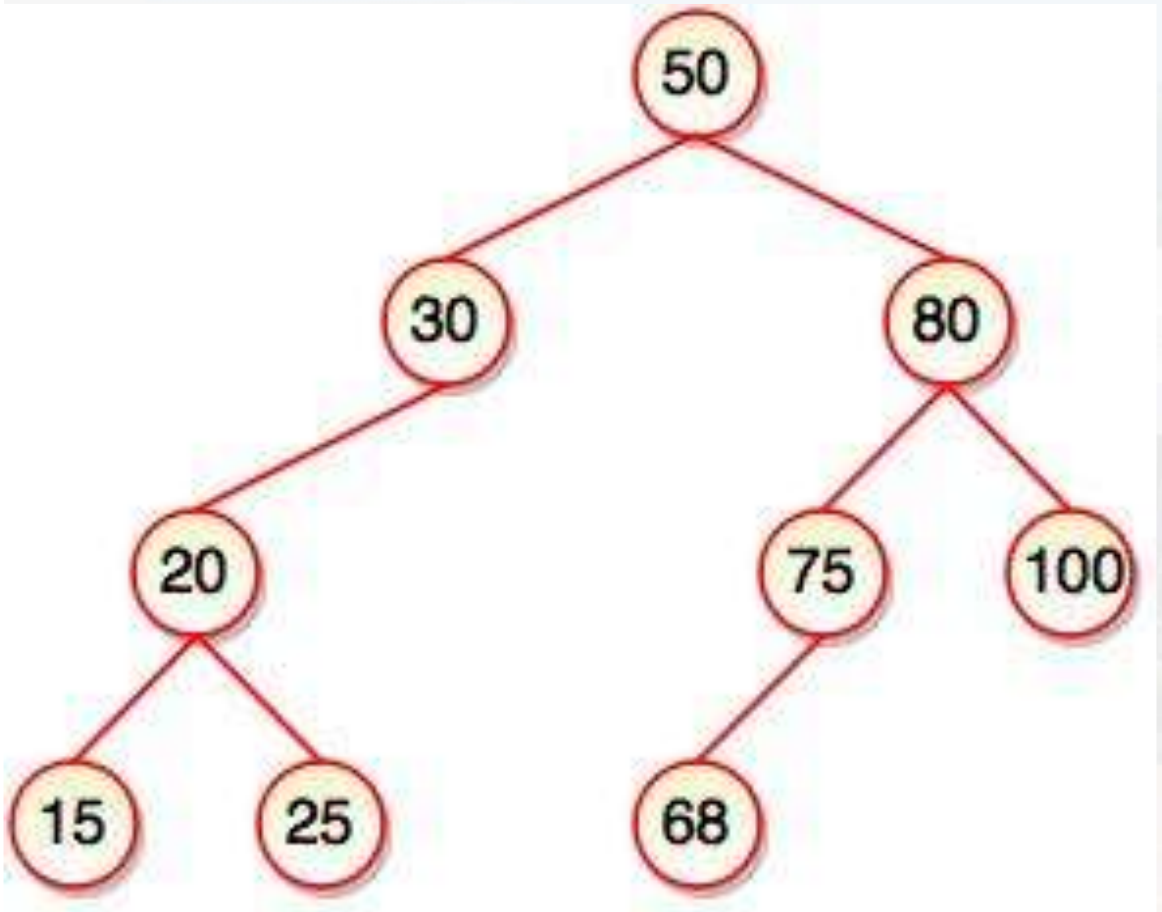
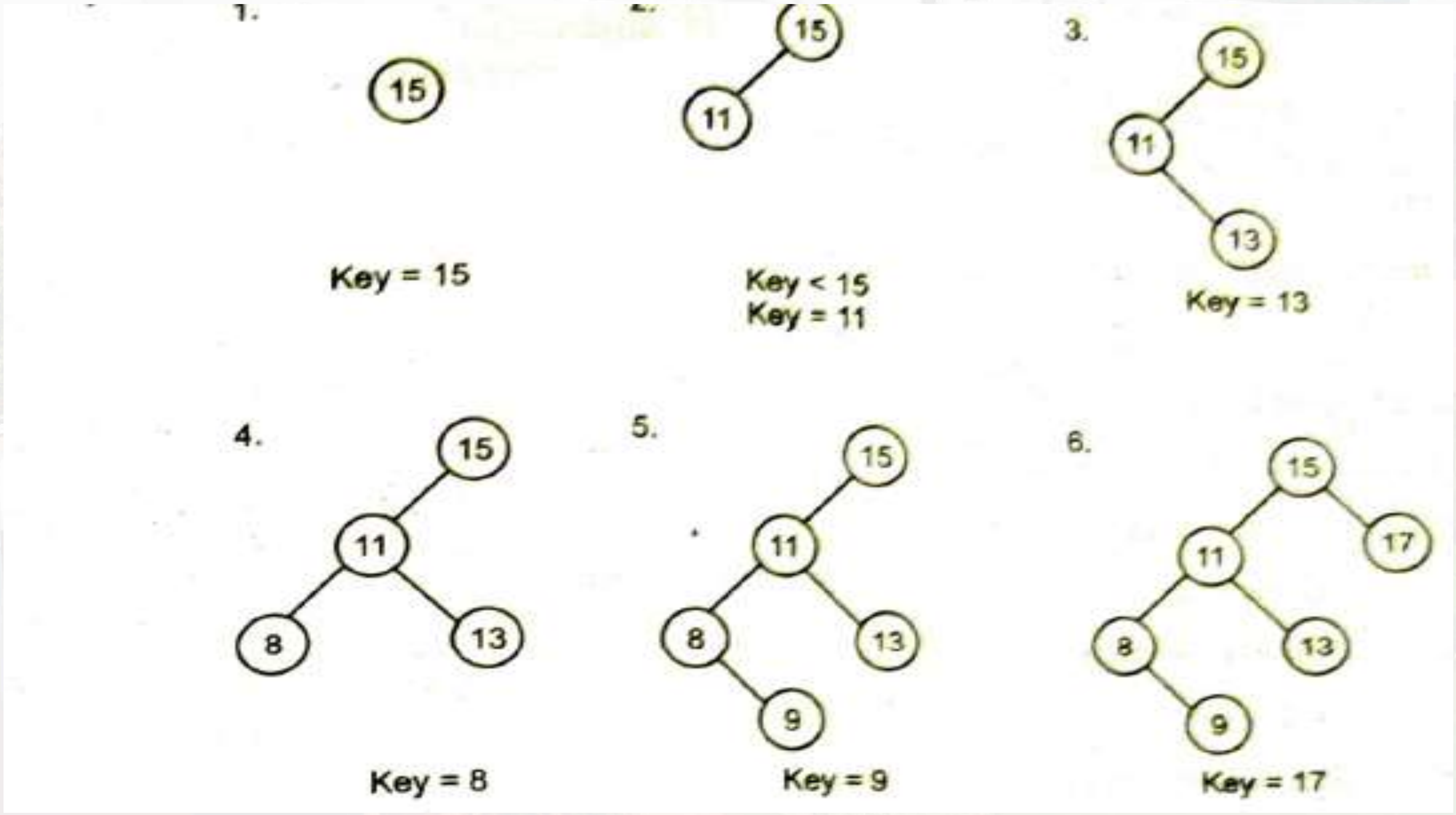


Fig. Binary Search Tree

Binary Search Tree

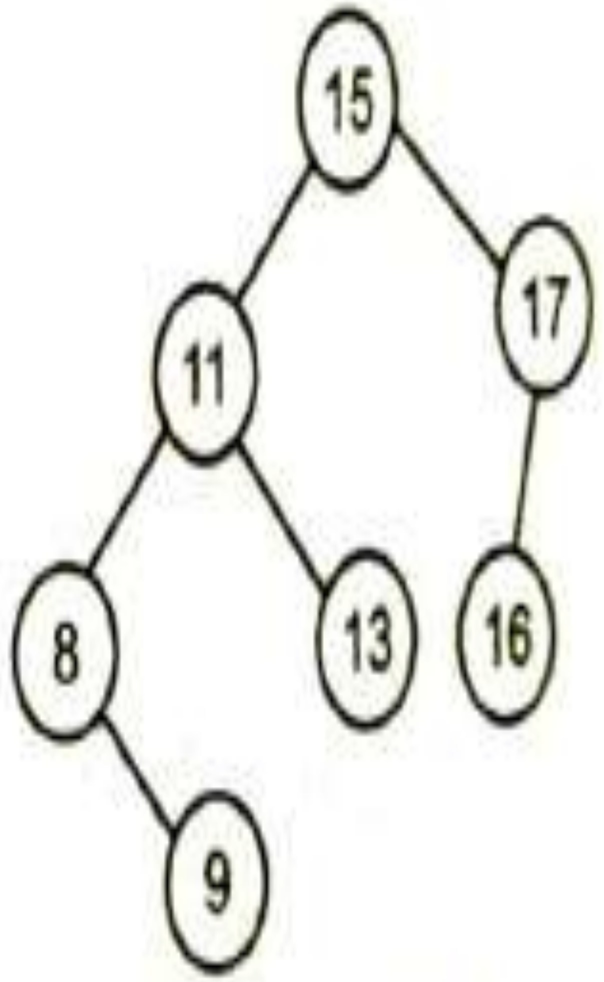
Create the binary search tree using the following data elements.

15, 11, 13, 8, 9, 17, 16, 18



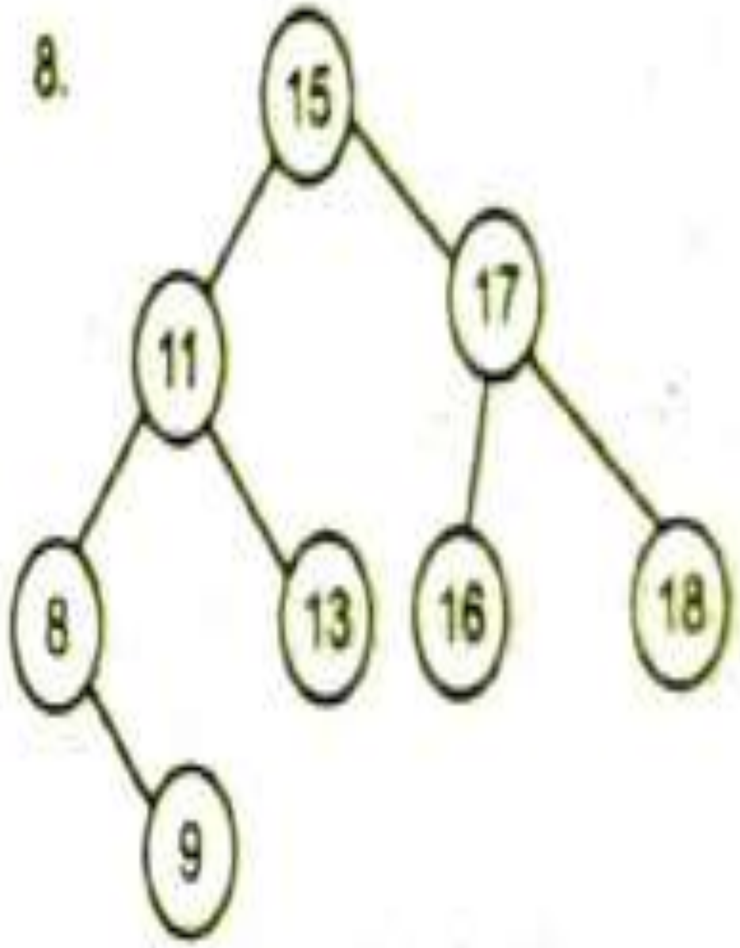
Binary Search Tree

7.



Key = 16

8.



Key = 18

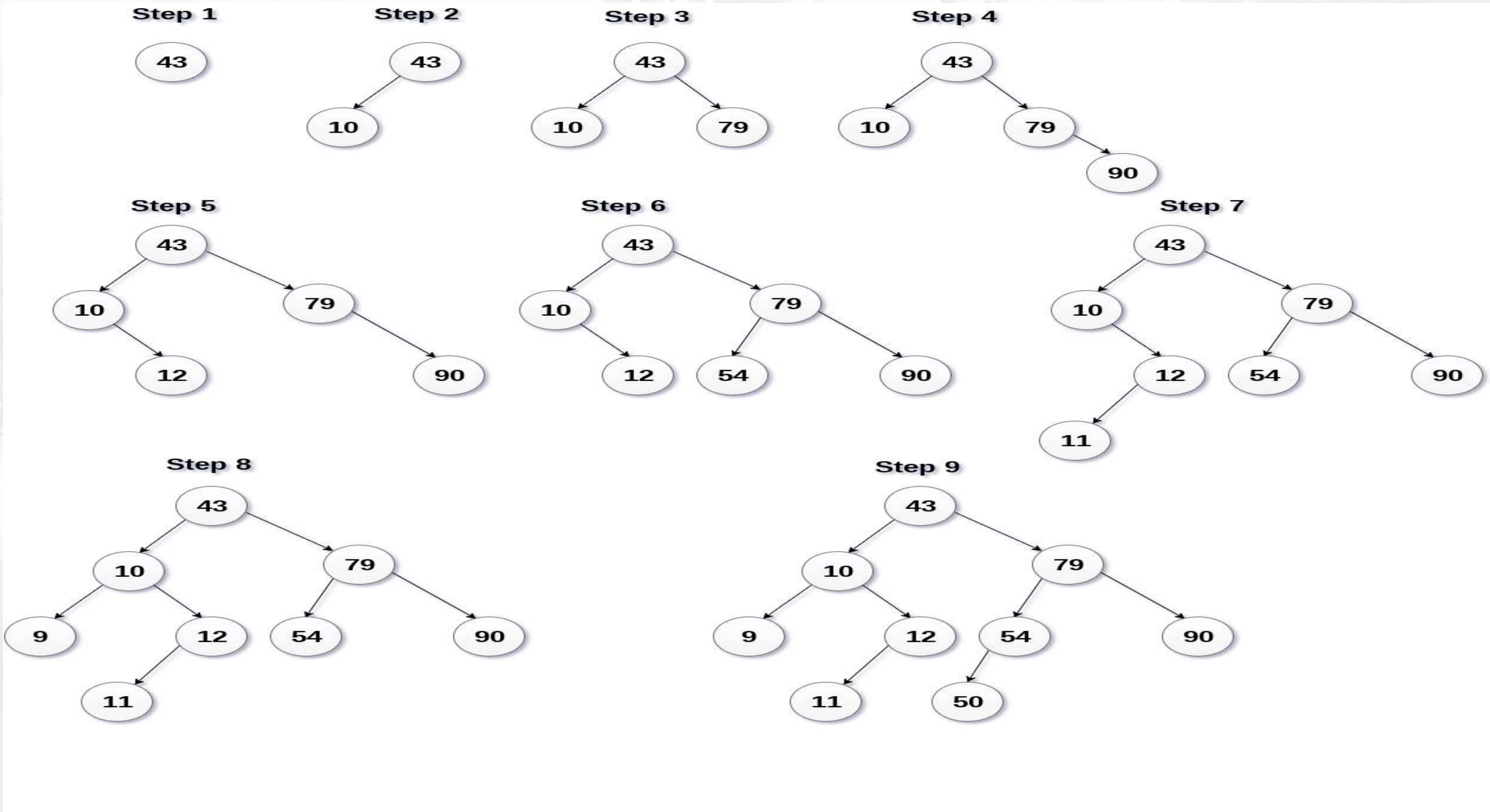
Fig. 6.23

- Here, all values in the left subtrees are less than the root and all values in the right subtrees are greater than the root.

Binary Search Tree

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

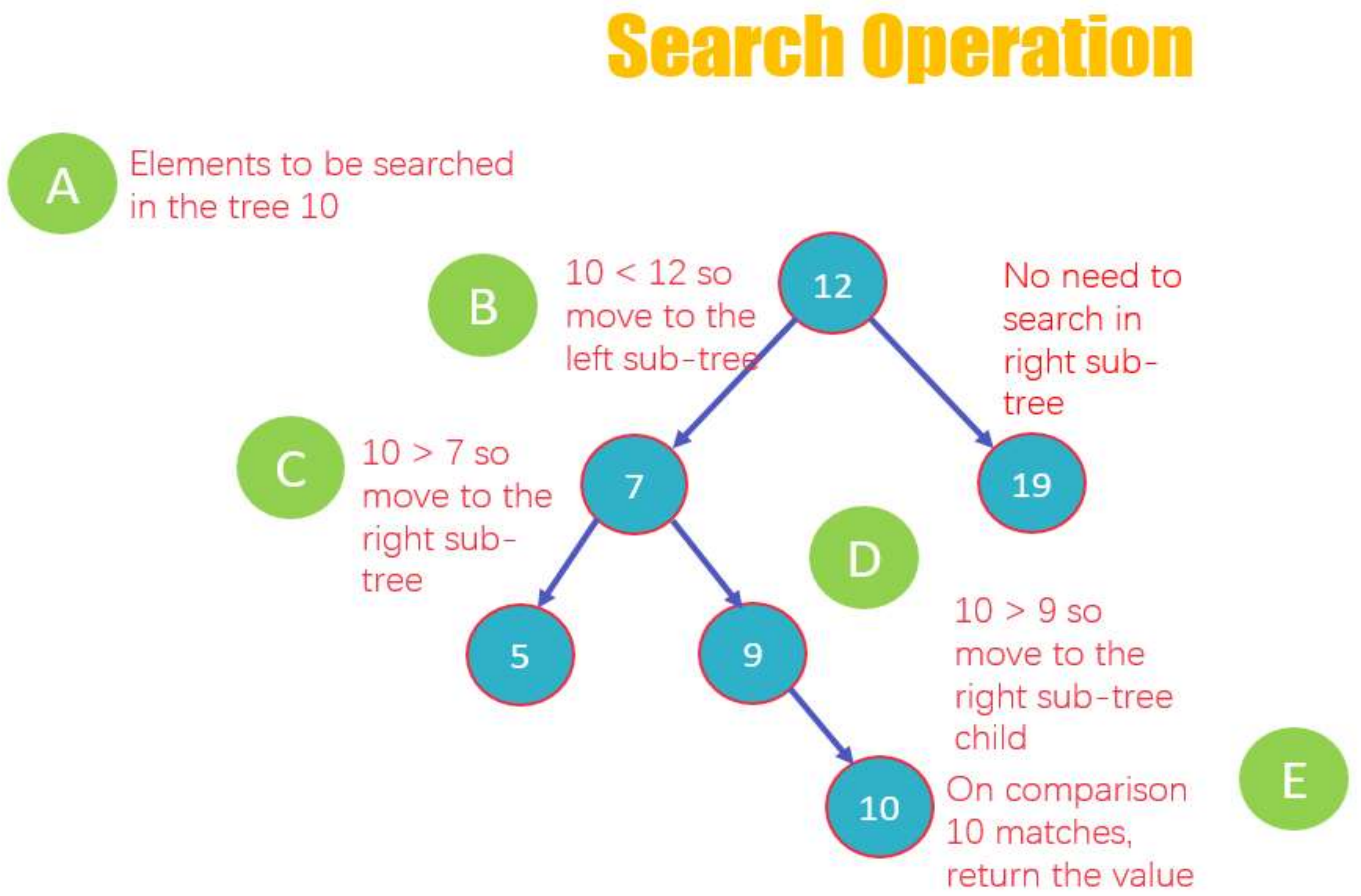


Binary search Tree Creation

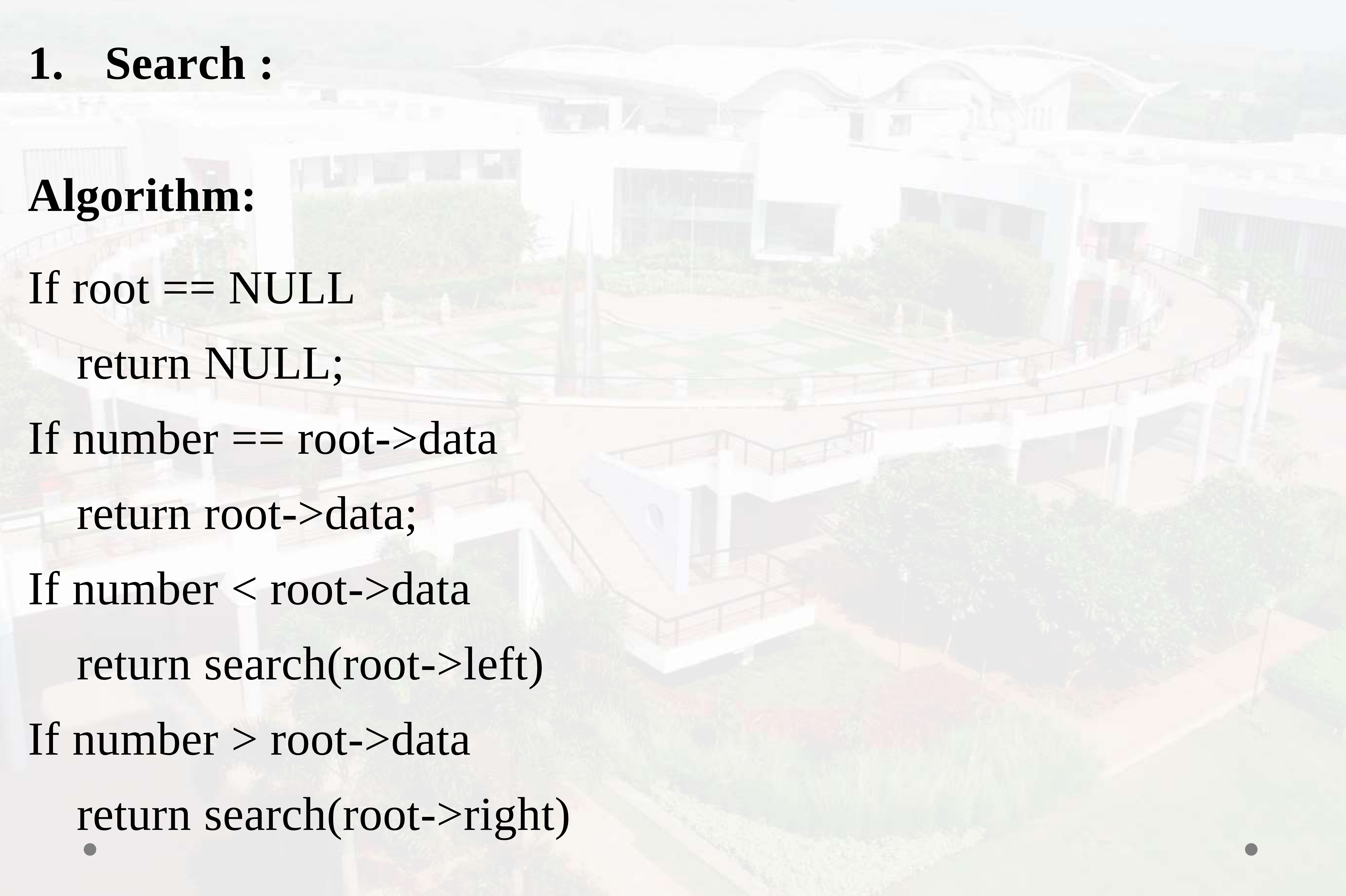
Operation of Binary Search Tree

1. **Create:** creates an empty tree.
2. **Insert:** insert a node in the tree.
3. **Search:** Searches for a node in the tree.
4. **Delete:** deletes a node from the tree.
5. **Inorder:** in-order traversal of the tree.
6. **Preorder:** pre-order traversal of the tree.
7. **Postorder:** post-order traversal of the tree

Operation of Binary Search Tree


Search Operation
A
Elements to be searched in the tree 10
B
10 < 12 so move to the left sub-tree
12
No need to search in right sub-tree
C
10 > 7 so move to the right sub-tree
7
19
D
5
9
10 > 9 so move to the right sub-tree child
10
On comparison 10 matches, return the value
E

Operation of Binary Search Tree



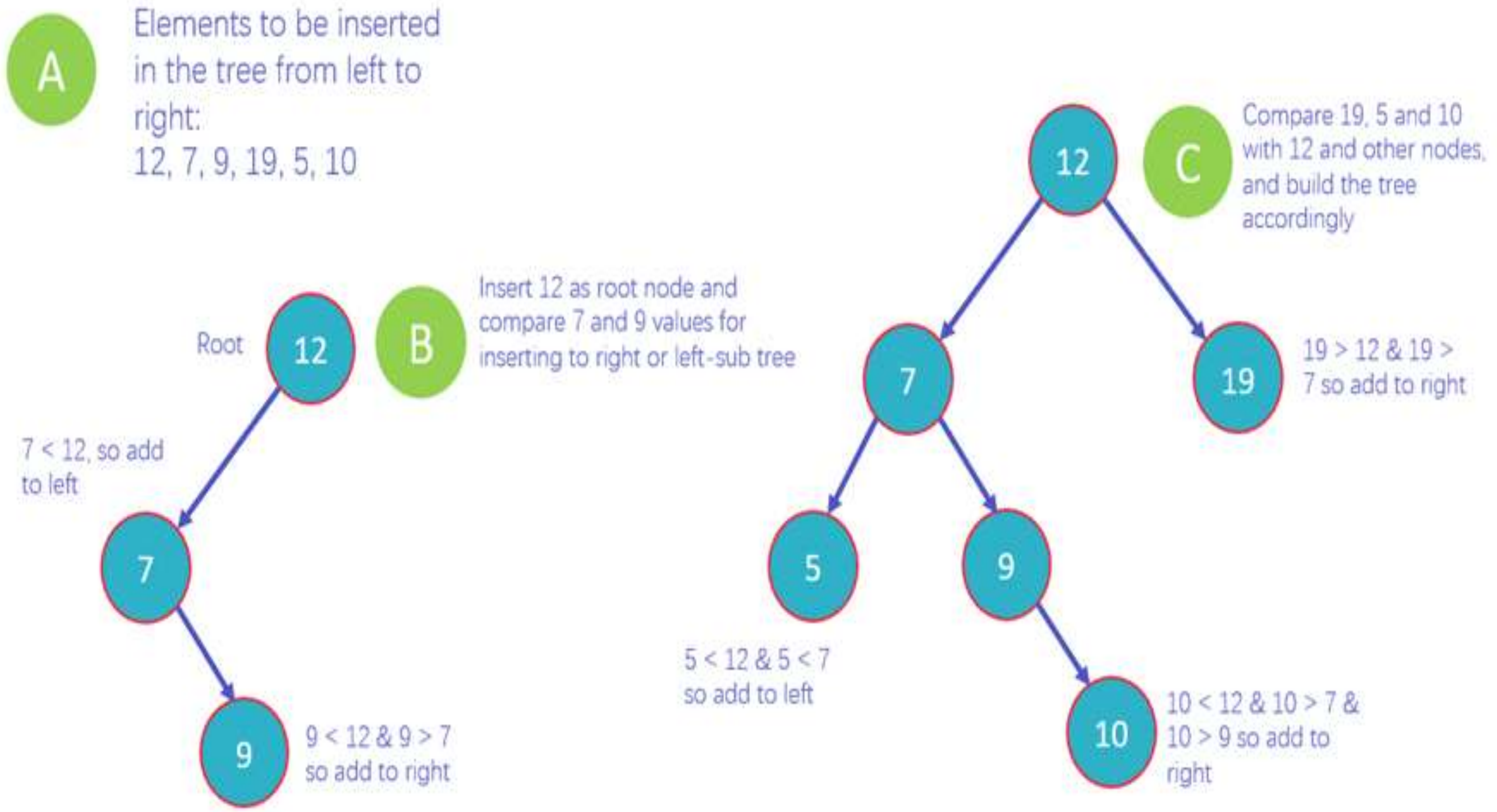
1. Search :

Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

Operation of Binary Search Tree

Insert Operation



A Elements to be inserted in the tree from left to right:
12, 7, 9, 19, 5, 10

B Insert 12 as root node and compare 7 and 9 values for inserting to right or left-sub tree

C Compare 19, 5 and 10 with 12 and other nodes, and build the tree accordingly

Root
12
7 < 12, so add to left
7
9
9 < 12 & 9 > 7 so add to right
12
7
19
19 > 12 & 19 > 7 so add to right
5
9
5 < 12 & 5 < 7 so add to left
10
10 < 12 & 10 > 7 & 10 > 9 so add to right

Operation of Binary Search Tree

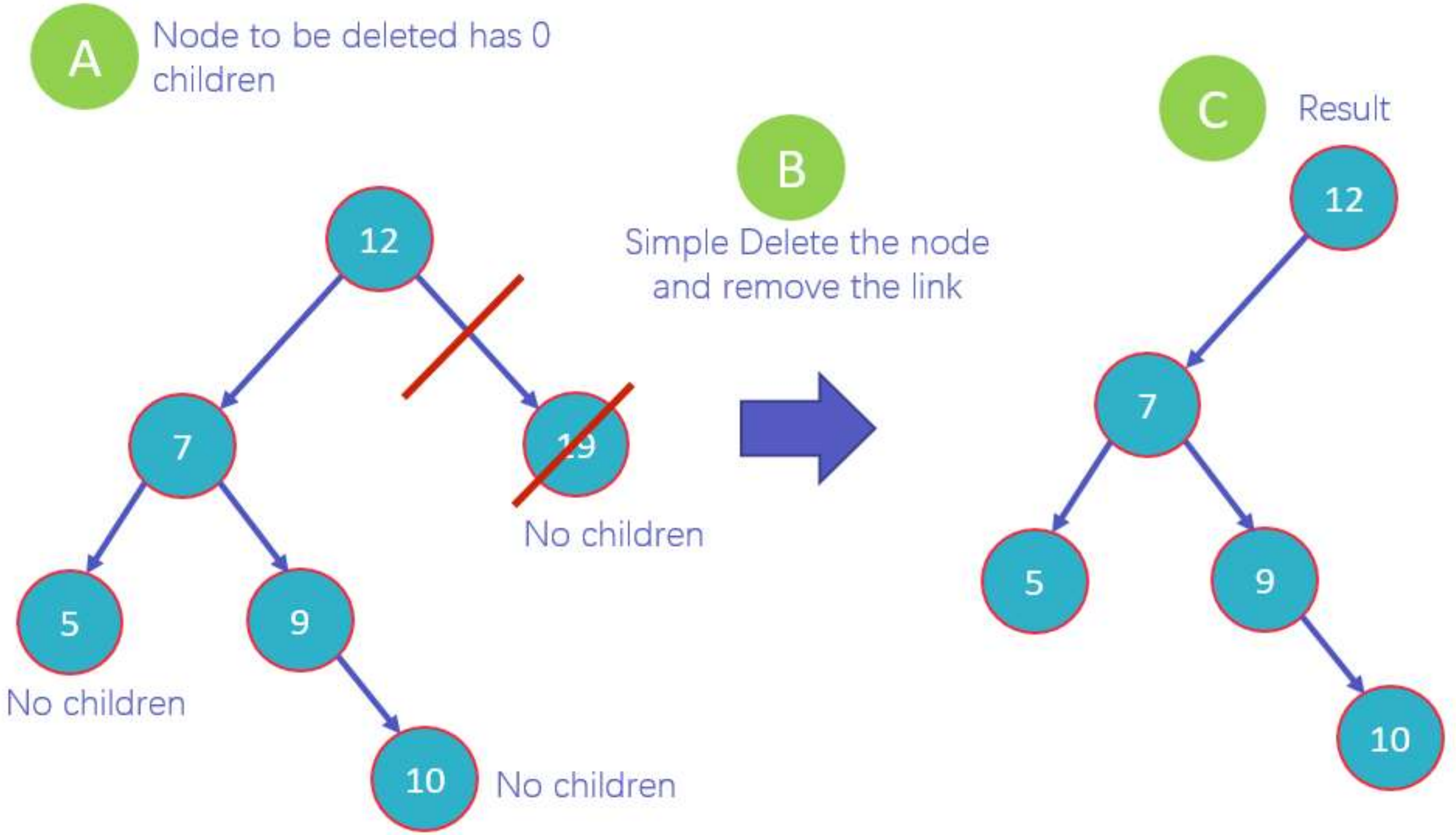
2. Insert :

Algorithm:

```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

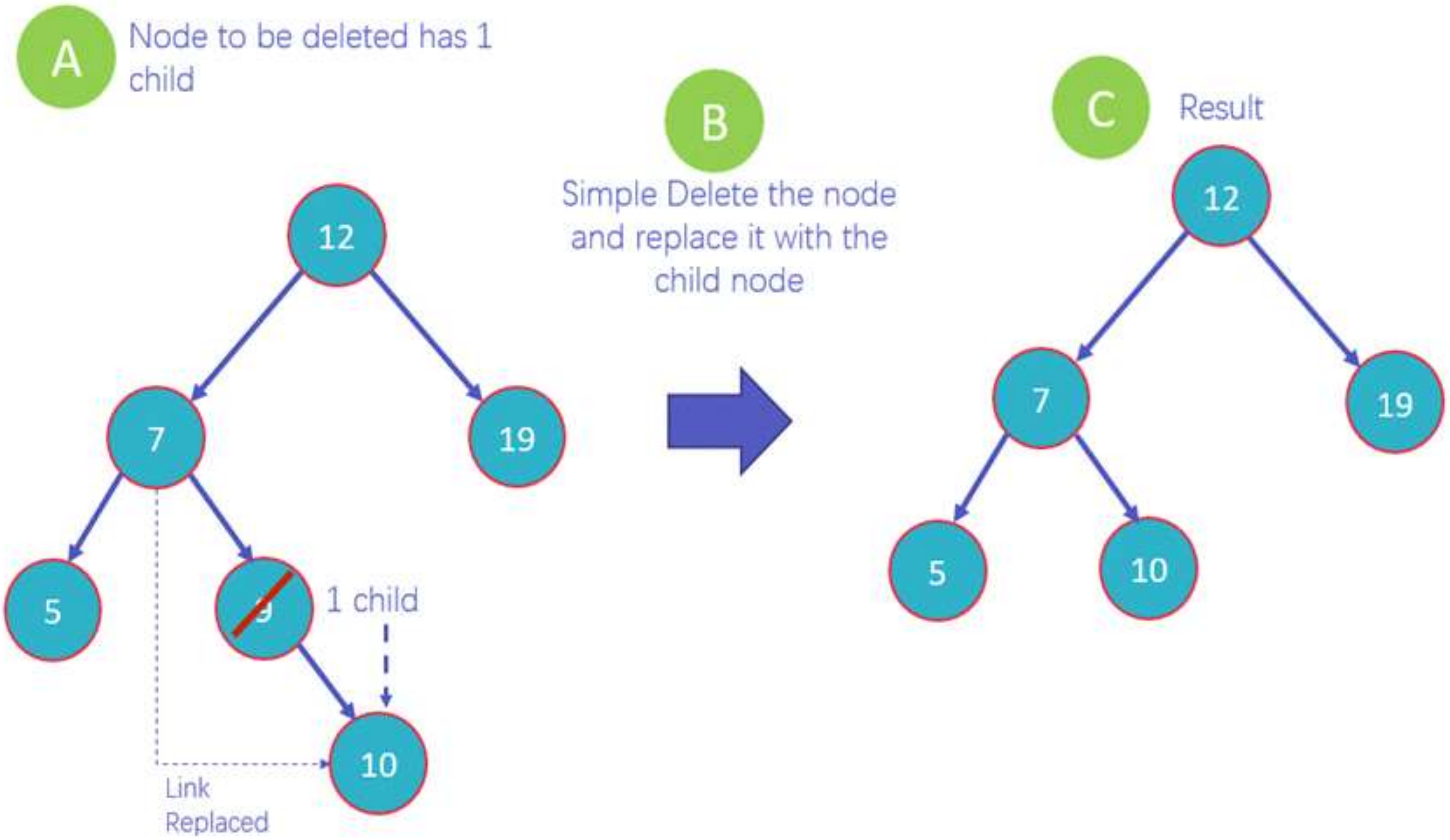
Operation of Binary Search Tree

Delete Operation – Case 1


A
Node to be deleted has 0 children
B
Simple Delete the node and remove the link
C
Result
12
7
19
5
9
10
No children
No children
No children
12
7
5
9
10

Operation of Binary Search Tree

Delete Operation – Case 2


A
Node to be deleted has 1 child
B
Simple Delete the node and replace it with the child node
C
Result
12
7
19
5
9
1 child
10
Link Replaced
12
7
19
5
10

Operation of Binary Search Tree

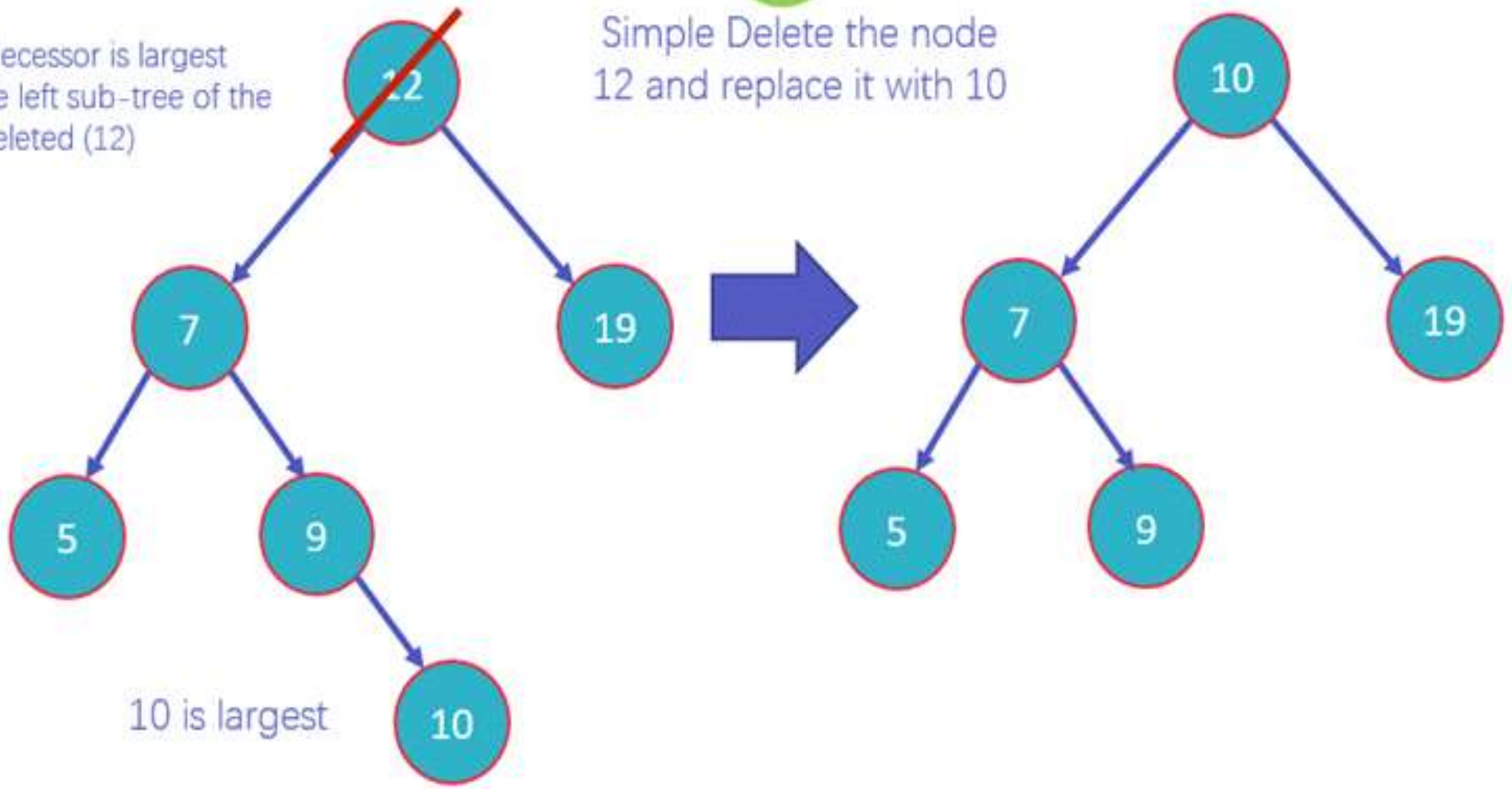
Delete Operation – Case 3 (a)

A Node to be deleted has 2 child
Replace Situation: In Order Predecessor

B In Order predecessor is largest element in the left sub-tree of the node to be deleted (12)

C Simple Delete the node 12 and replace it with 10

D Result



Operation of Binary Search Tree

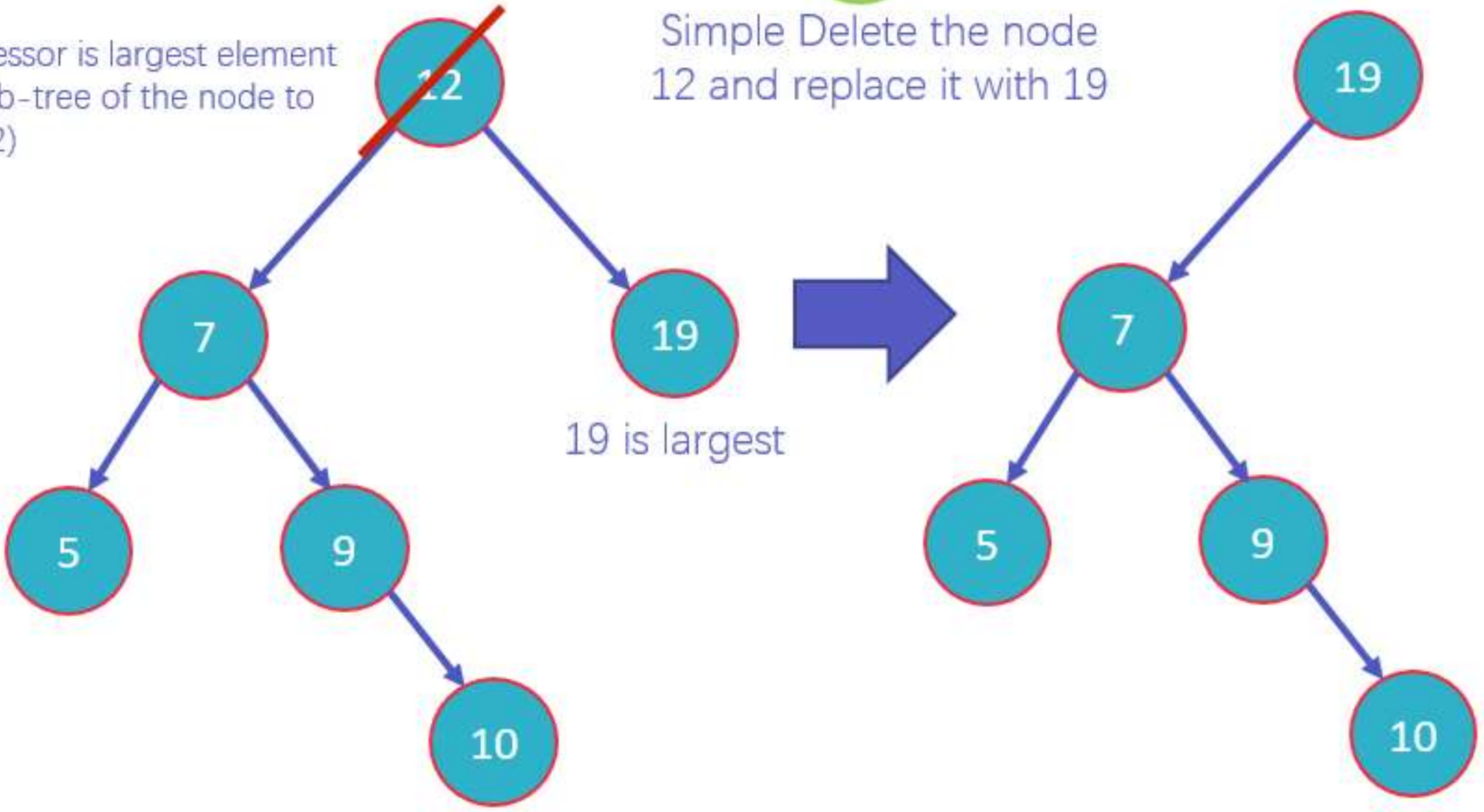
Delete Operation – Case 3 (b)

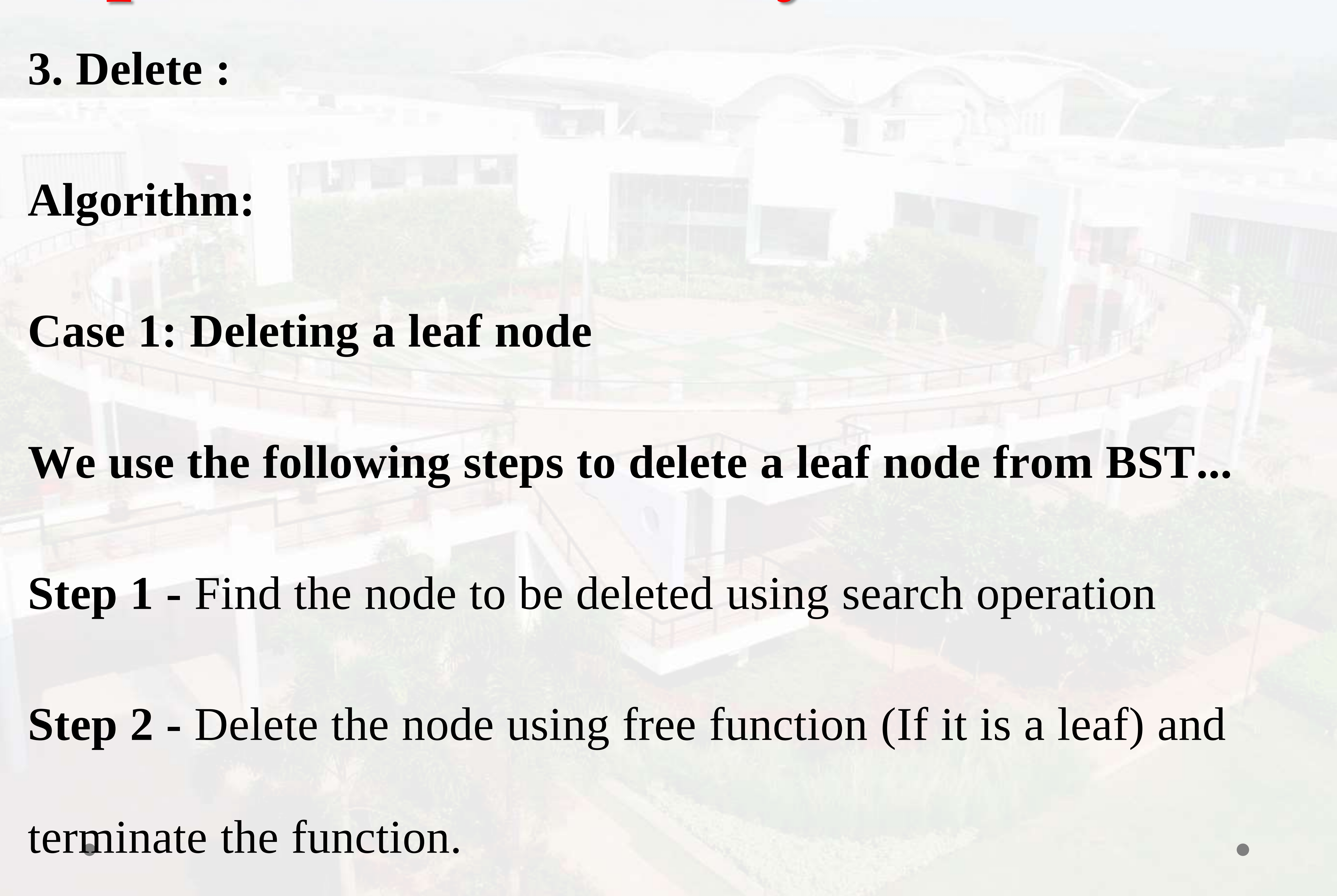
A Node to be deleted has 2 child
Replace Situation: In Order Successor

B In Order successor is largest element in the right sub-tree of the node to be deleted (12)

C Simple Delete the node 12 and replace it with 19

D Result





Operation of Binary Search Tree

3. Delete :

Algorithm:

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - Delete the node using free function (If it is a leaf) and

terminate the function.

Operation of Binary Search Tree

3. Delete :

Algorithm:

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using free function and terminate the function.

Operation of Binary Search Tree

3. Delete :

Algorithm:

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3 - Swap both deleting node and node which is found in the above step.

Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2

Operation of Binary Search Tree

3. Delete :

Algorithm:

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.

Step 7 - Repeat the same process until the node is deleted from the tree.

Convert a Generic Tree (N-array Tree) to Binary Tree

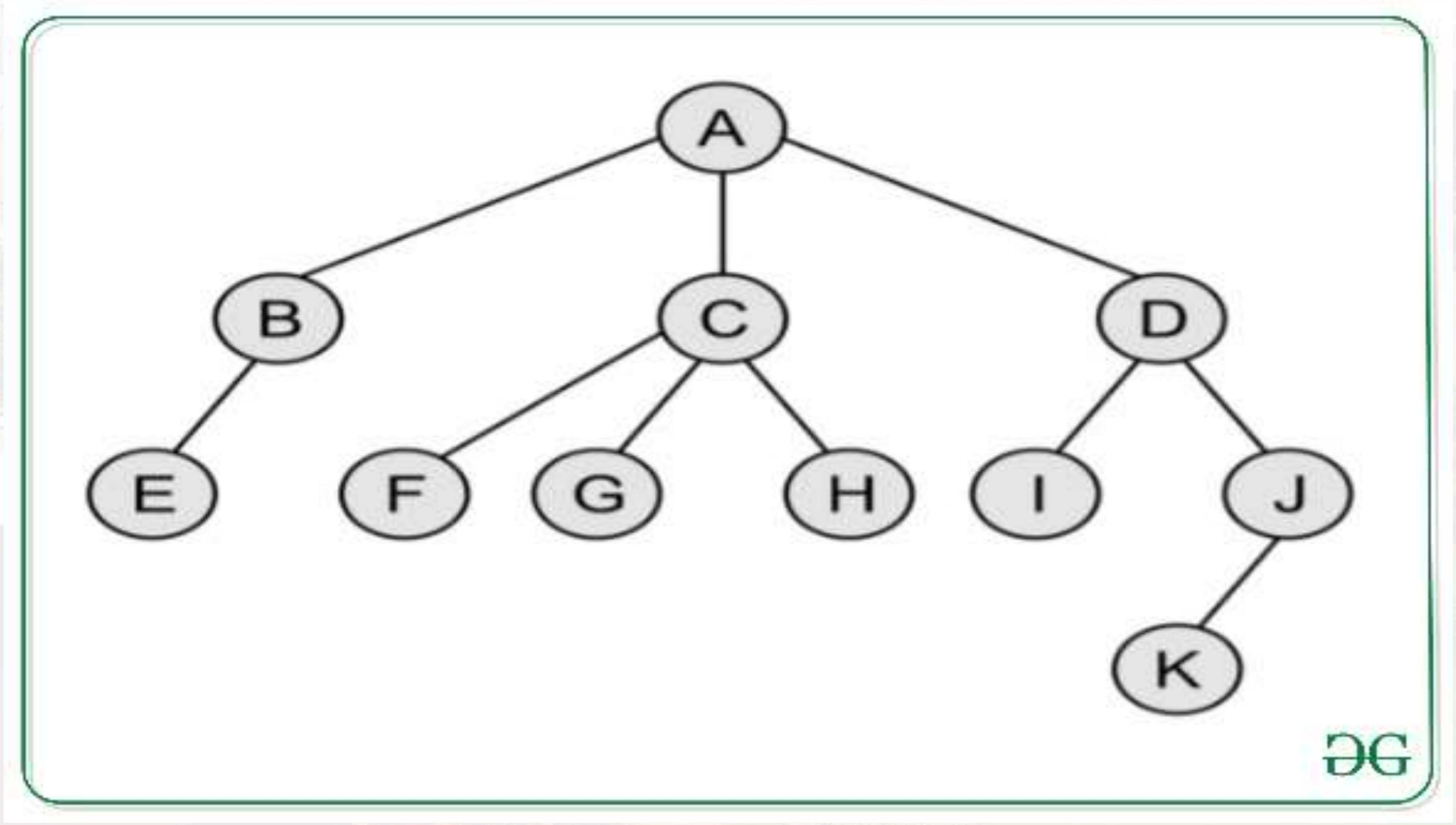
Following are the rules to convert a Generic(N-array tree) to binary tree :

1. The root of the Binary Tree is the Root of the Generic Tree.
2. The left child of a node in the Generic Tree is the Left child of that node in the Binary Tree.
3. The right sibling of any node in the Generic Tree is the Right child of that node in the Binary Tree.

Convert a Generic Tree(N-array Tree) to Binary Tree

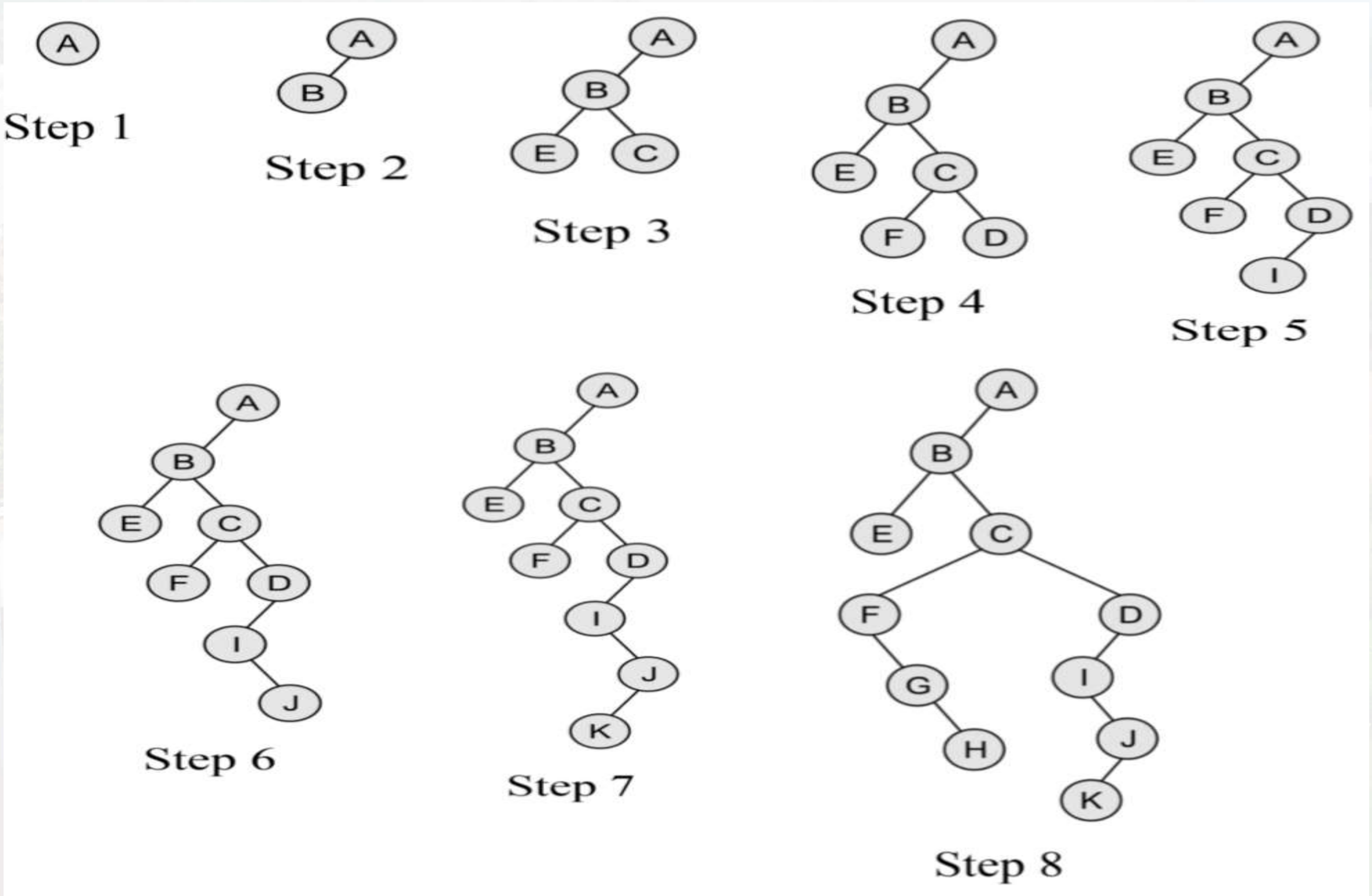
Examples:

Convert the following Generic Tree to Binary Tree:



Convert a Generic Tree(N-array Tree) to Binary Tree

Below is the Binary Tree of the above Generic Tree:



Convert a Generic Tree(N-array Tree) to Binary Tree

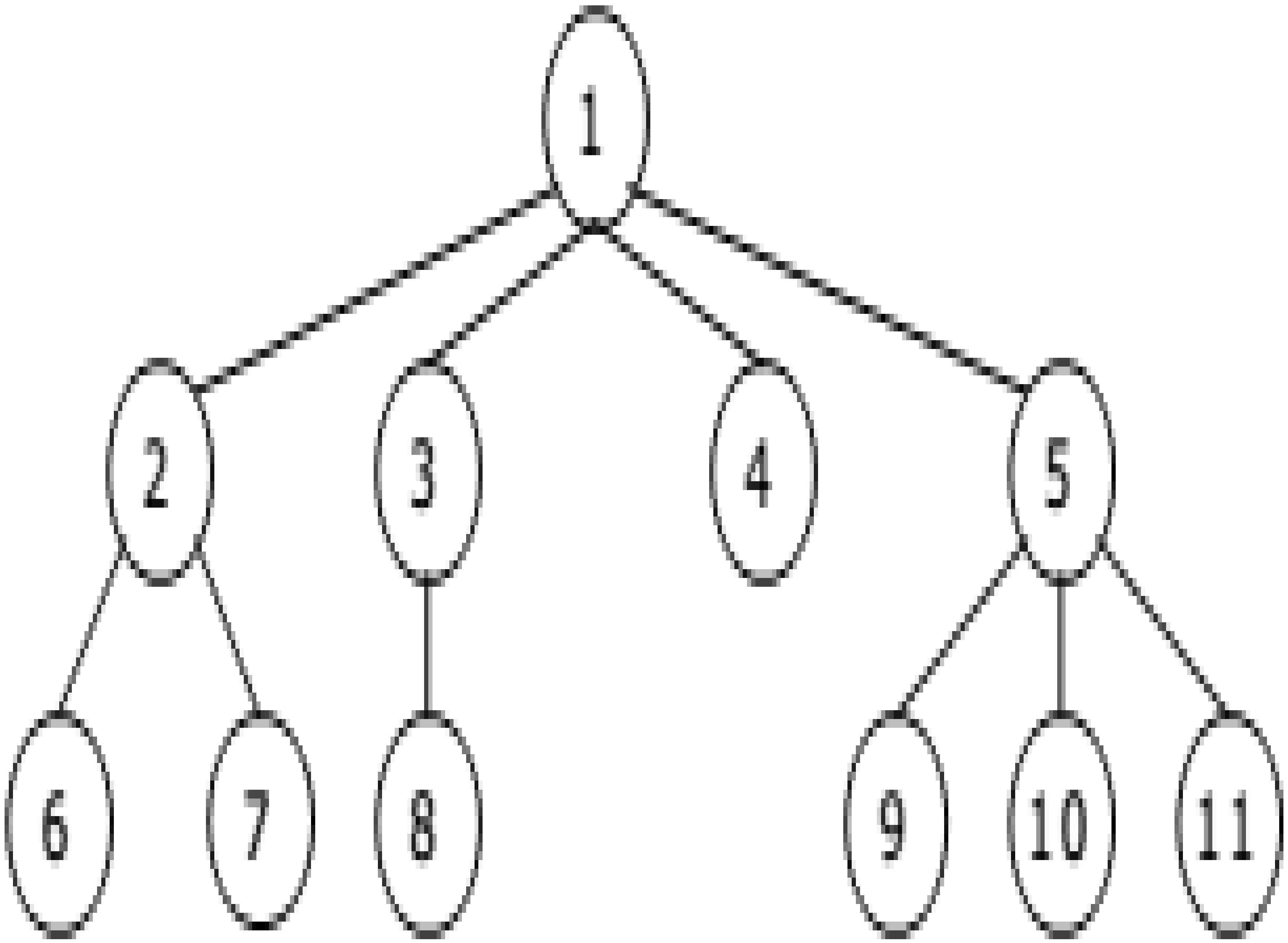
Note: If the parent node has only the right child in the general tree then it becomes the rightmost child node of the last node following the parent node in the binary tree.

In the above example, if node B has the right child node L then in binary tree representation L would be the right child of node D.

Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

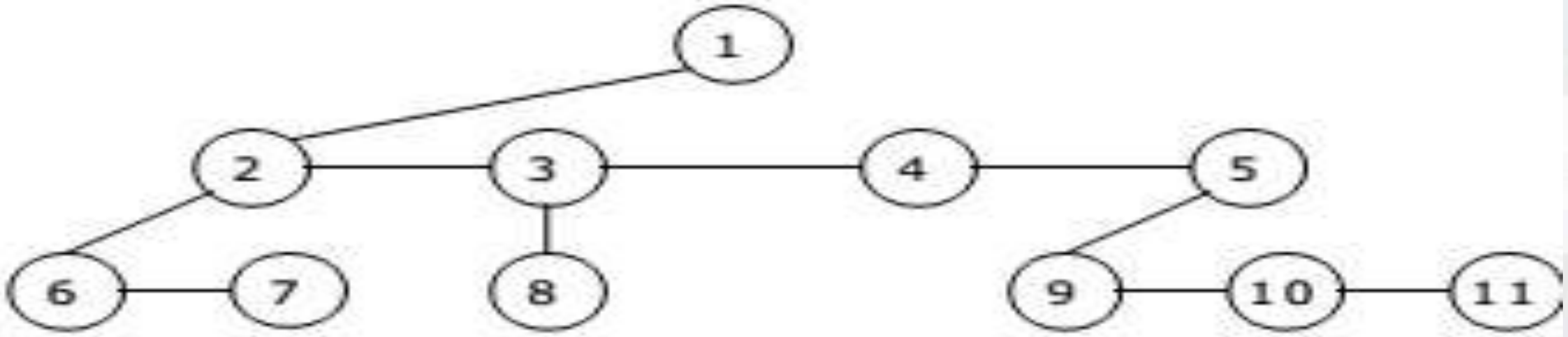
Convert the following Generic Tree to Binary Tree:



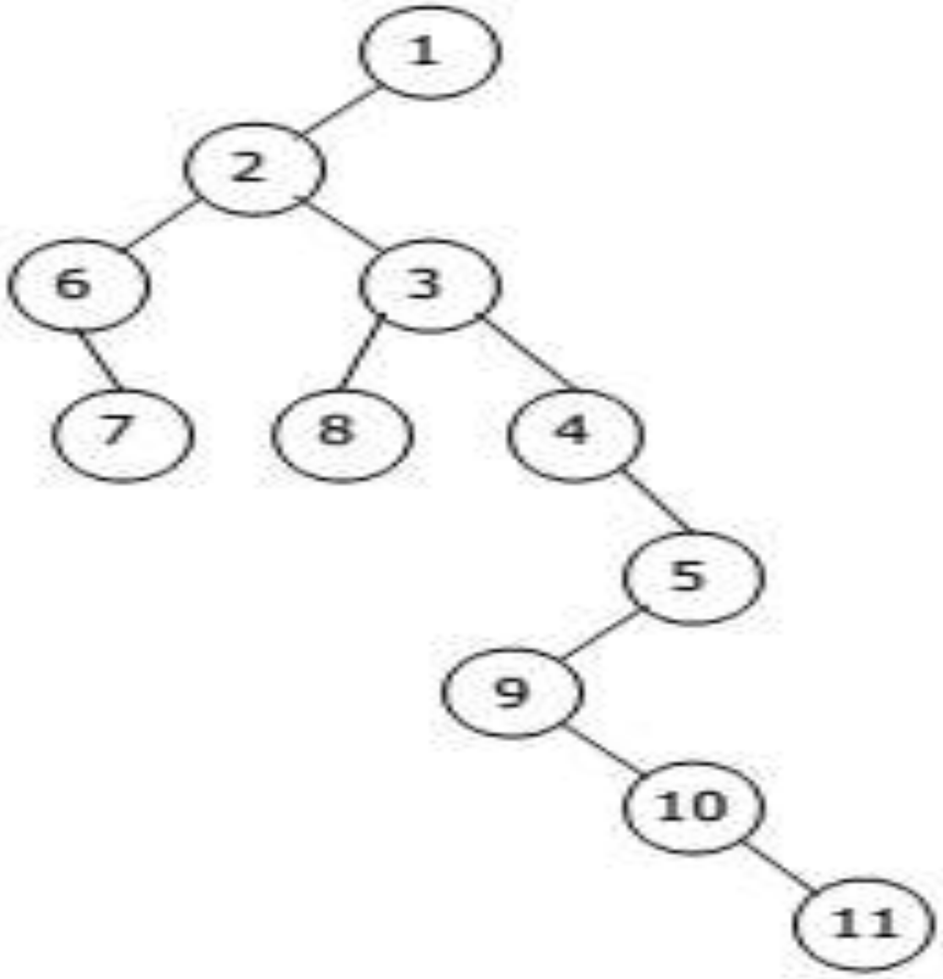
Convert a Generic Tree(N-array Tree) to Binary Tree

Solution:

Stage 1 tree by using the above mentioned procedure is as follows:



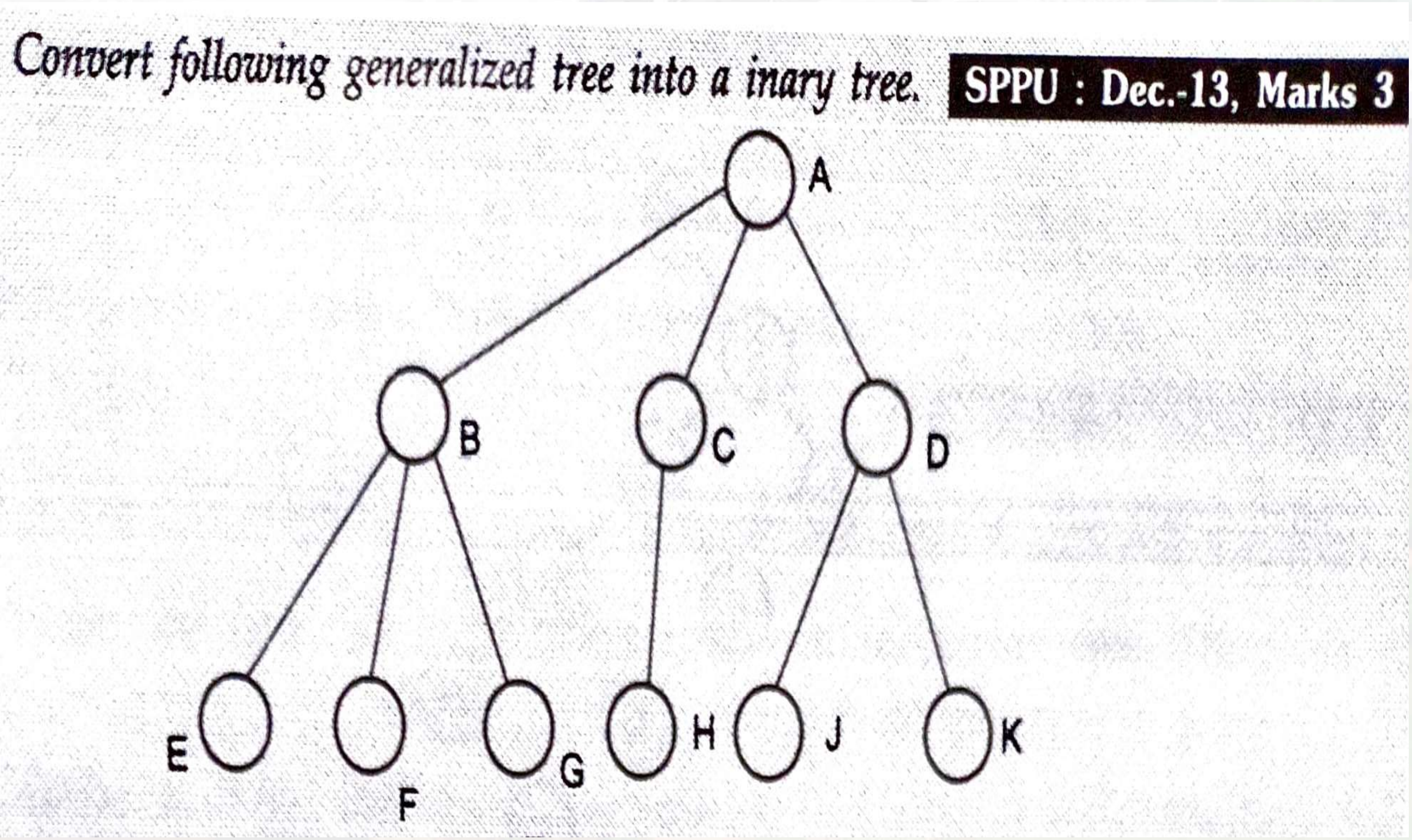
Stage 2 tree by using the above mentioned procedure is as follows:



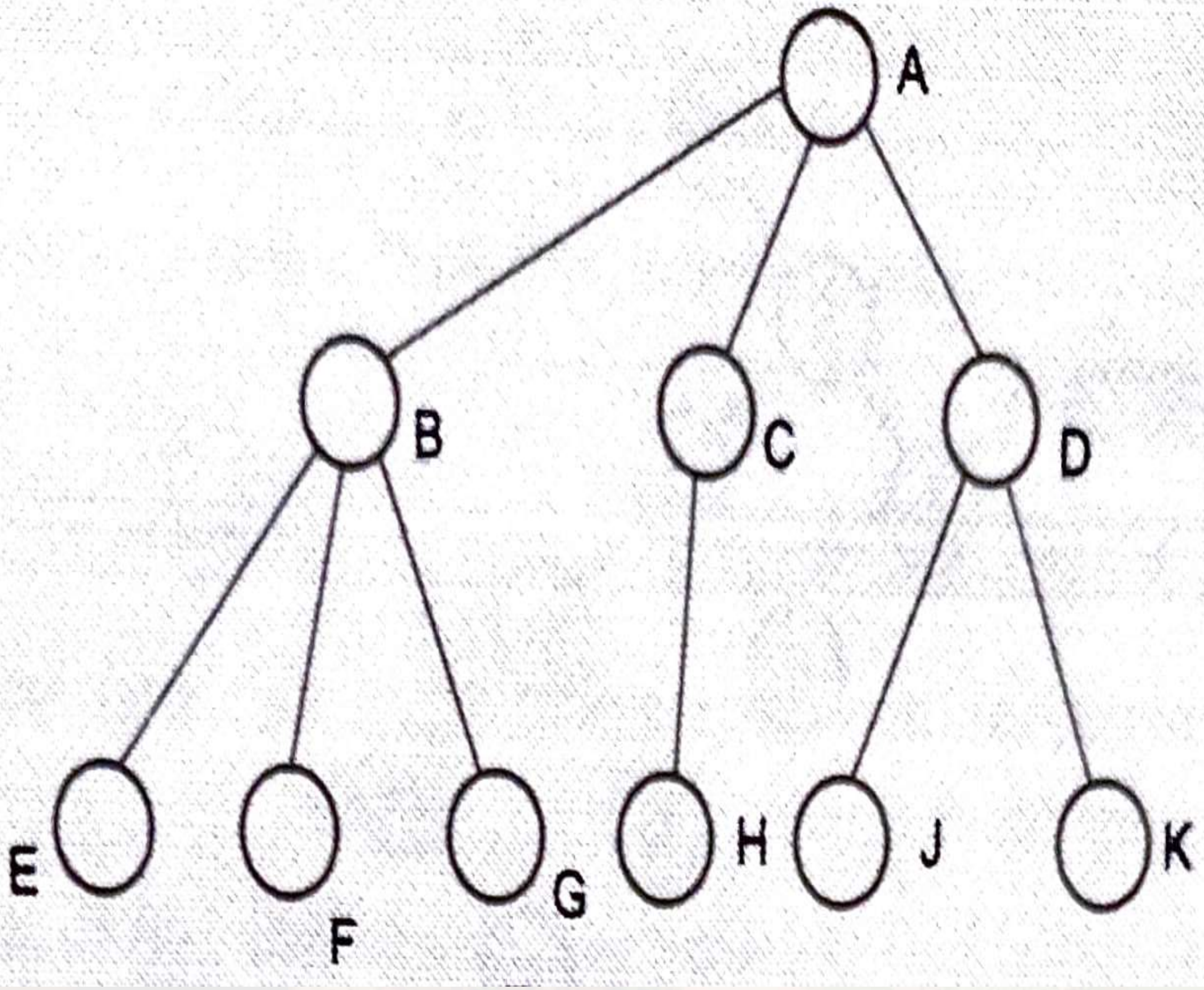
Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

Convert the following Generic Tree to Binary Tree:

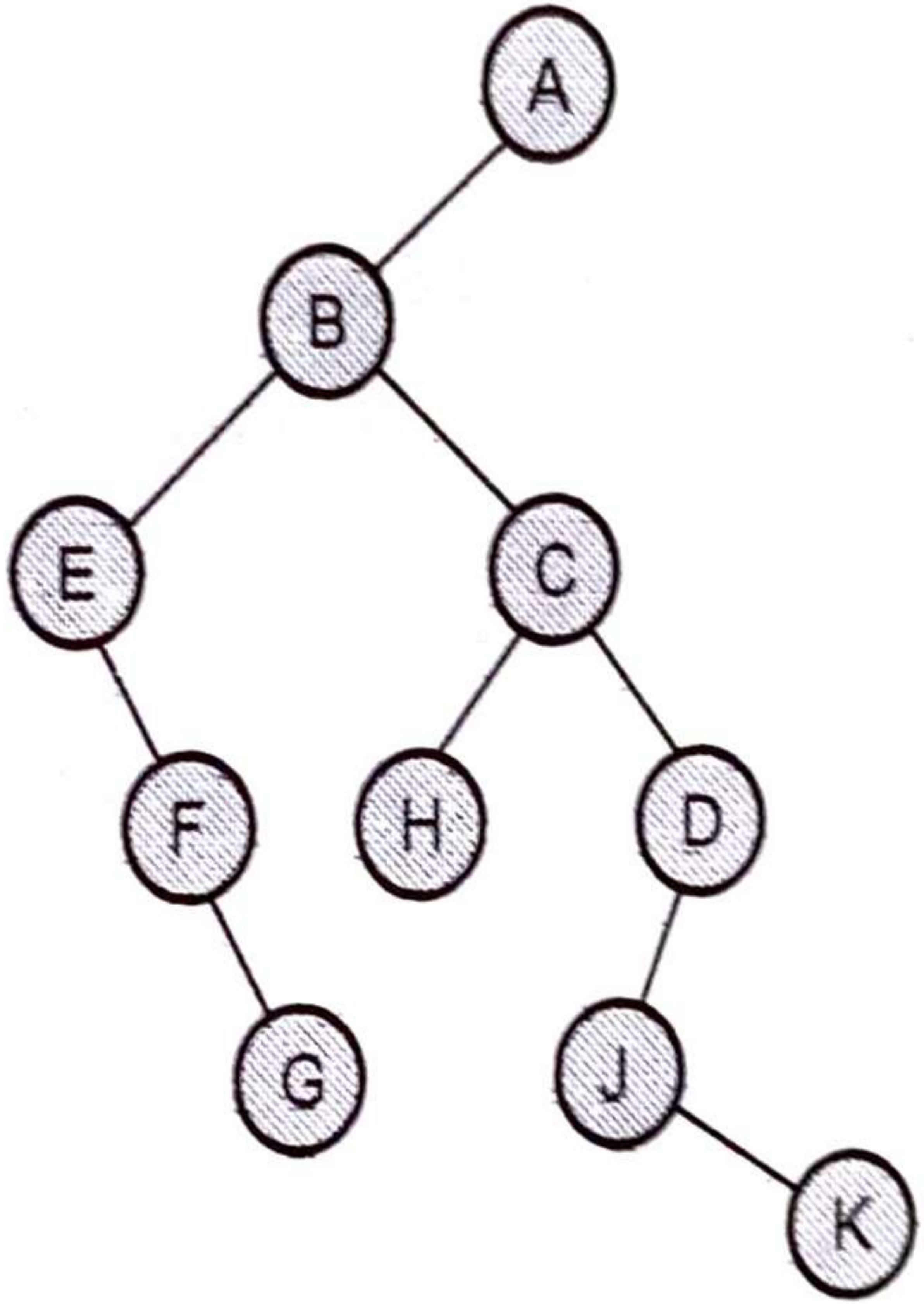


Convert following generalized tree into a inary tree. **SPPU : Dec.-13, Marks 3**



Convert a Generic Tree(N-array Tree) to Binary Tree

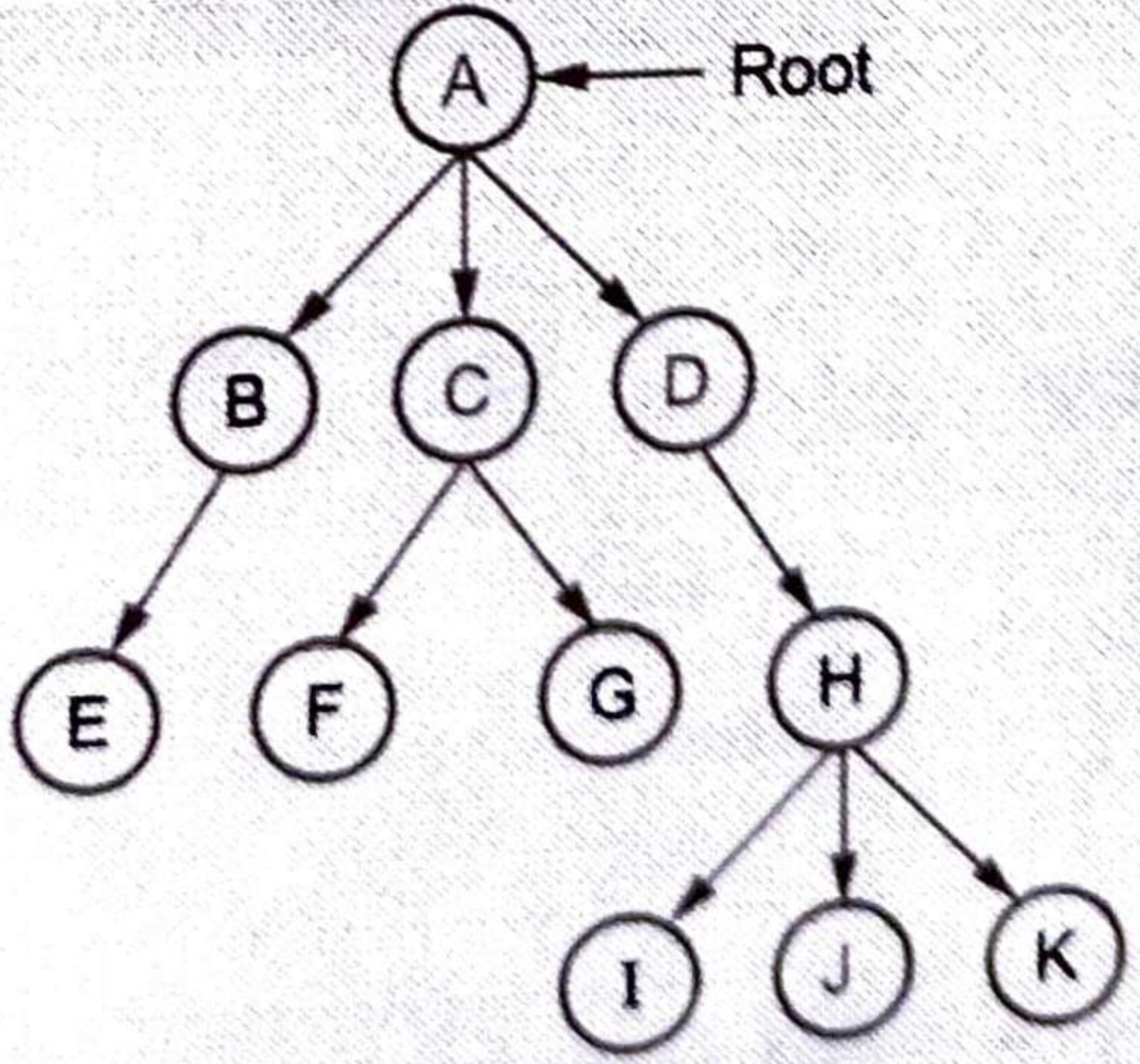
Solution :



Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

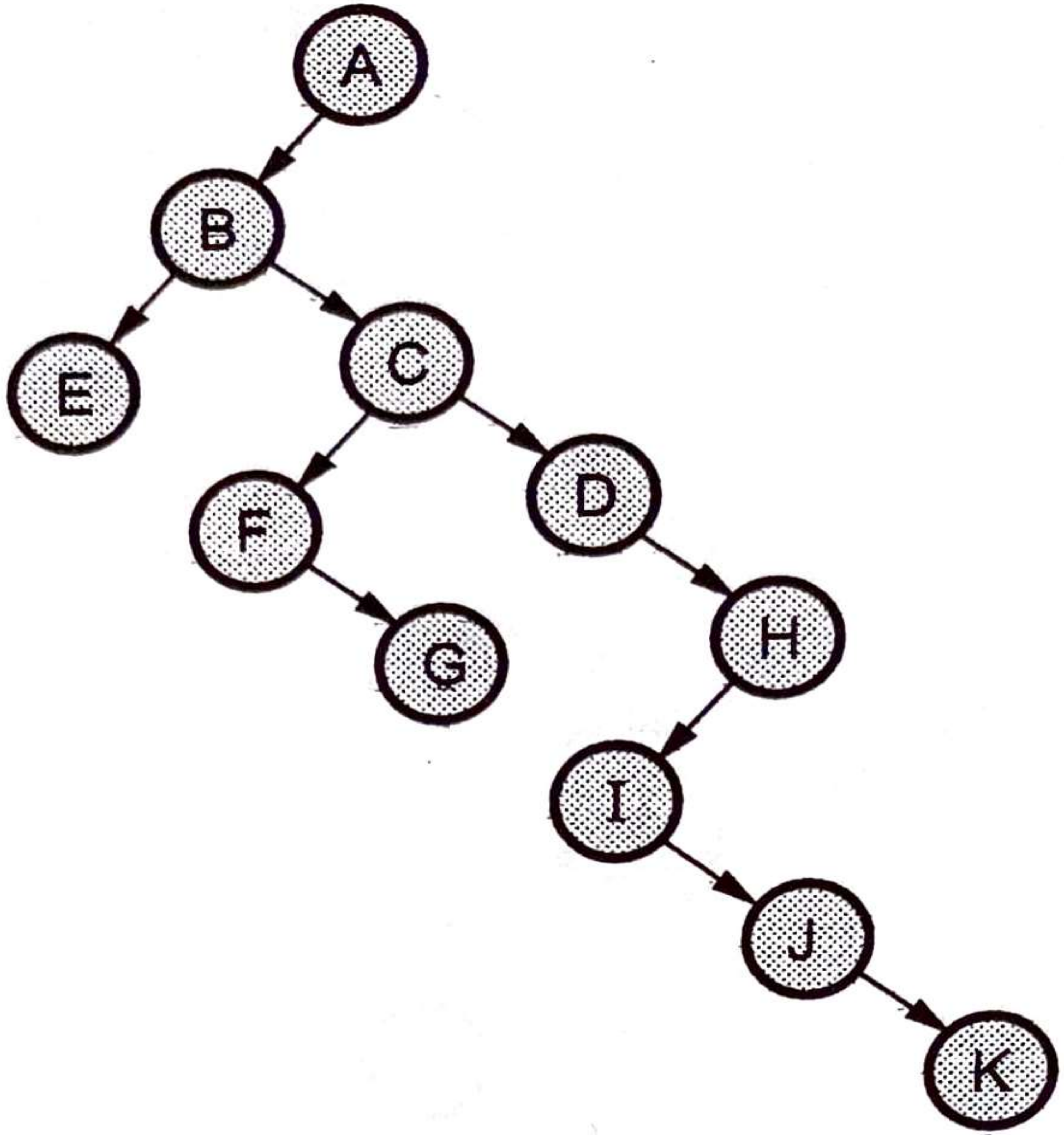
Convert the following tree into Binary tree.



Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

The binary tree will be -

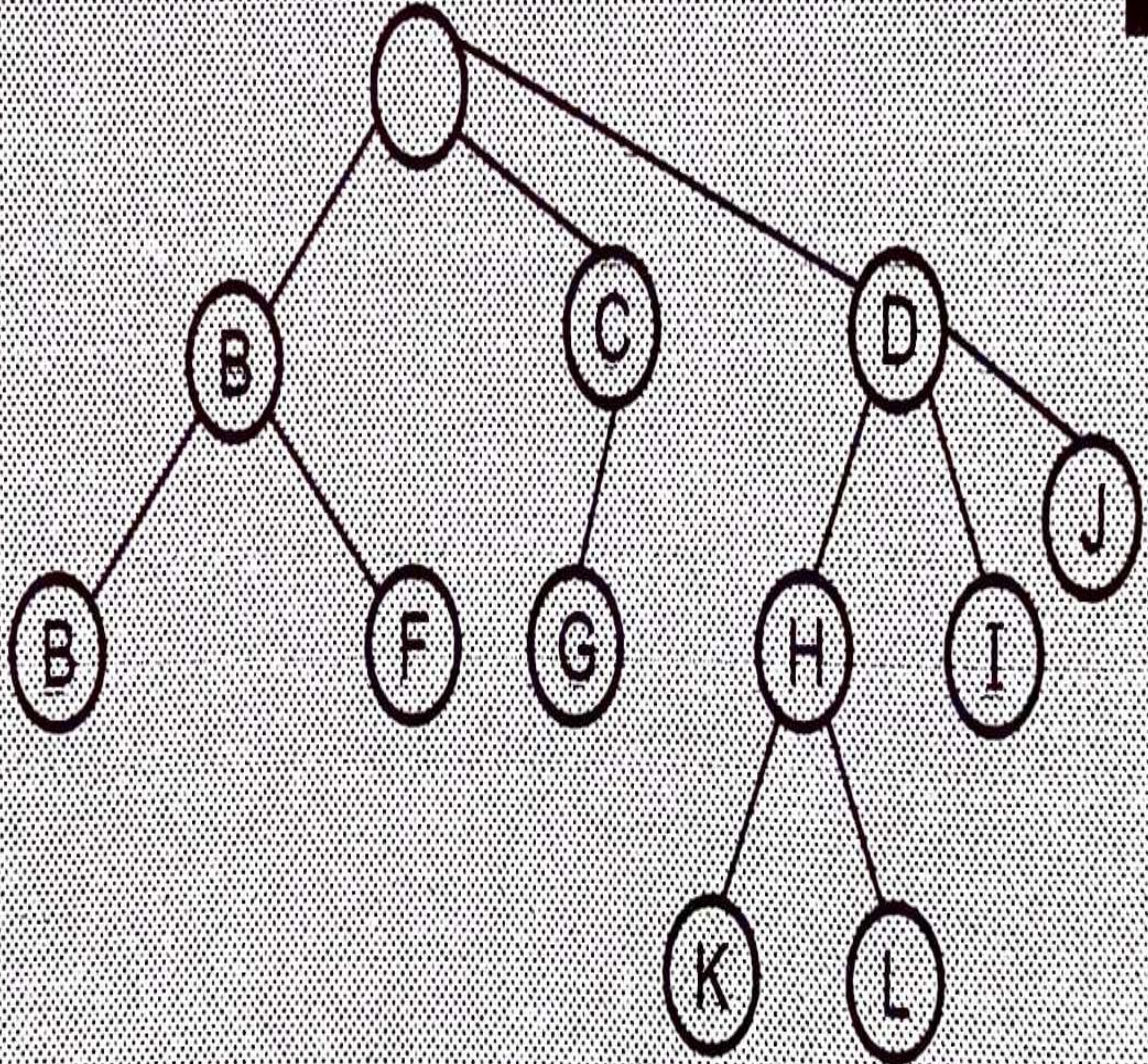


Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

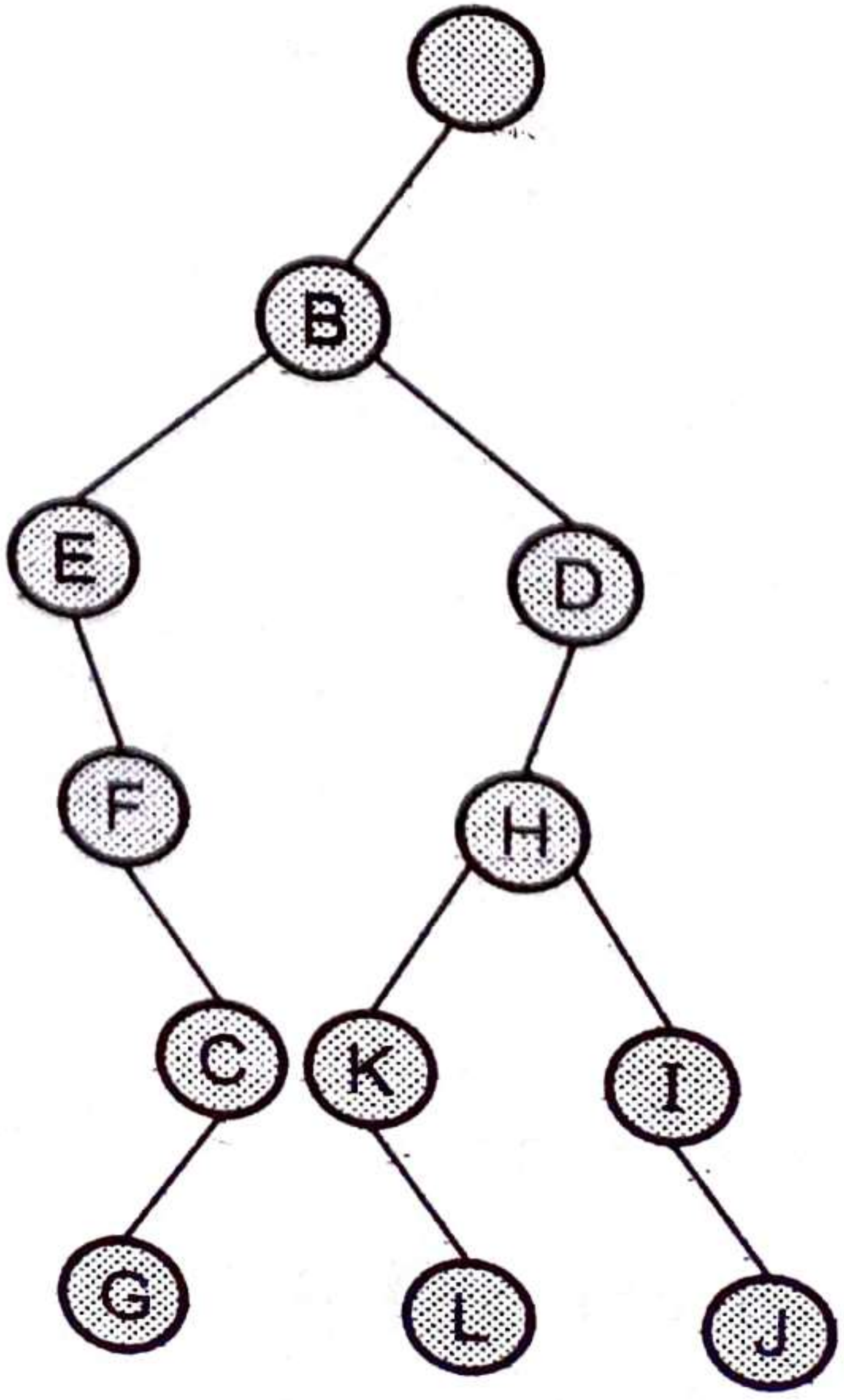
Convert the given general tree to its equivalent binary tree.

SPPU : May-19, Marks 6



Convert a Generic Tree(N-array Tree) to Binary Tree

Solution : Binary tree is



Tree Traversal

Traversal of the tree in data structures is a process of visiting each node and prints their value. There are three ways to traverse tree data structure.

1. Pre-order Traversal
2. In-Order Traversal
3. Post-order Traversal

Tree Traversal

Tree traversal means traversing or visiting each node of a tree. Linear data structures like Stack, Queue, linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node. The following are the three different ways of traversal:

1. Inorder traversal
2. Preorder traversal
3. Postorder traversal

Inorder Traversal

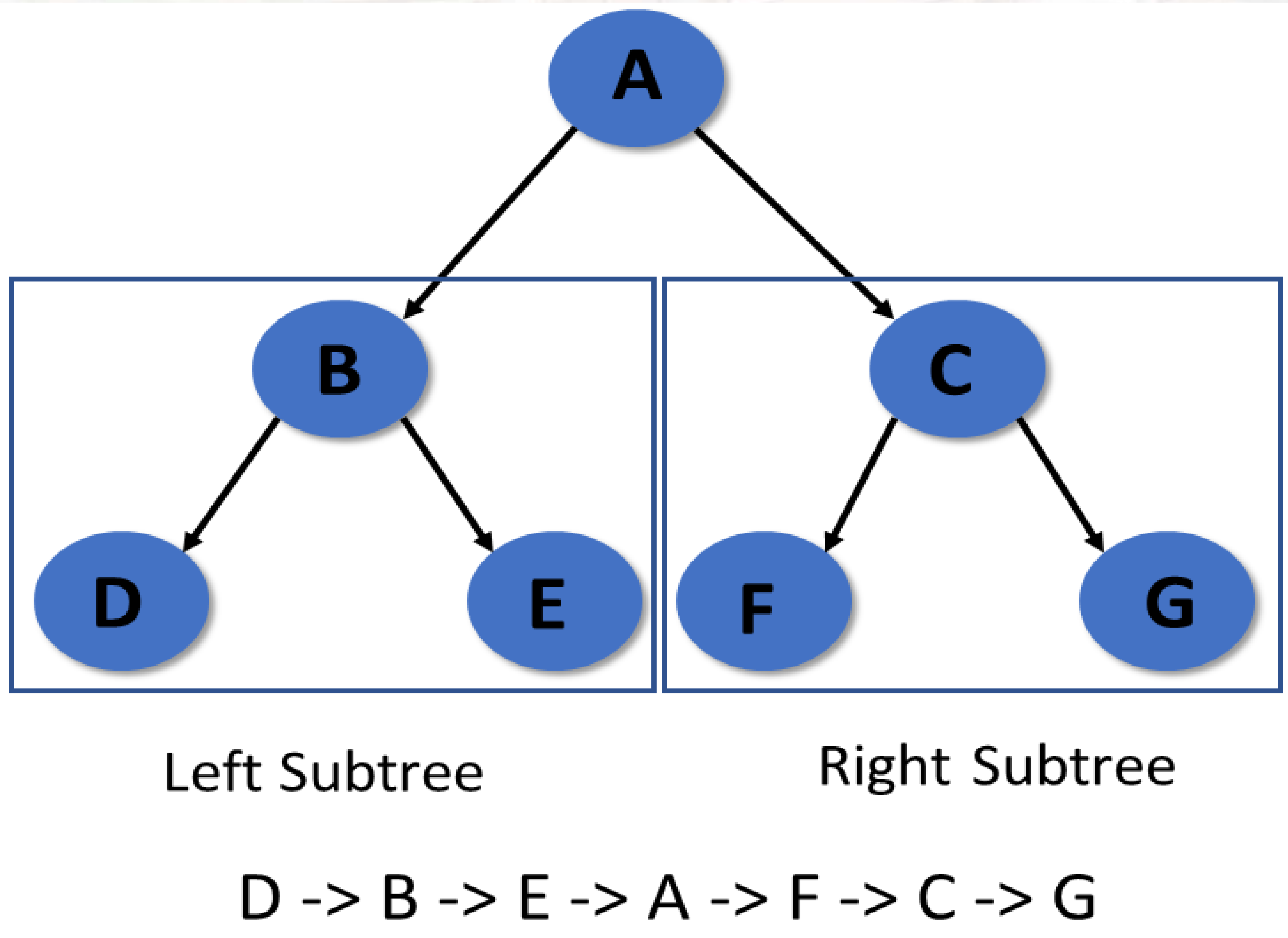
In the in-order traversal, the left subtree is visited first, then the root, and later the right subtree.

Algorithm:

Step 1- Recursively traverse the left subtree

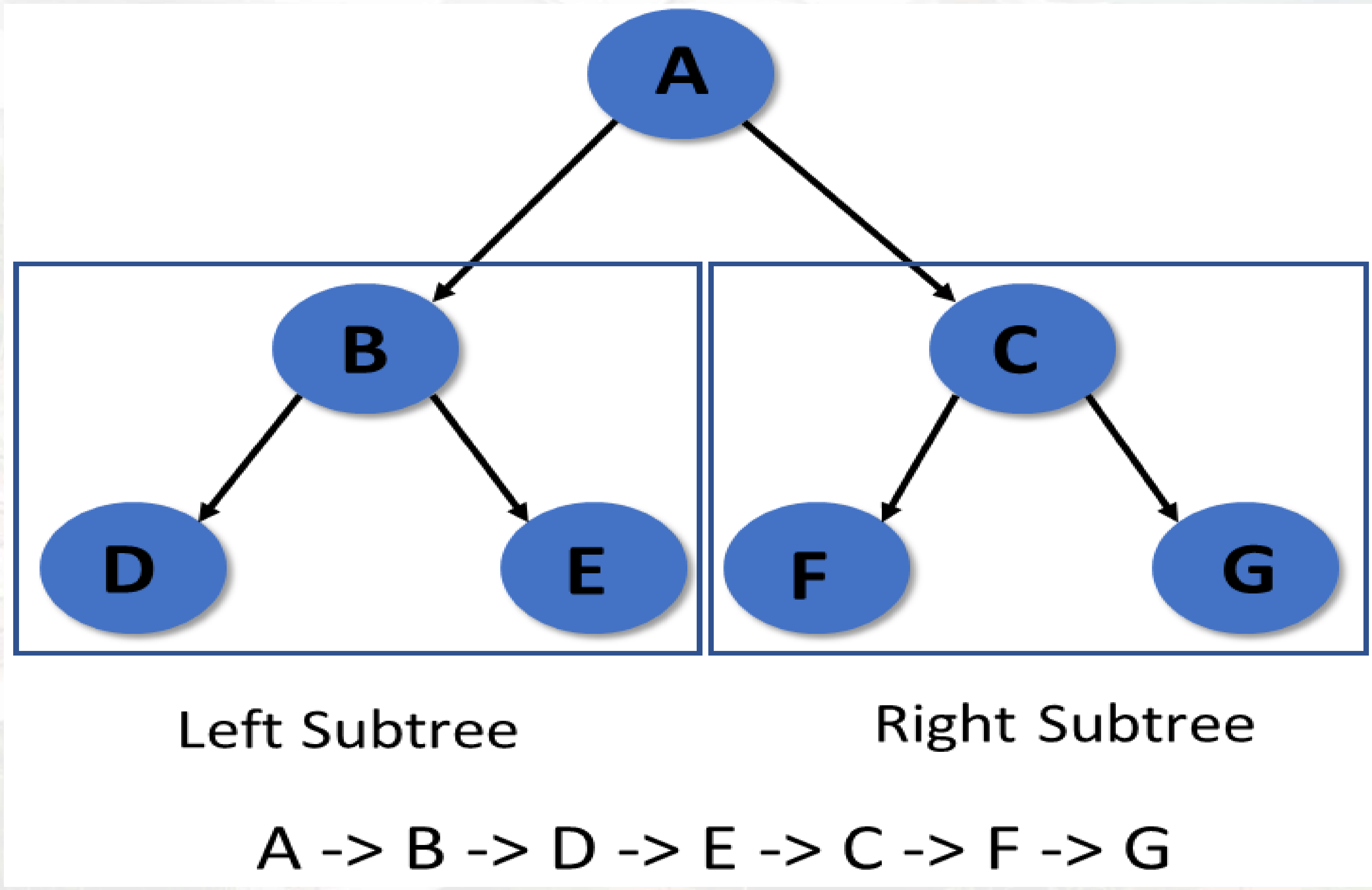
Step 2- Visit root node

Step 3- Recursively traverse right



Pre-order Traversal

In pre-order traversal, it visits the root node first, then the left subtree, and lastly right subtree.



Algorithm:

Step 1- Visit root node

Step 2- Recursively traverse the left subtree

Step 3- Recursively traverse right subtree

Post-order Traversal

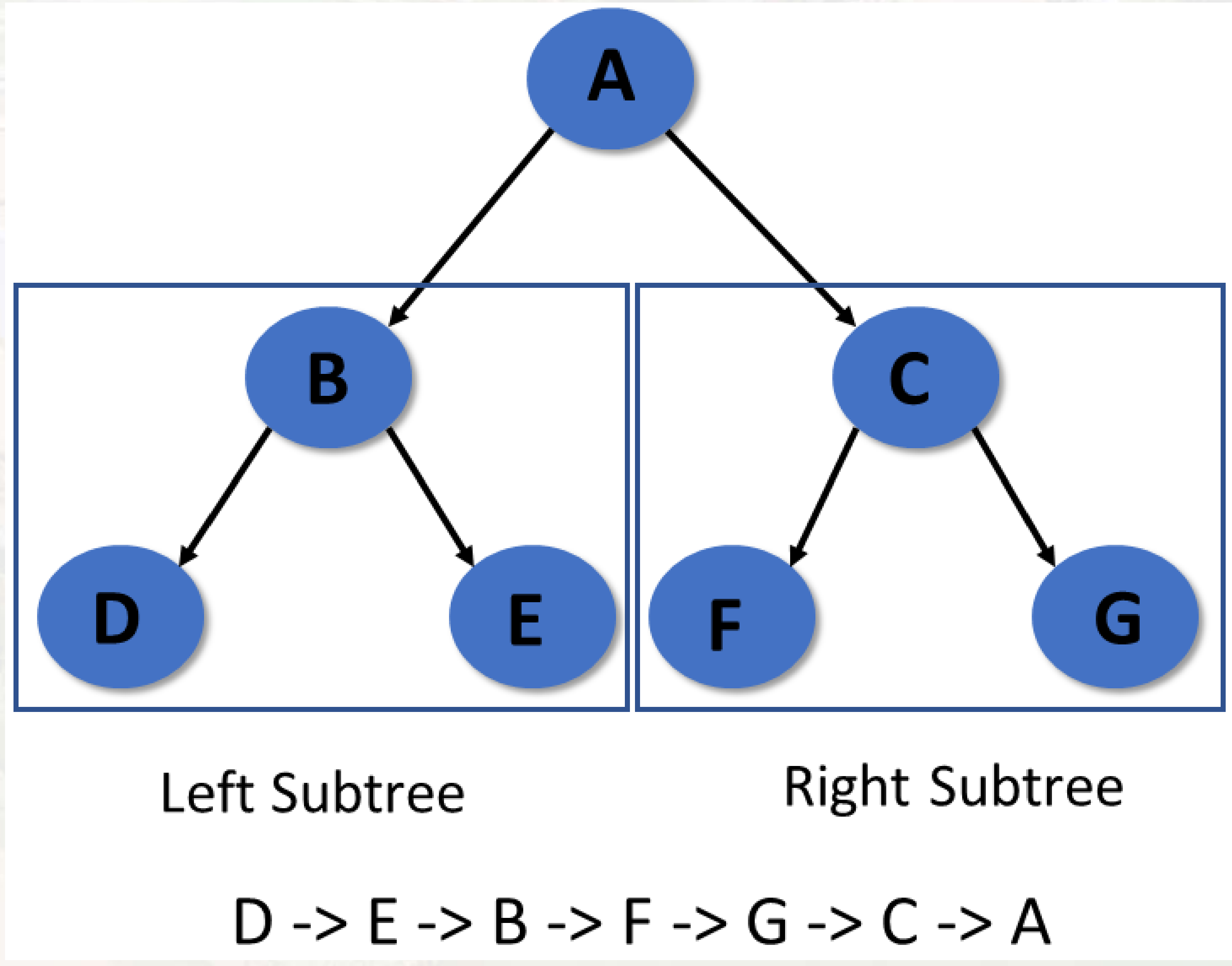
It visits the left subtree first in post-order traversal, then the right subtree, and finally the root node.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Recursively traverse right

Step 3- Visit root node



Tree Traversal Example

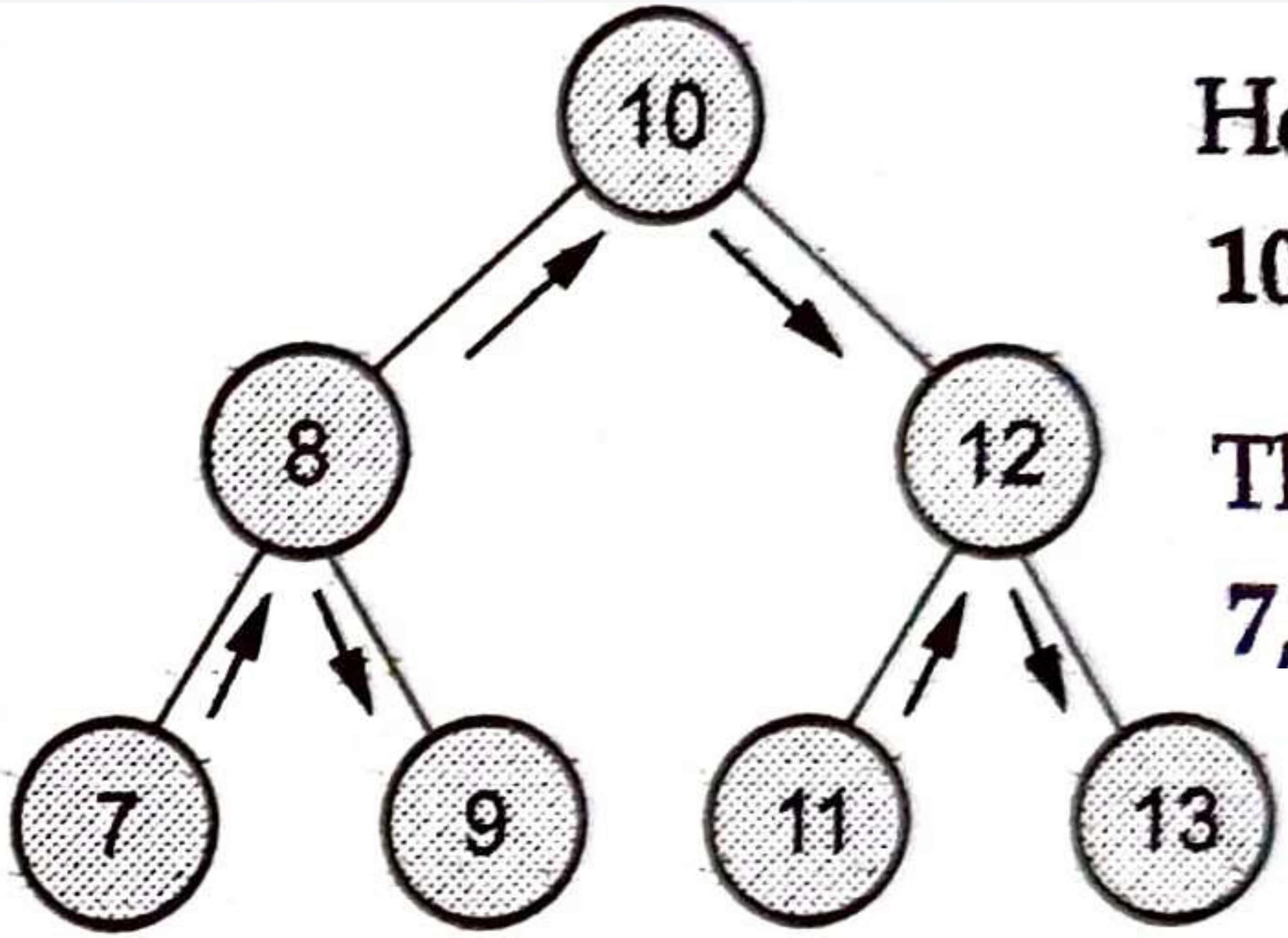


Fig. 2.6.1 Binary tree

Hence, Preorder traversal is

10, 8, 7, 9, 12, 11, 13.

The Postorder sequence is

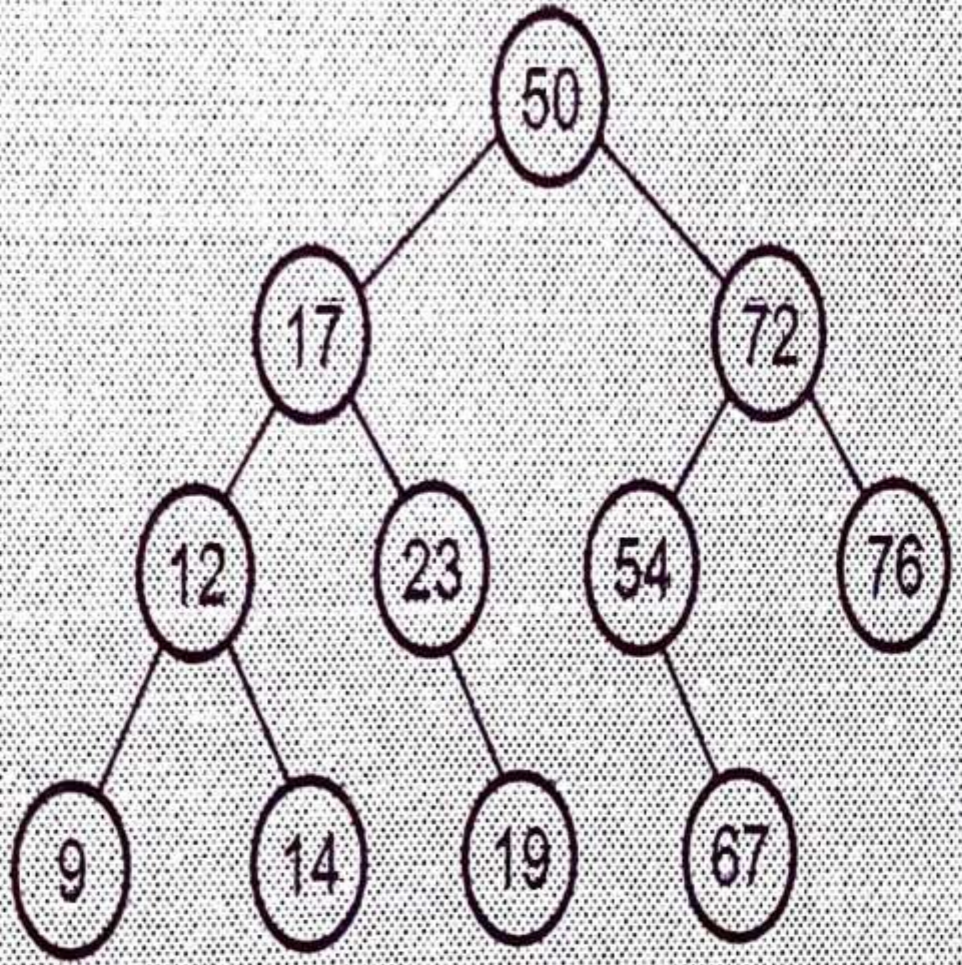
7, 9, 8, 11, 13, 12, 10.

Inorder sequence is 7, 8, 9, 10, 11, 12, 13

Tree Traversal Example

Example 2.6.1 *Consider the following tree given in the problem. Show a Postorder, Preorder and In order Traversal of tree.*

SPPU : May-14, Marks 3



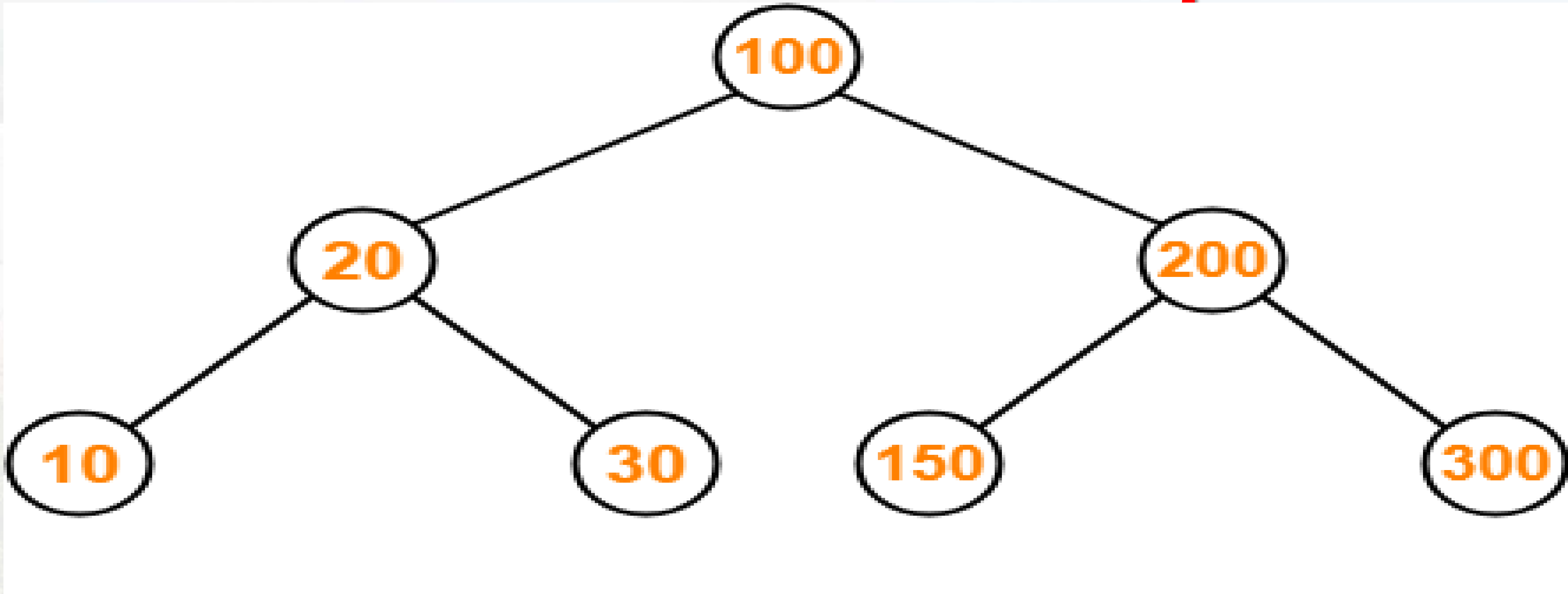
SPECIMEN CO

Solution : Preorder Sequence : 50, 17, 12, 9, 14, 23, 19, 72, 54, 67, 76.

Inorder Sequence : 9, 12, 14, 17, 23, 19, 50, 54, 67, 72, 76.

Postorder Sequence : 9, 14, 12, 19, 23, 17, 67, 54, 76, 72, 50.

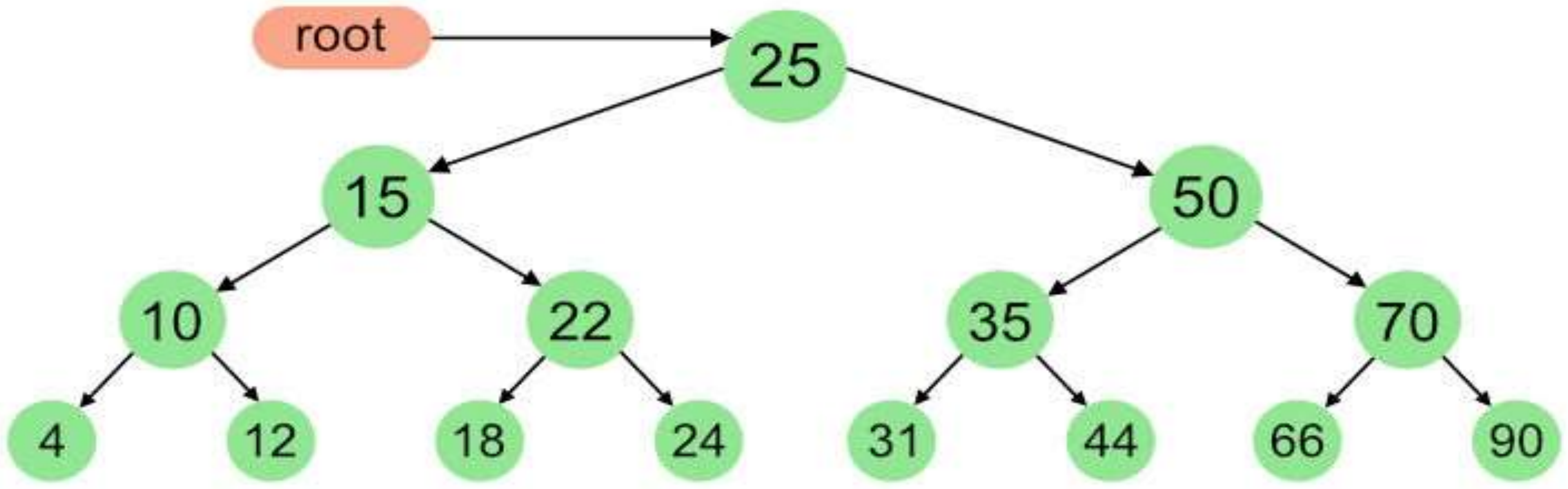
Tree Traversal Example



Binary Search Tree

Preorder Traversal -	100 , 20 , 10 , 30 , 200 , 150 , 300
Inorder Traversal-	10 , 20 , 30 , 100 , 150 , 200 , 300
Postorder Traversal :	10 , 30 , 20 , 150 , 300 , 200 , 100

Tree Traversal Example



InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

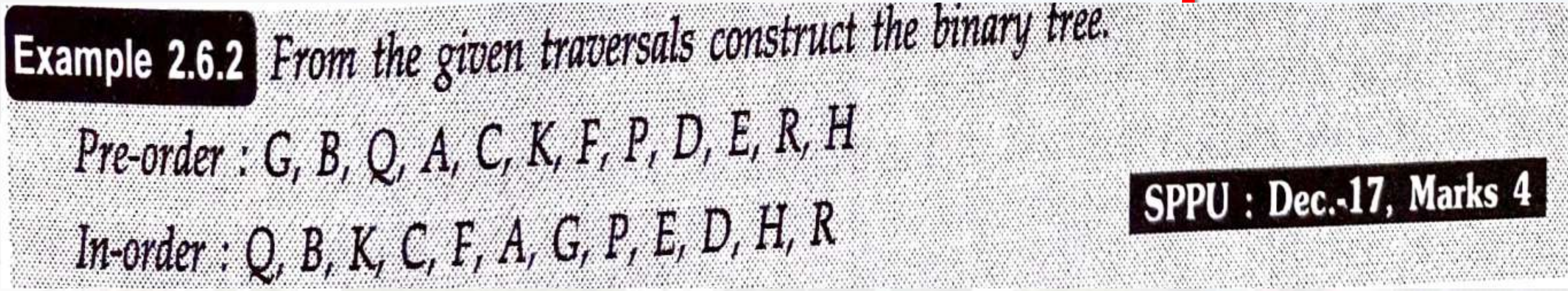
A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Tree Traversal Example

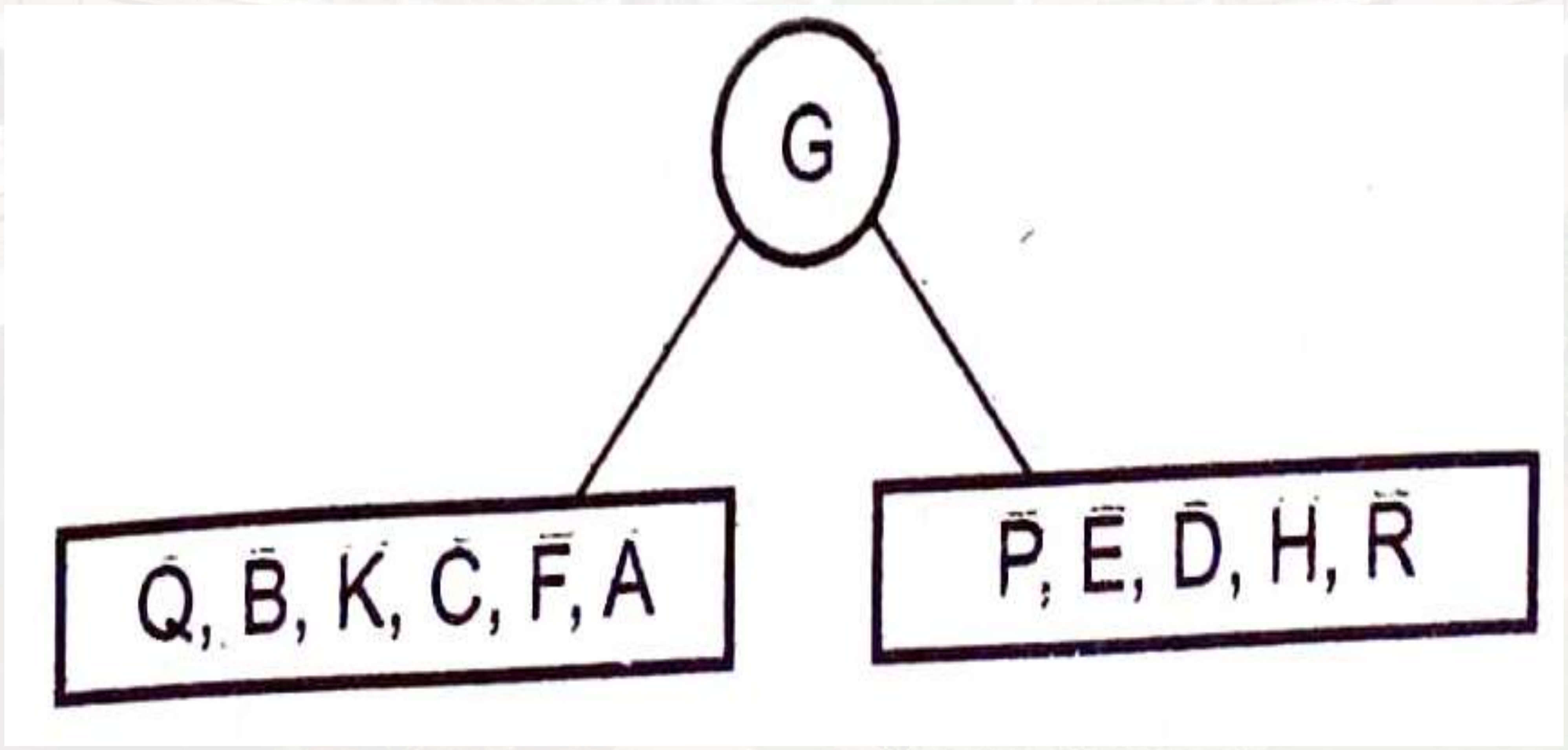


Example 2.6.2 *From the given traversals construct the binary tree.*

SPPU : Dec.-17, Marks 4

Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

In-order : Q, B, K, C, F, A, G, P, E, D, H, R



Tree Traversal Example

Example 2.6.2 *From the given traversals construct the binary tree.*

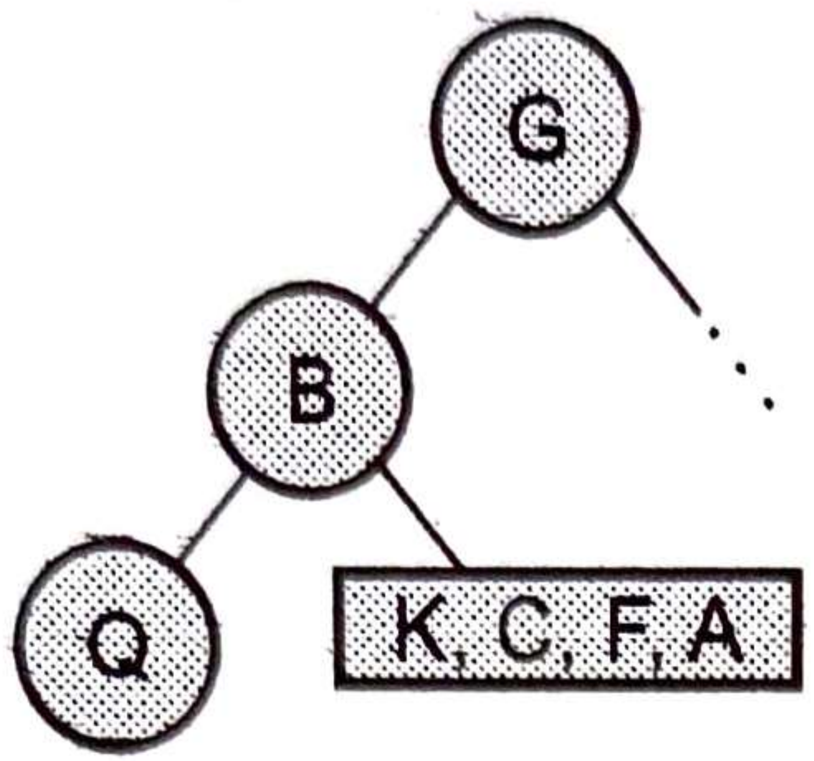
Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

SPPU : Dec.-17, Marks 4

In-order : Q, B, K, C, F, A, G, P, E, D, H, R

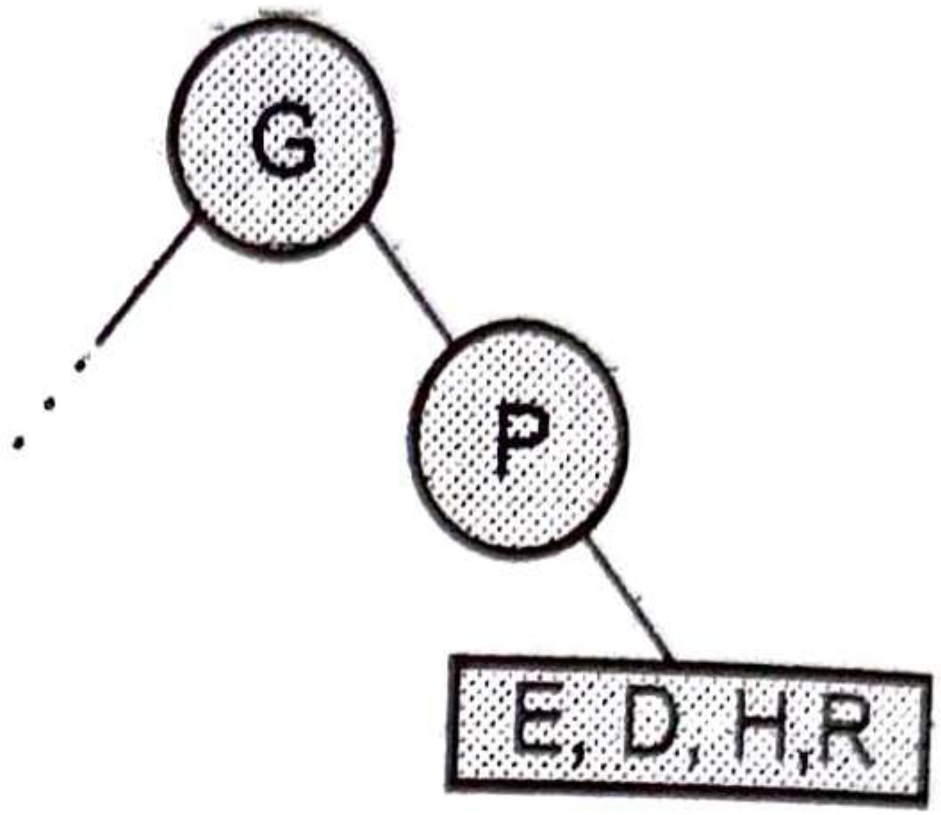
Preorder : $\boxed{B}$, Q, A, C, K, F,

Inorder : Q, $\boxed{B}$, K, C, F, A



Preorder : $\boxed{P}$, D, E, R, H,

Inorder : $\boxed{P}$, E, D, H, R,



Tree Traversal Example

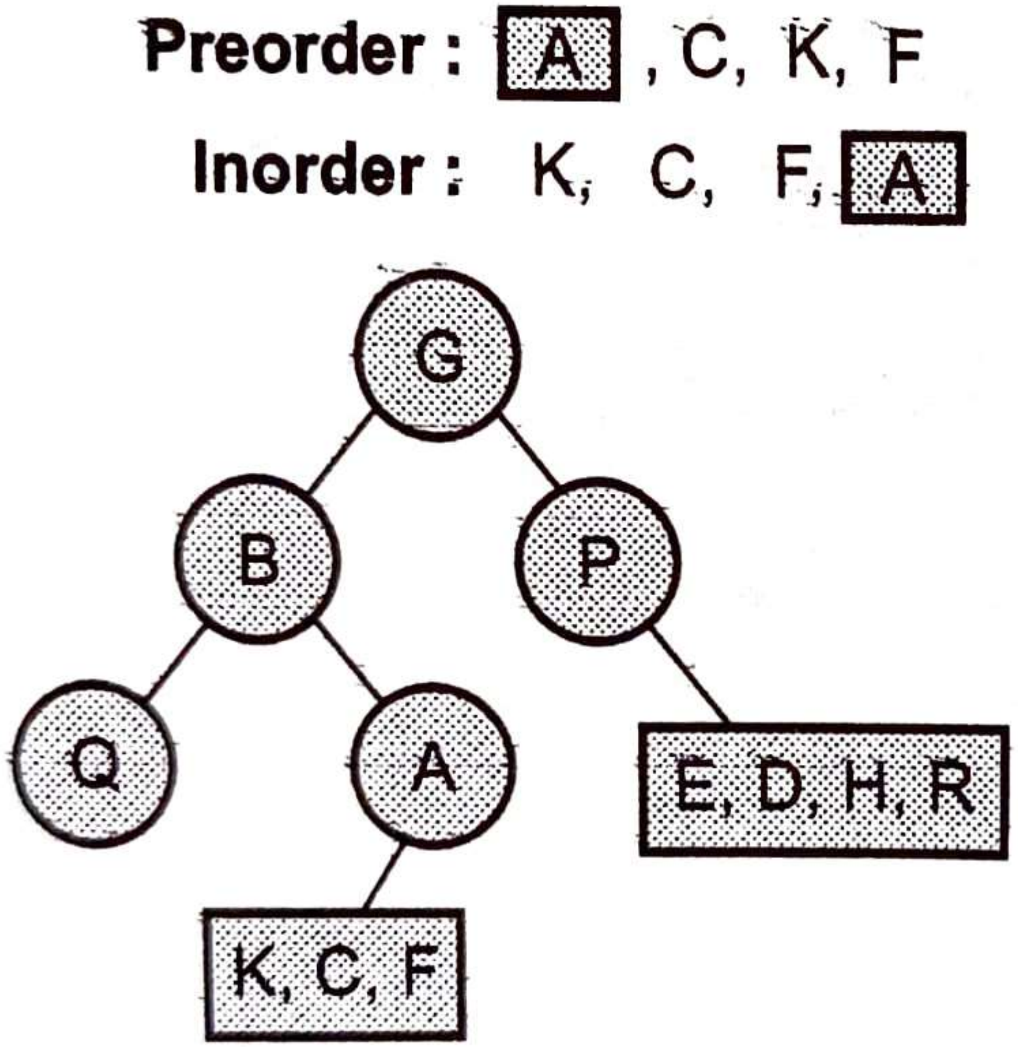
Example 2.6.2 *From the given traversals construct the binary tree.*

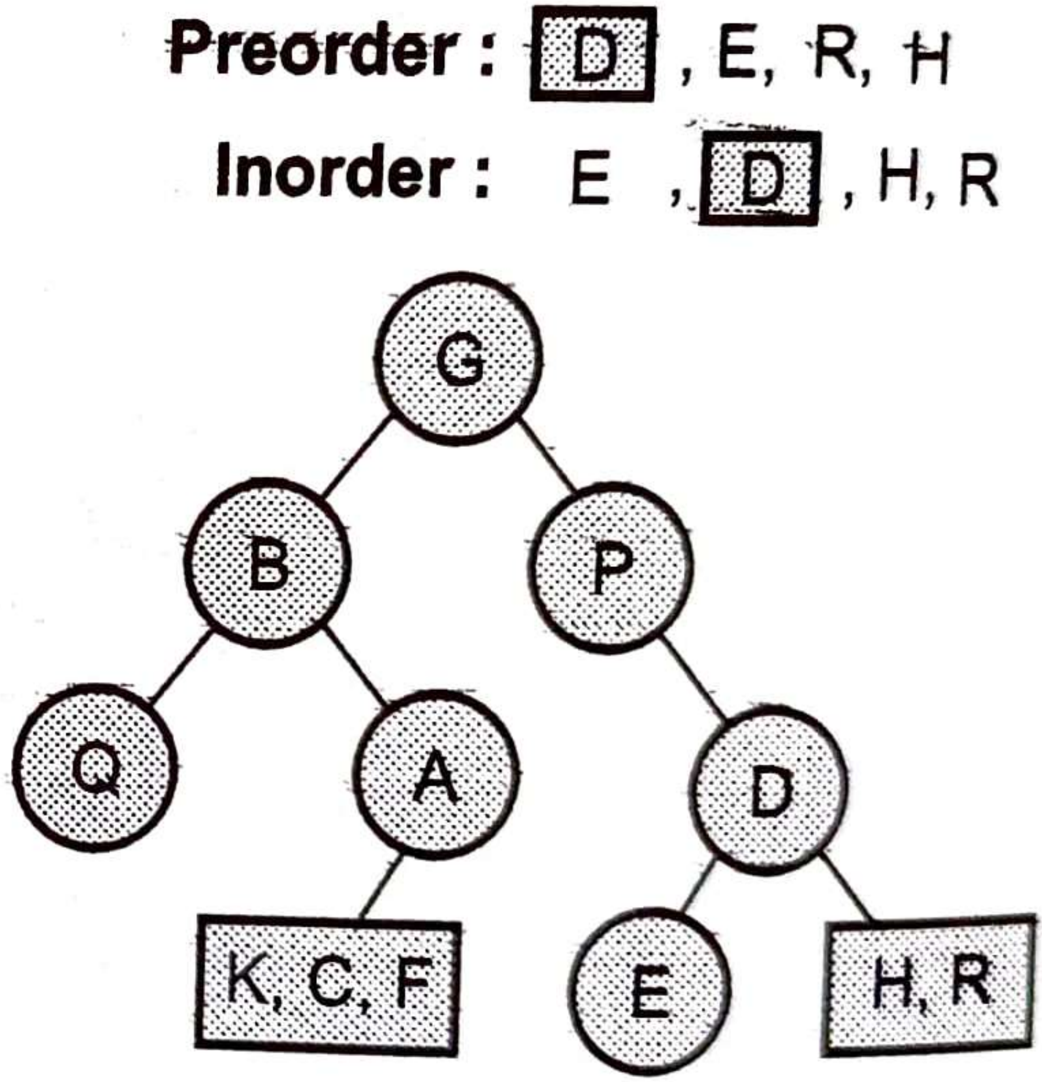
Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

In-order : Q, B, K, C, F, A, G, P, E, D, H, R

SPPU : Dec.-17, Marks 4

Step 3 :





Tree Traversal Example

Example 2.6.2 *From the given traversals construct the binary tree.*

Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

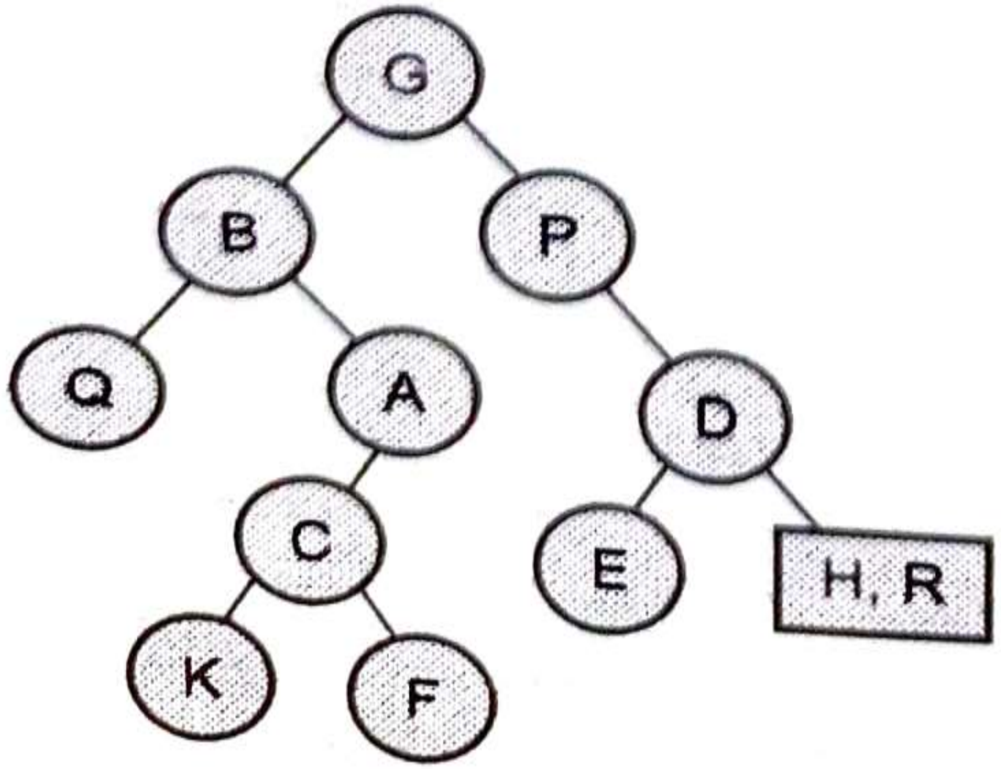
In-order : Q, B, K, C, F, A, G, P, E, D, H, R

SPPU : Dec.-17, Marks 4

Step 4 :

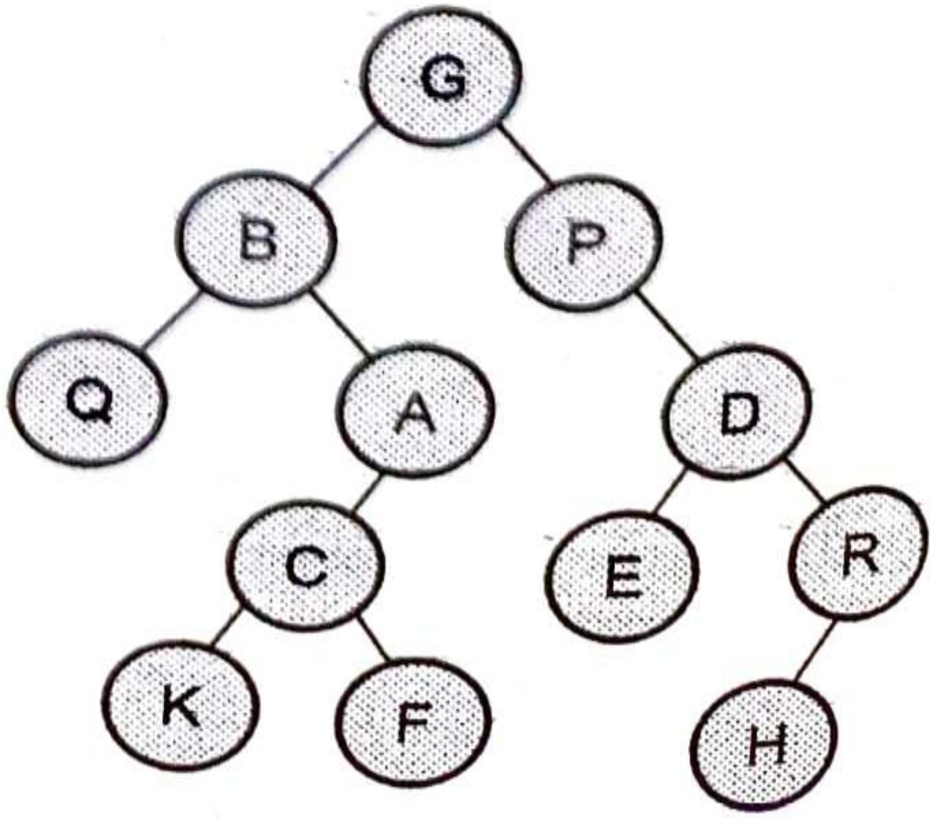
Preorder : $\boxed{C}$, K, F,

Inorder : K, $\boxed{C}$, F,



Preorder : $\boxed{R}$, H,

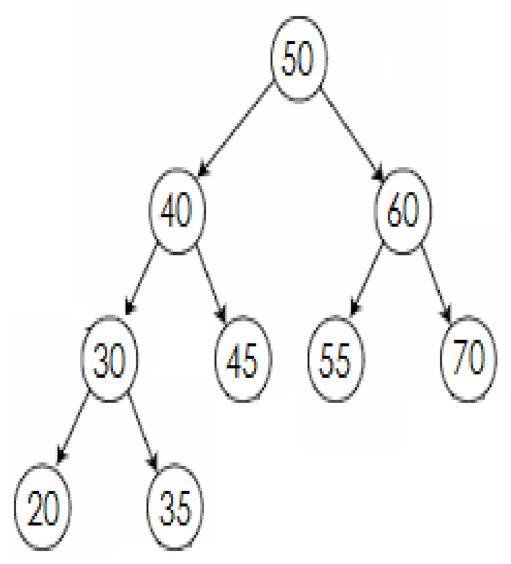
Inorder : H, $\boxed{R}$

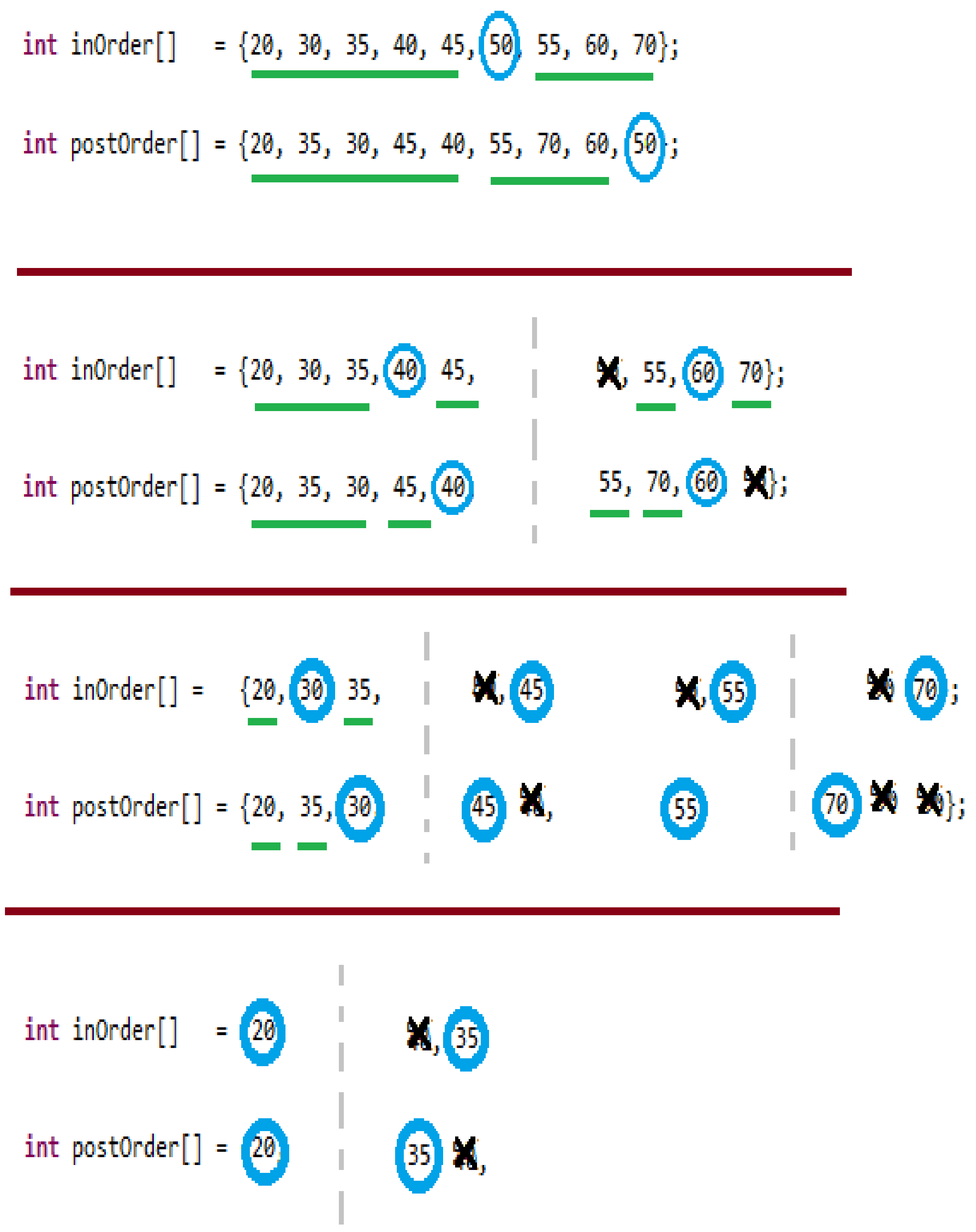


Tree Traversal Example

```
int inOrder[] =   {20, 30, 35, 40, 45, 50, 55, 60, 70};
int postOrder[] = {20, 35, 30, 45, 40, 55, 70, 60, 50};
```

Binary Tree





int inOrder[] = {20, 30, 35, 40, 45, 50, 55, 60, 70};

int postOrder[] = {20, 35, 30, 45, 40, 55, 70, 60, 50};

int inOrder[] = {20, 30, 35, 40, 45, | ~~50~~, 55, 60, 70};

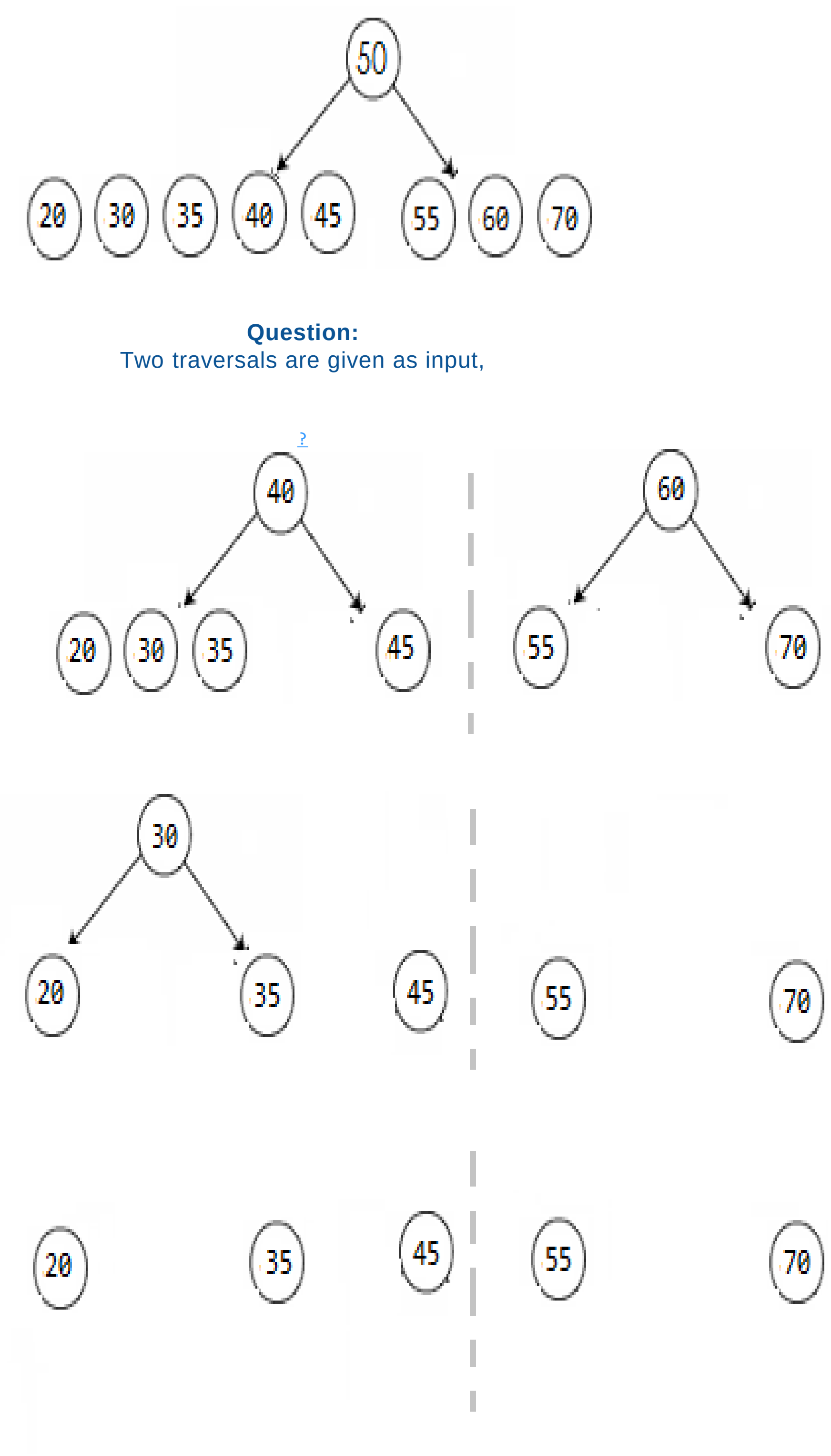
int postOrder[] = {20, 35, 30, 45, 40 | 55, 70, 60, ~~50~~};

int inOrder[] = {20, 30, 35, | ~~40~~, 45 | ~~50~~, 55 | ~~60~~ 70;

int postOrder[] = {20, 35, 30 | 45 ~~40~~, | 55 | 70 ~~60~~ ~~50~~};

int inOrder[] = 20 | ~~30~~, 35

int postOrder[] = 20 | 35 ~~30~~,



Question:
Two traversals are given as input,

Tree Traversal Example

Example Input:

Postorder = [10, 18, 9, 22, 4]

Inorder = [10, 4, 18, 22, 9]

Example :

preorder = [1,2,4,5,3,6,7],

postorder = [4,5,2,6,7,3,1]

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf(“%d”, ptr->data);
        indorder(ptr->right_child);
    }
}
```

A / B * C * D + E

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf(“%d”, ptr->data);
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postdorder(ptr->right_child);
        printf(“%d”, ptr->data);
    }
}
```

A B / C * D * E +

DFS Algorithms

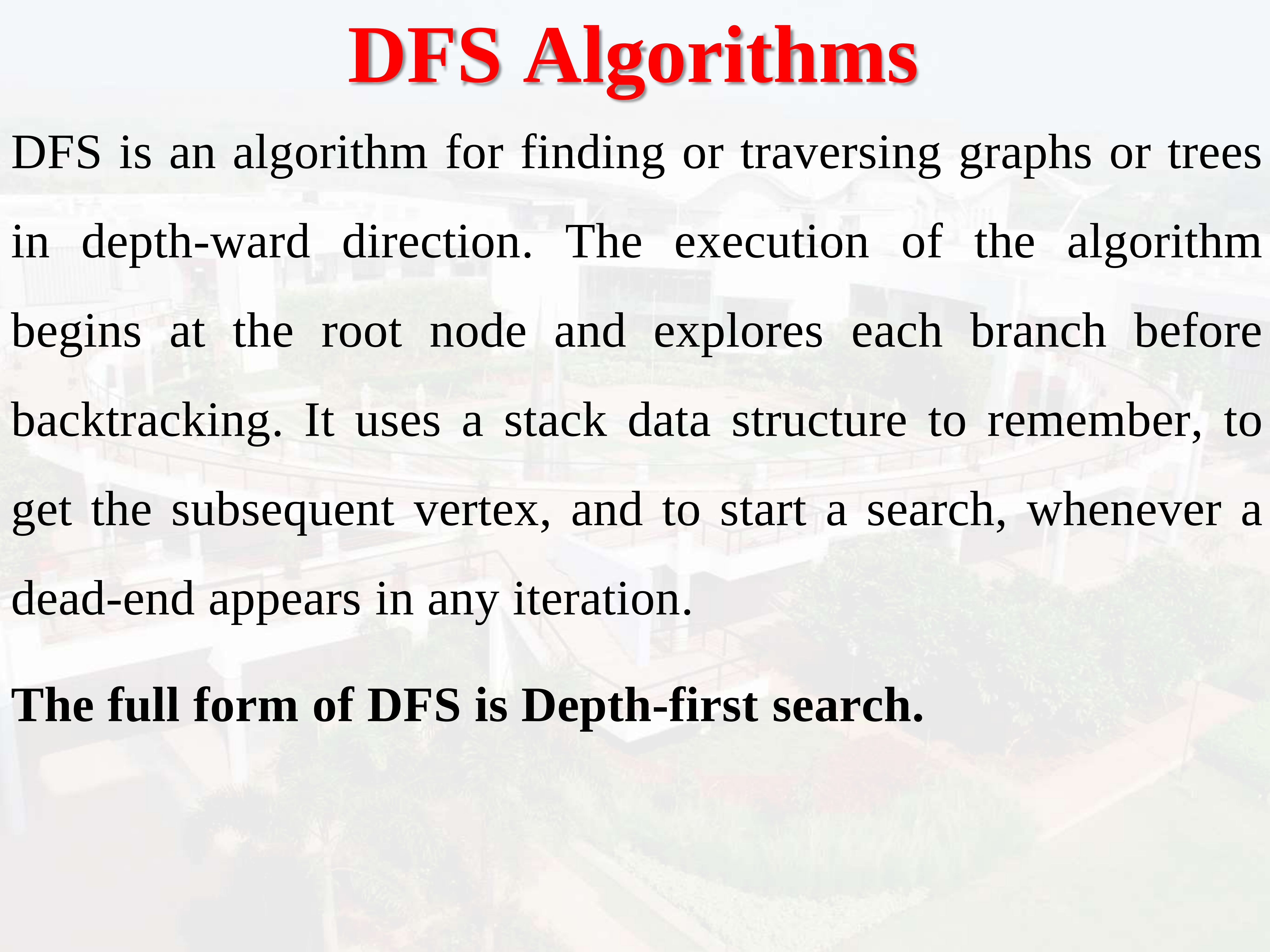
Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

DFS Algorithms

DFS is an algorithm for finding or traversing graphs or trees in depth-ward direction. The execution of the algorithm begins at the root node and explores each branch before backtracking. It uses a stack data structure to remember, to get the subsequent vertex, and to start a search, whenever a dead-end appears in any iteration.

The full form of DFS is Depth-first search.

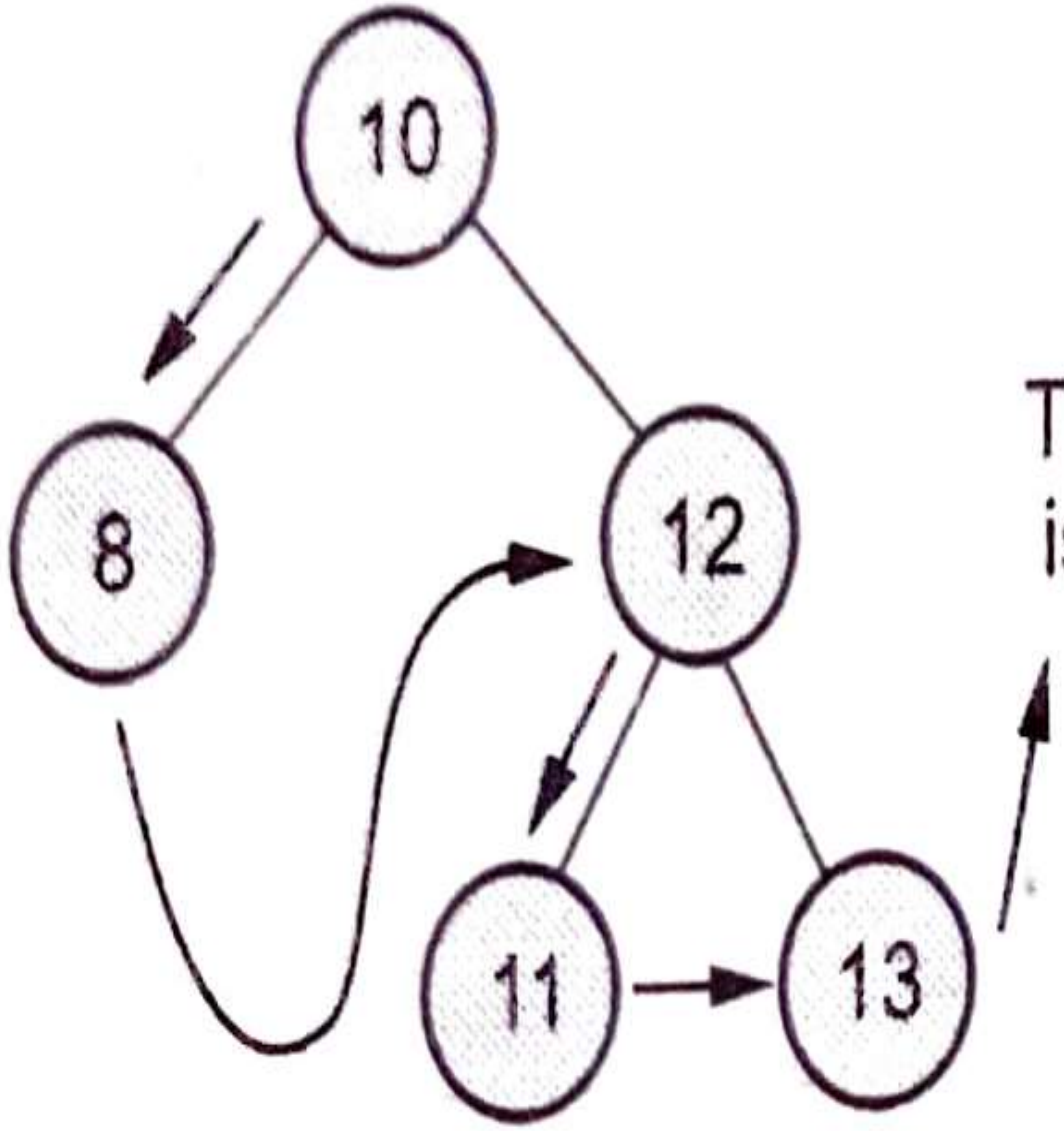


DFS Algorithms

The step by step process to implement the DFS traversal is given as follows -

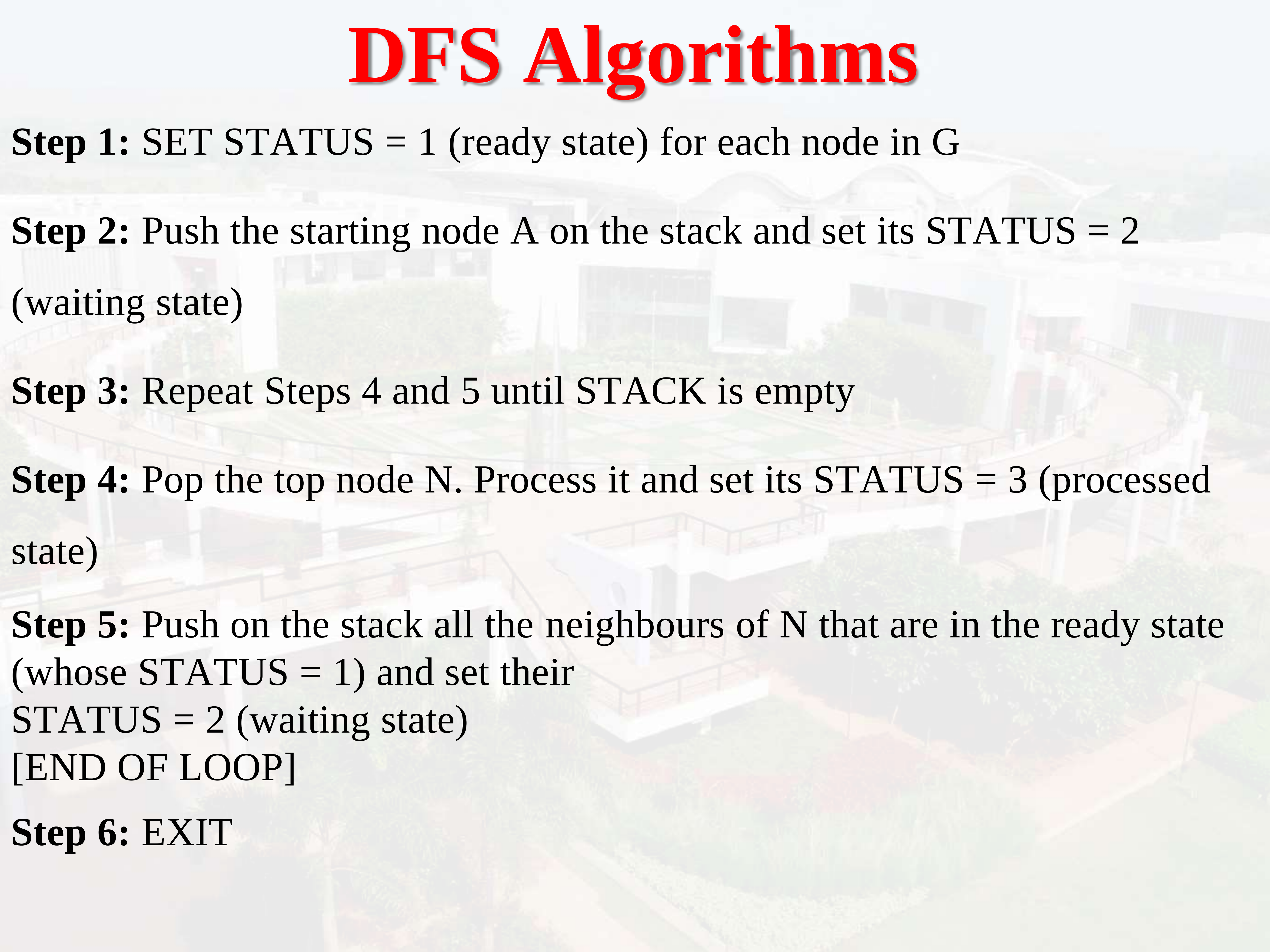
1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

DFS Algorithms



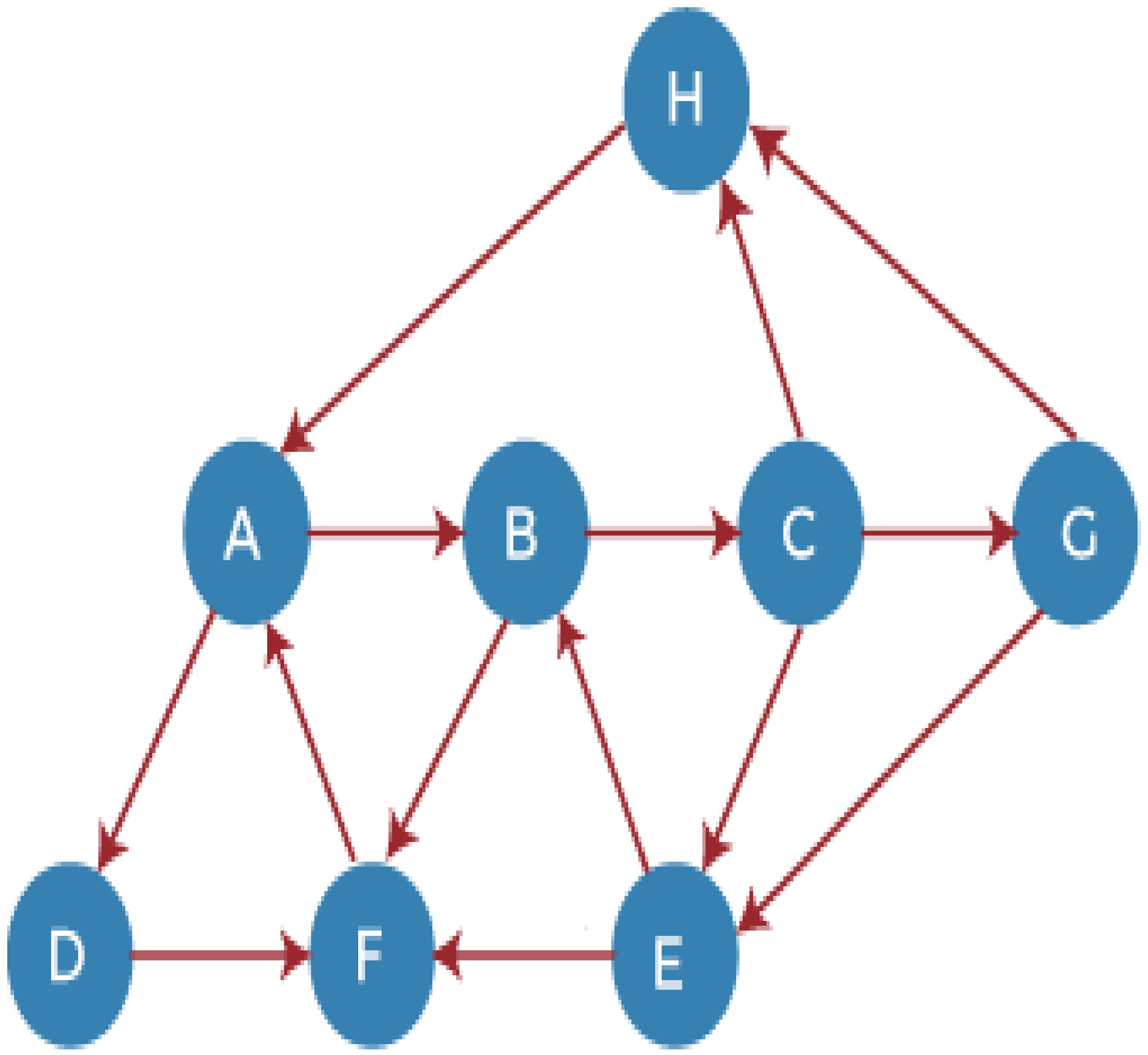
The DFS sequence is **10, 8, 12, 11, 13**

DFS Algorithms



```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2
(waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed
state)
Step 5: Push on the stack all the neighbours of N that are in the ready state
(whose STATUS = 1) and set their
STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

DFS Algorithms Example



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

Complexity of DFS Algorithms

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

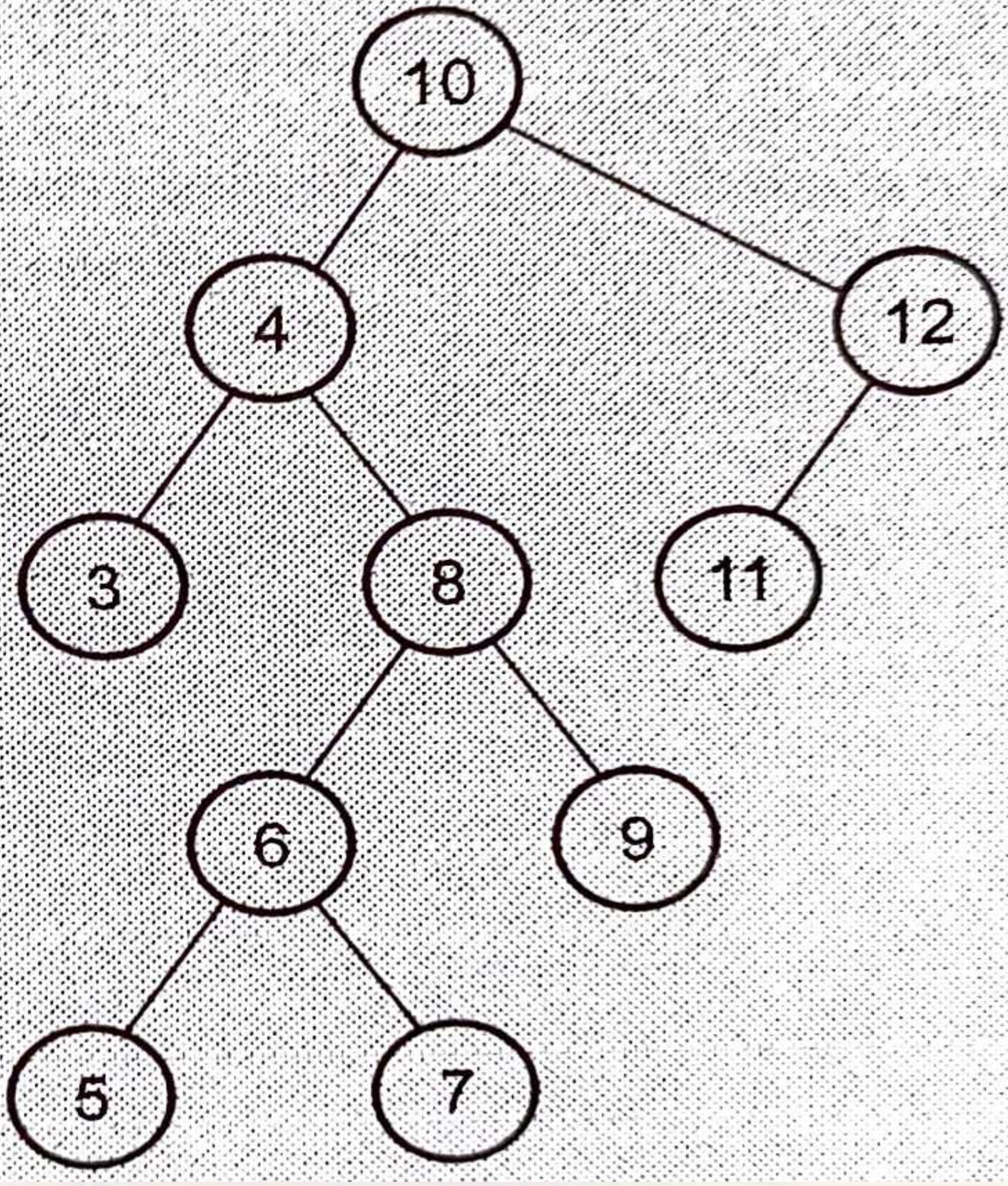
The space complexity of the algorithm is O(V).

Application of DFS Algorithms

1. DFS algorithm can be used to implement the topological sorting.
2. It can be used to find the paths between two vertices.
3. It can also be used to detect cycles in the graph.
4. DFS algorithm is also used for one solution puzzles.
5. DFS is used to determine if a graph is bipartite or not.

Example of DFS Algorithms

Example 2.7.1 *Consider following tree and obtain its DFS sequence.*



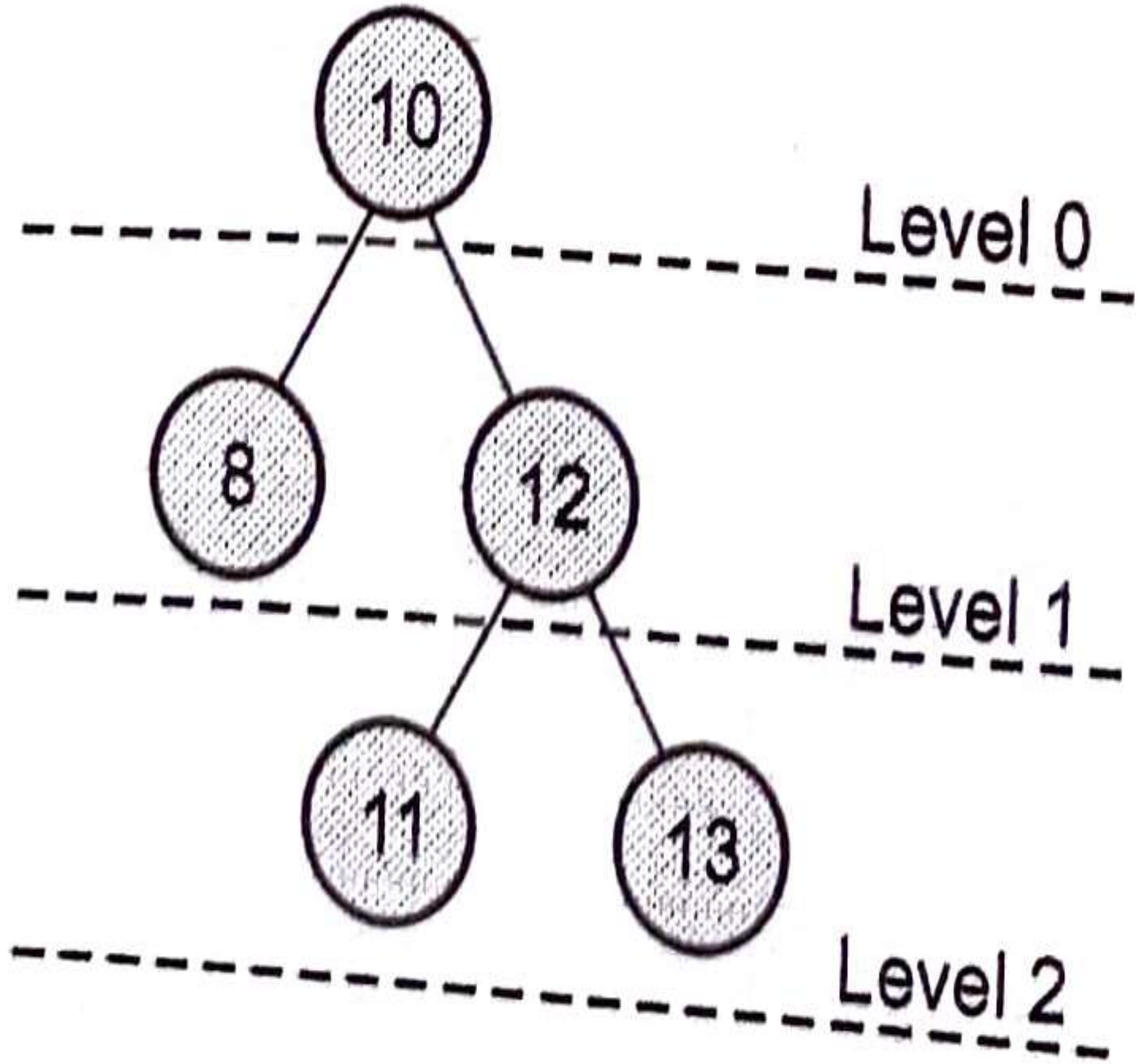
10, 4, 3, 8, 6, 5, 7, 9, 12, 11

BFS Algorithms

1. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unvisited nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

2. BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node

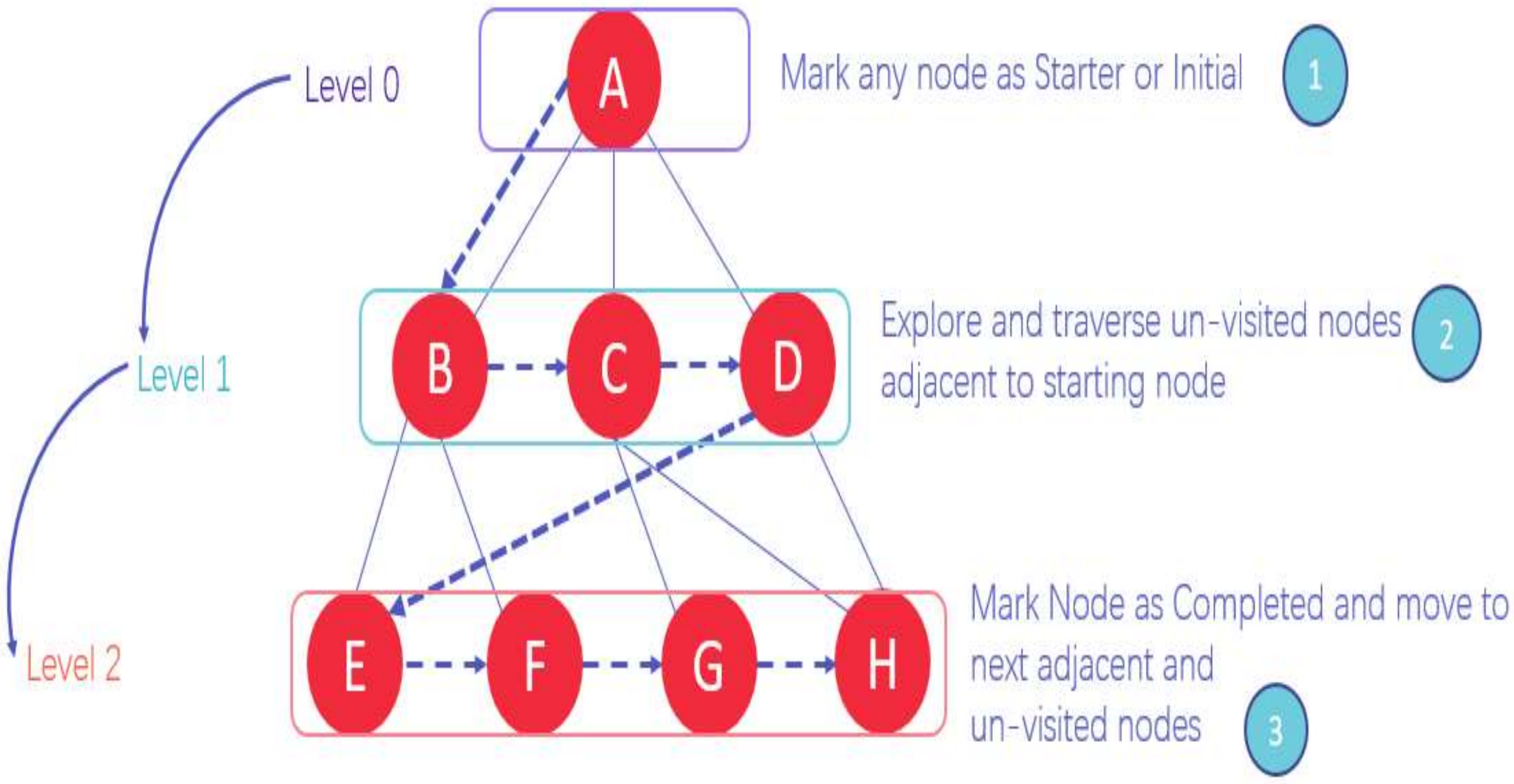
BFS Algorithms



visit 10,
visit 8, 12
visit 11, 13
∴ The BFS sequence will be
10, 8, 12, 11, 13

BFS Algorithms

CONCEPT DIAGRAM



Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3

BFS Algorithms

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

BFS Algorithms

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

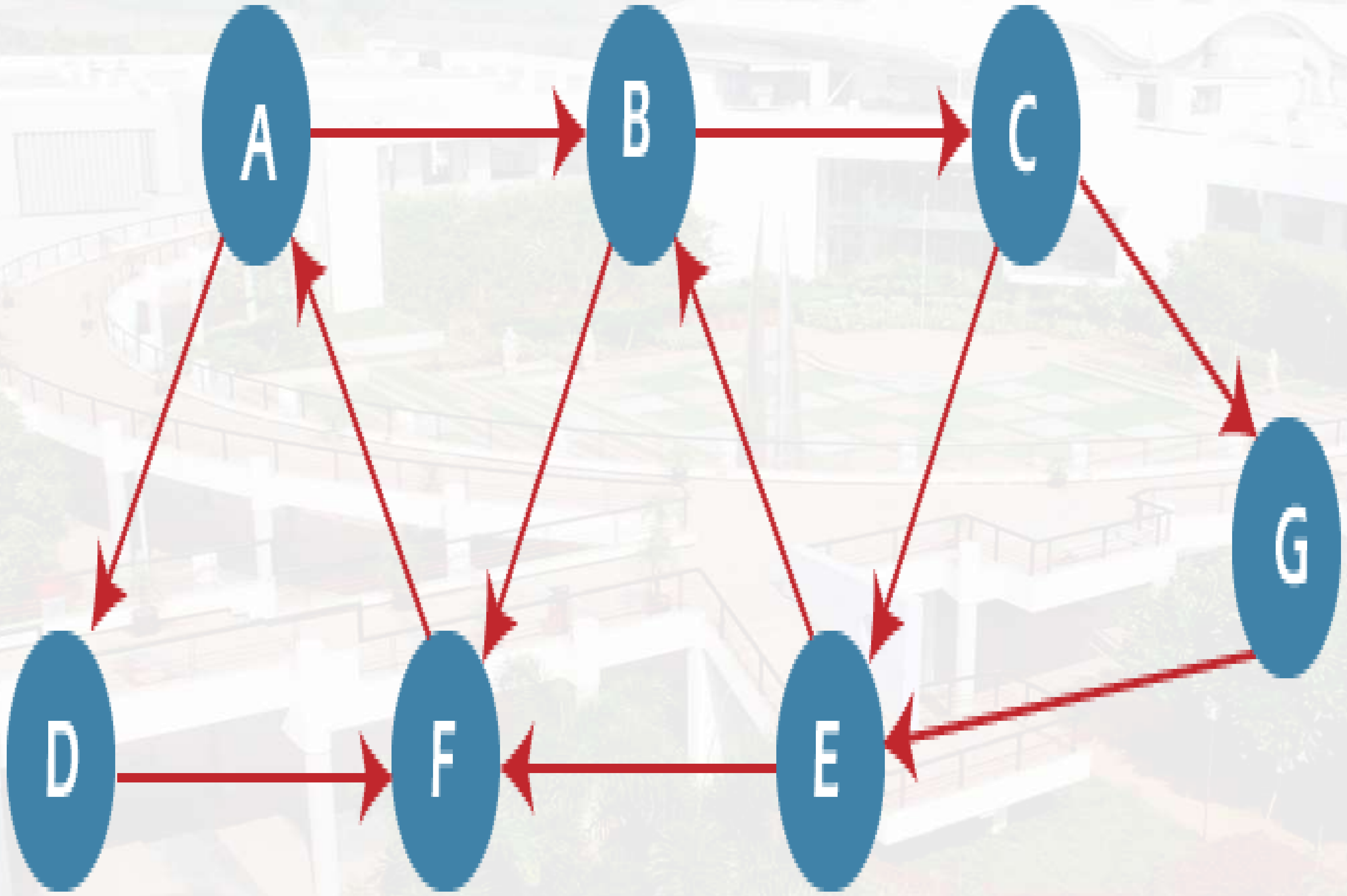
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Example of BFS Algorithms

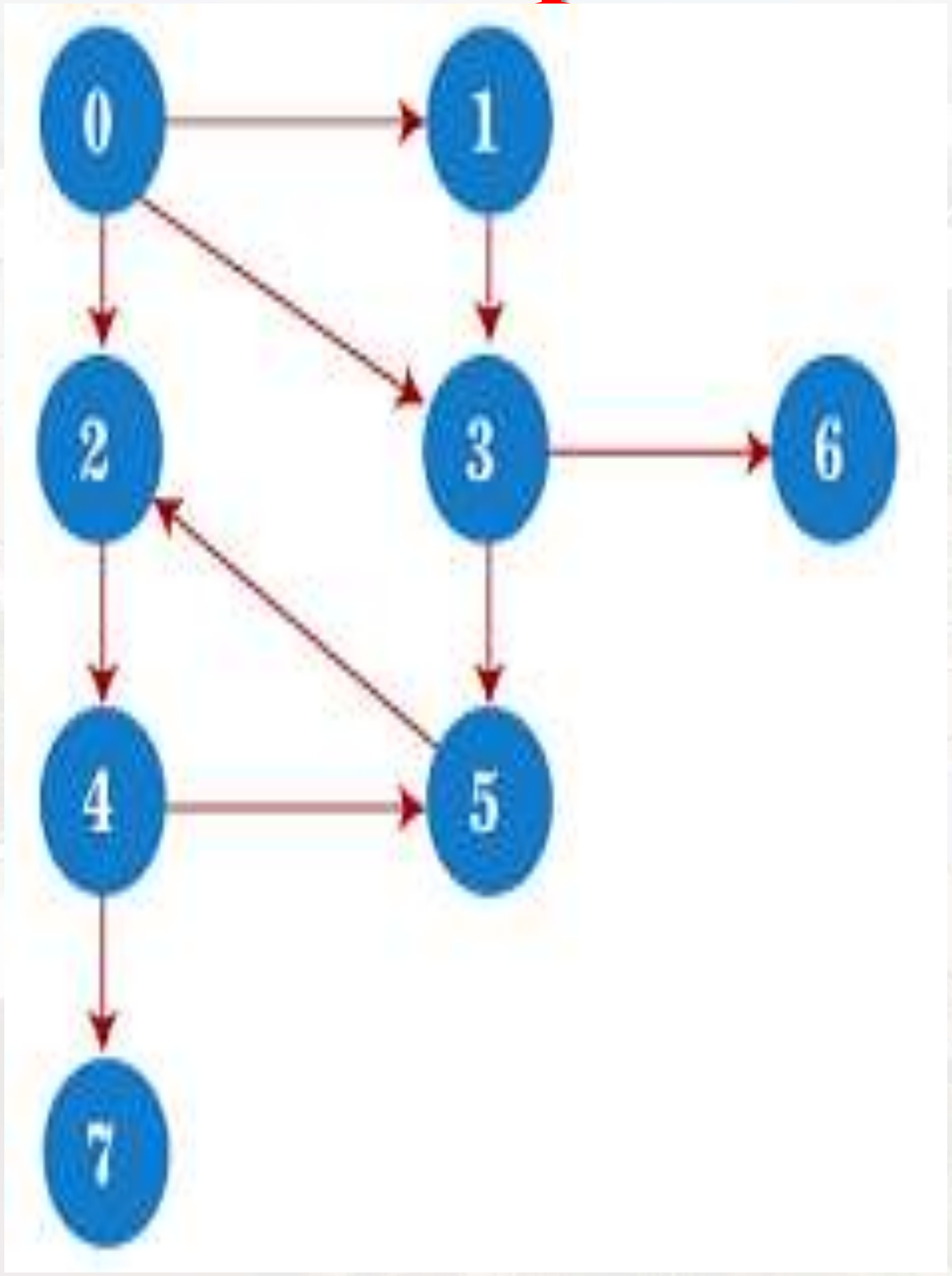


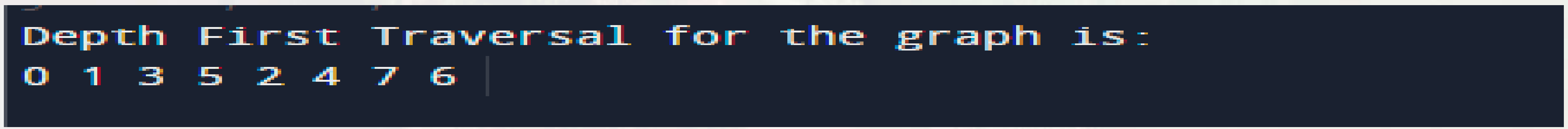
Adjacency Lists

A	B, D
B	C, F
C	E, G
G	E
E	B, F
F	A
D	F

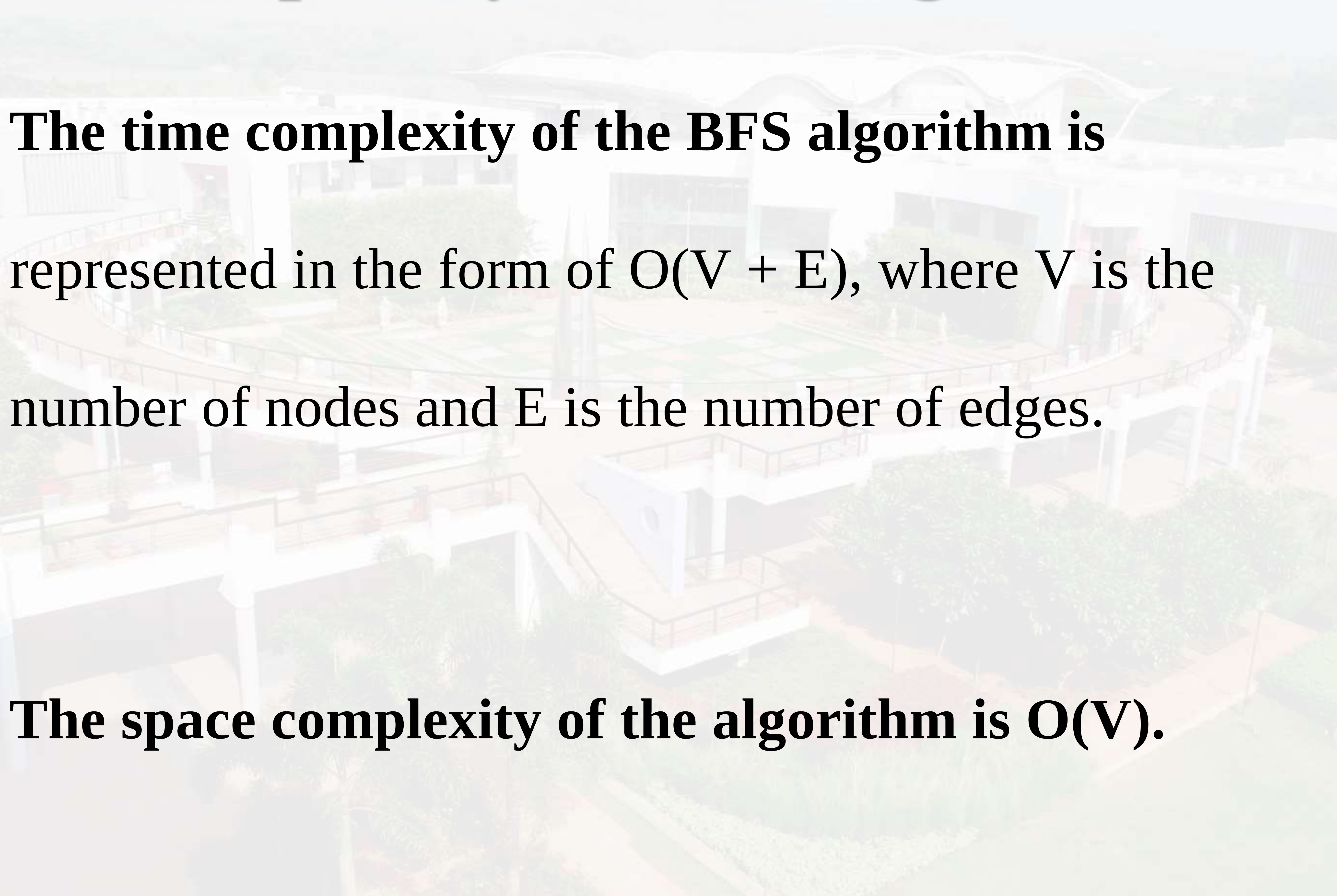
A → B → D → C → F → E → G

Example of BFS Algorithms





```
Depth First Traversal for the graph is:
0 1 3 5 2 4 7 6
```



Complexity of BFS Algorithms

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Application of BFS Algorithms

1. BFS can be used to find the neighboring locations from a given source location.

2. In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.

3. BFS is used to determine the shortest path and minimum spanning tree.

4. BFS is also used in Cheney's technique to duplicate the garbage collection.

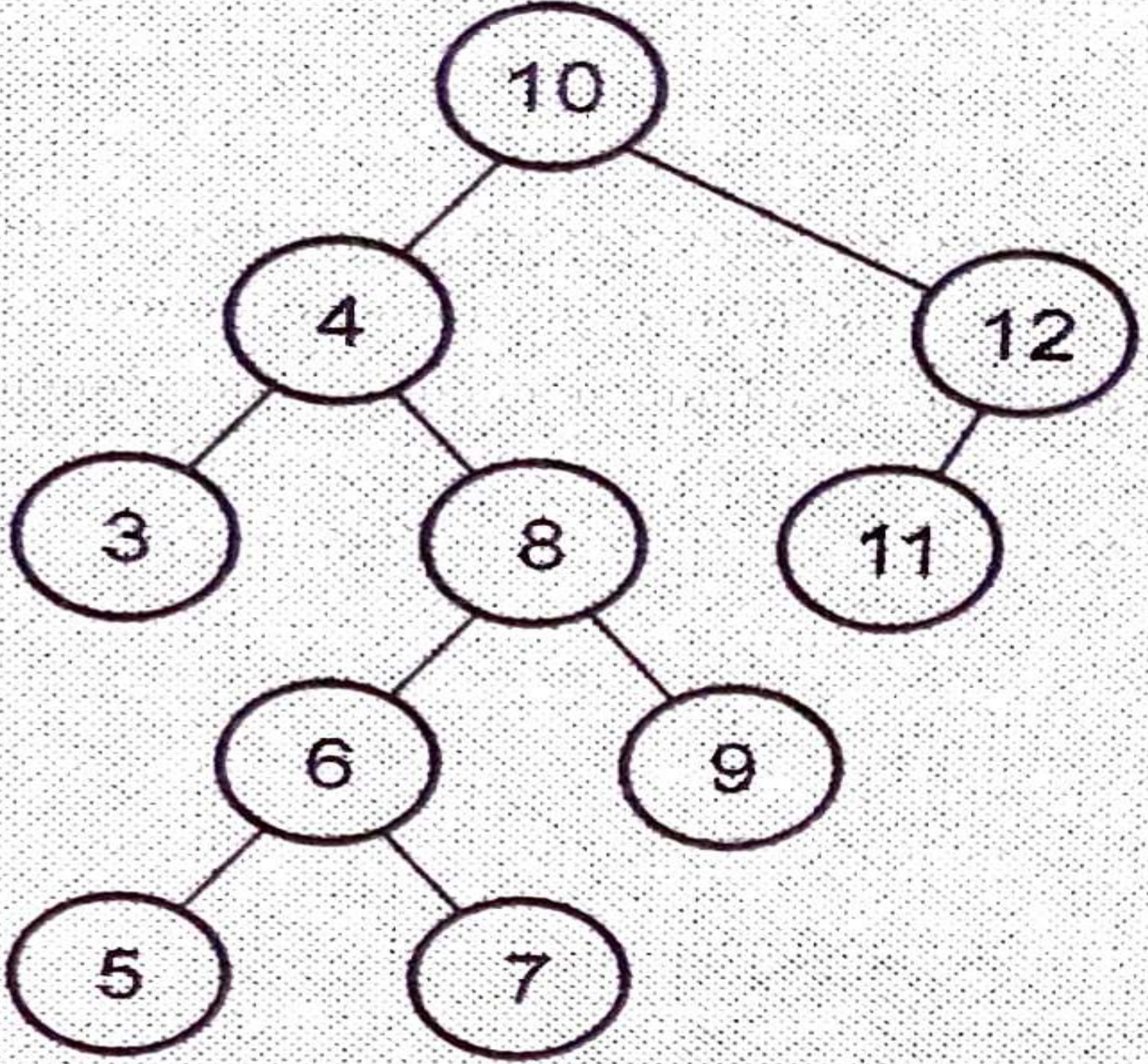
5. It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Application of BFS Algorithms

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. Cycle detection in an undirected graph
5. In minimum spanning tree

BFS EXAMPLE

Consider following tree and find its BFS sequence.



e 7, display it.

BFS sequence : 10, 4, 12, 3, 8, 11, 6, 9, 5, 7

BFS Vs DFS

sn	BFS	DFS
1	BFS finds the shortest path to the destination.	DFS goes to the bottom of a subtree, then backtracks.
2	The full form of BFS is Breadth-First Search.	The full form of DFS is Depth First Search.
3	It uses a **queue** to keep track of the next location to visit.	It uses a **stack** to keep track of the next location to visit.
4	BFS traverses according to **tree level.**	DFS traverses according to **tree depth.**
5	It is implemented using **FIFO list.**	It is implemented using **LIFO list.**
6	It requires **more memory** as compare to DFS.	It requires **less memory** as compare to BFS.

BFS Vs DFS

sn	BFS	DFS
7	There is no need of backtracking in BFS.	There is a need of backtracking in DFS.
8	You can never be trapped into finite loops.	You can be trapped into infinite loops.
9	If you do not find any goal, you may need to expand many nodes before the solution is found.	If you do not find any goal, the leaf node backtracking may occur.

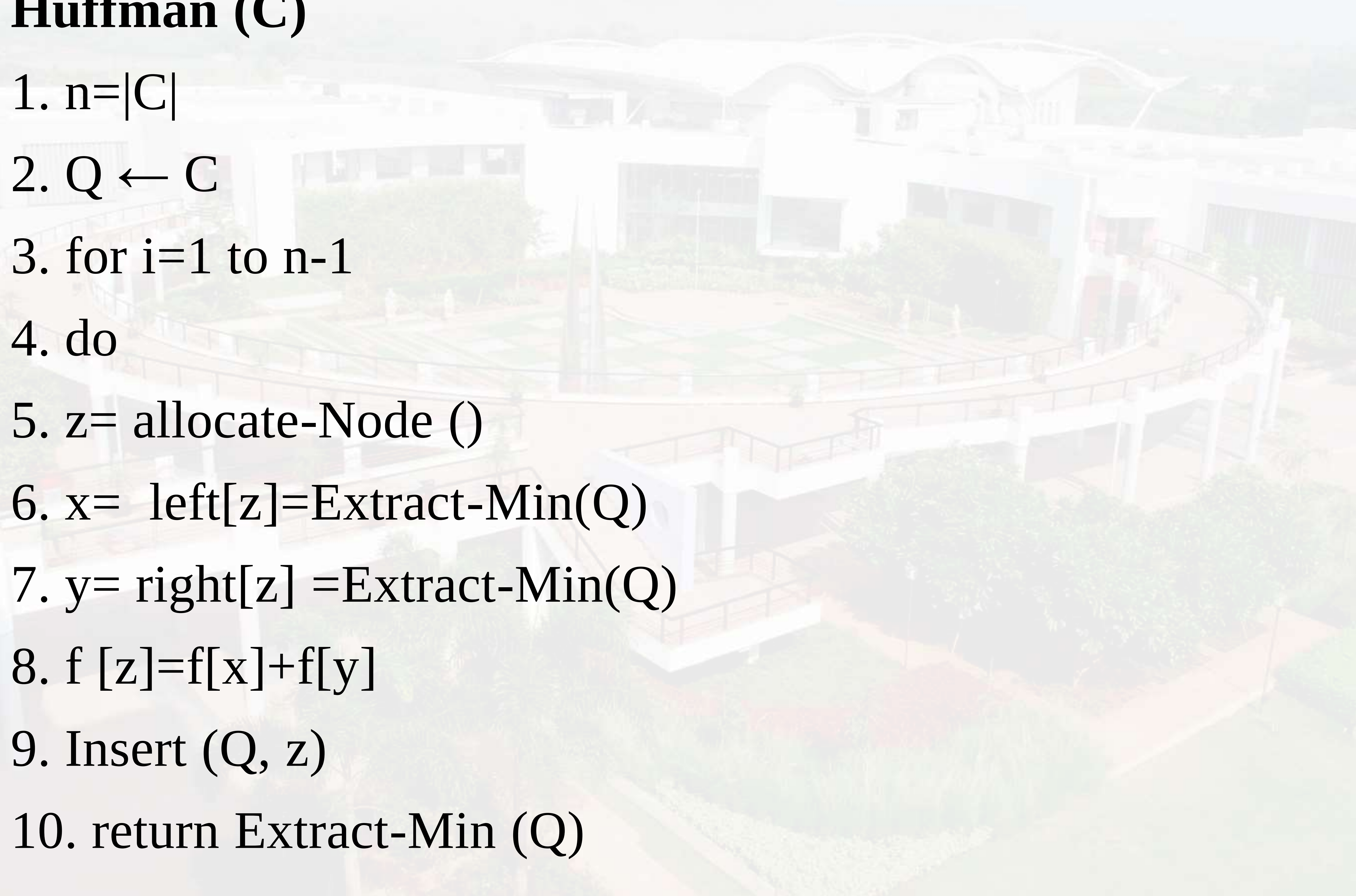
Huffman Coding Tree

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
- Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Huffman Coding Tree

- (i) Data can be encoded efficiently using Huffman Codes.
- (ii) It is a widely used and beneficial technique for compressing data.
- (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

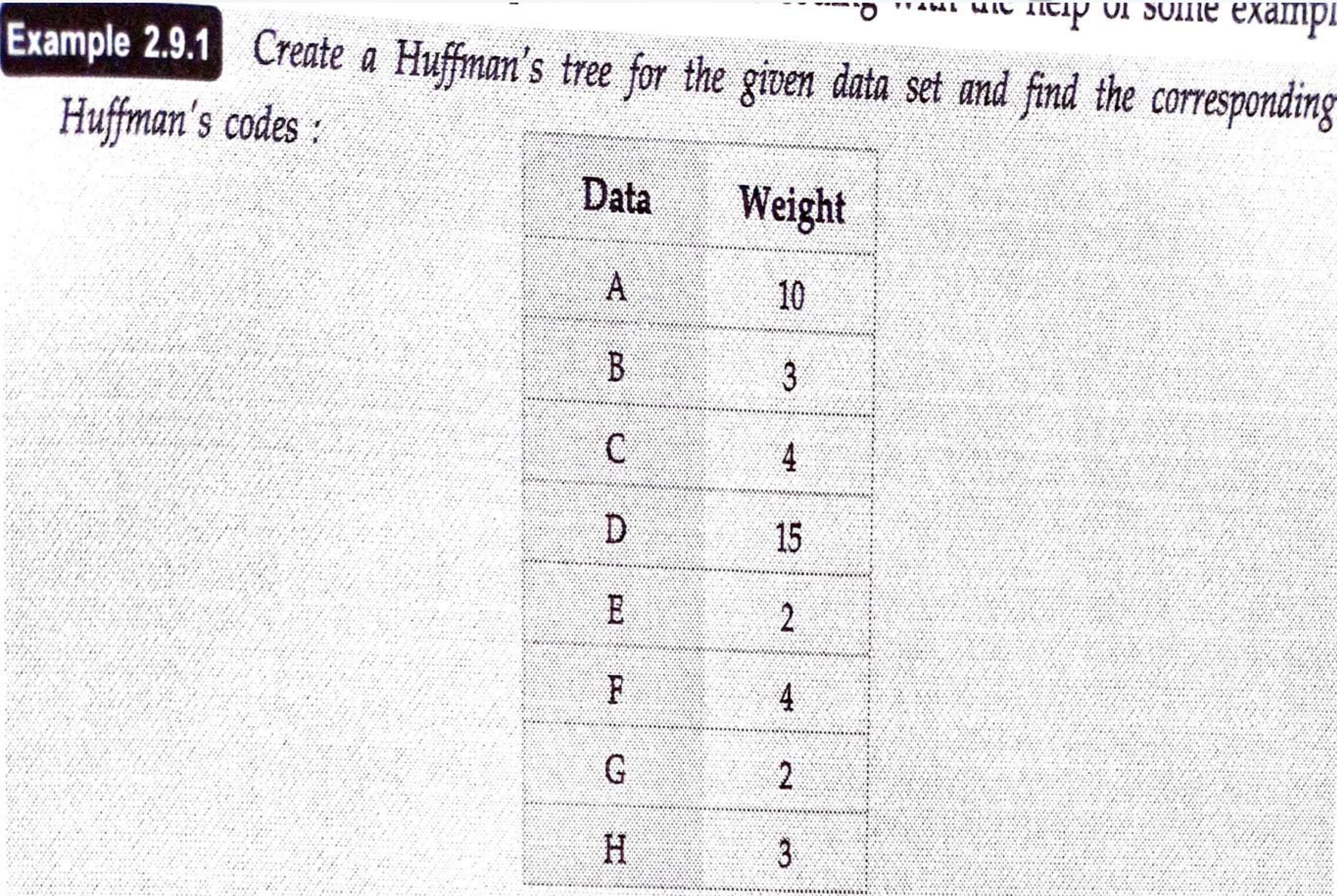
Algorithms of Huffman Coding



Huffman (C)

```
1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x=  left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)
```

Huffman's Tree Example



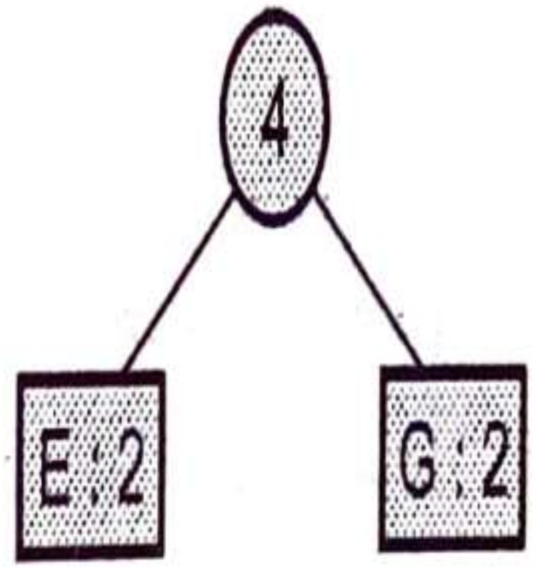
Example 2.9.1 *Create a Huffman's tree for the given data set and find the corresponding Huffman's codes :*

Data	Weight
A	10
B	3
C	4
D	15
E	2
F	4
G	2
H	3

Huffman's Tree Example

Step 1 : We will combine first two least values form the table. Creating a parent node.

E	2
G	2
B	3
H	3
C	4
F	4
A	10
D	15

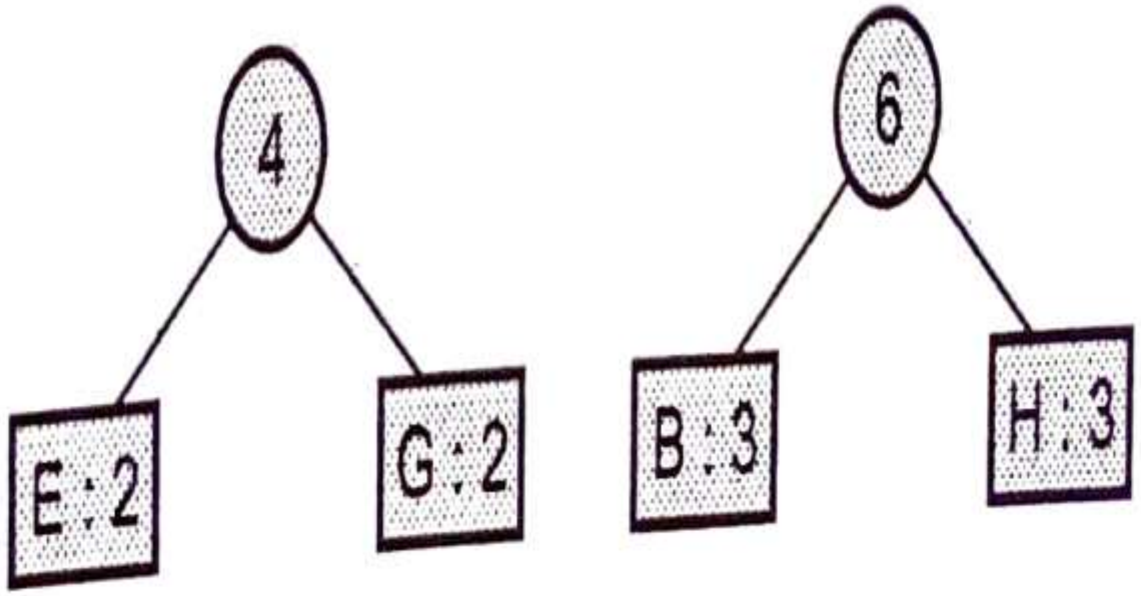


Huffman's Tree Example

Step 2 : Remove first two entries, from the table in step 1 and add the entry of parent node created in step 1 in the table by maintaining sorted order.

Combine first two least values to form a parent node.

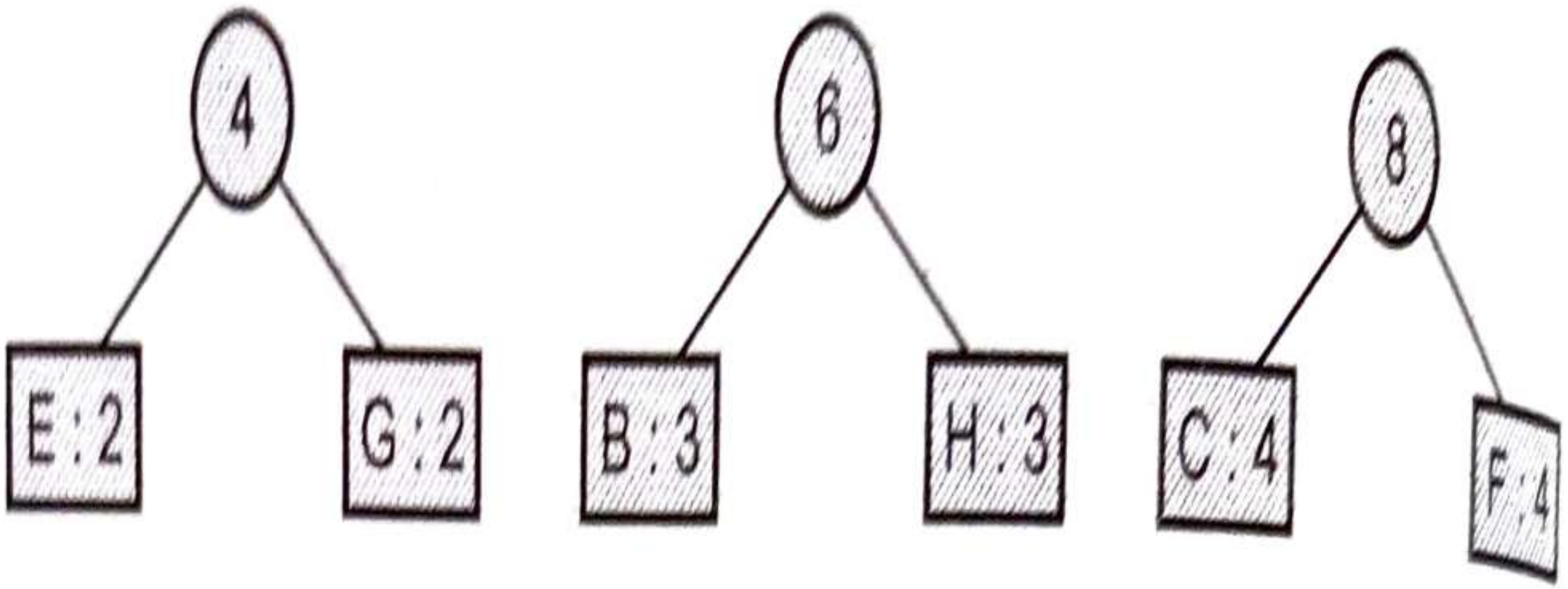
B	3
H	3
C	4
F	4
EG	4
A	10
D	15



Huffman's Tree Example

Step 3 : Remove first entries from the table and add entry of parent node at appro
position in the table.

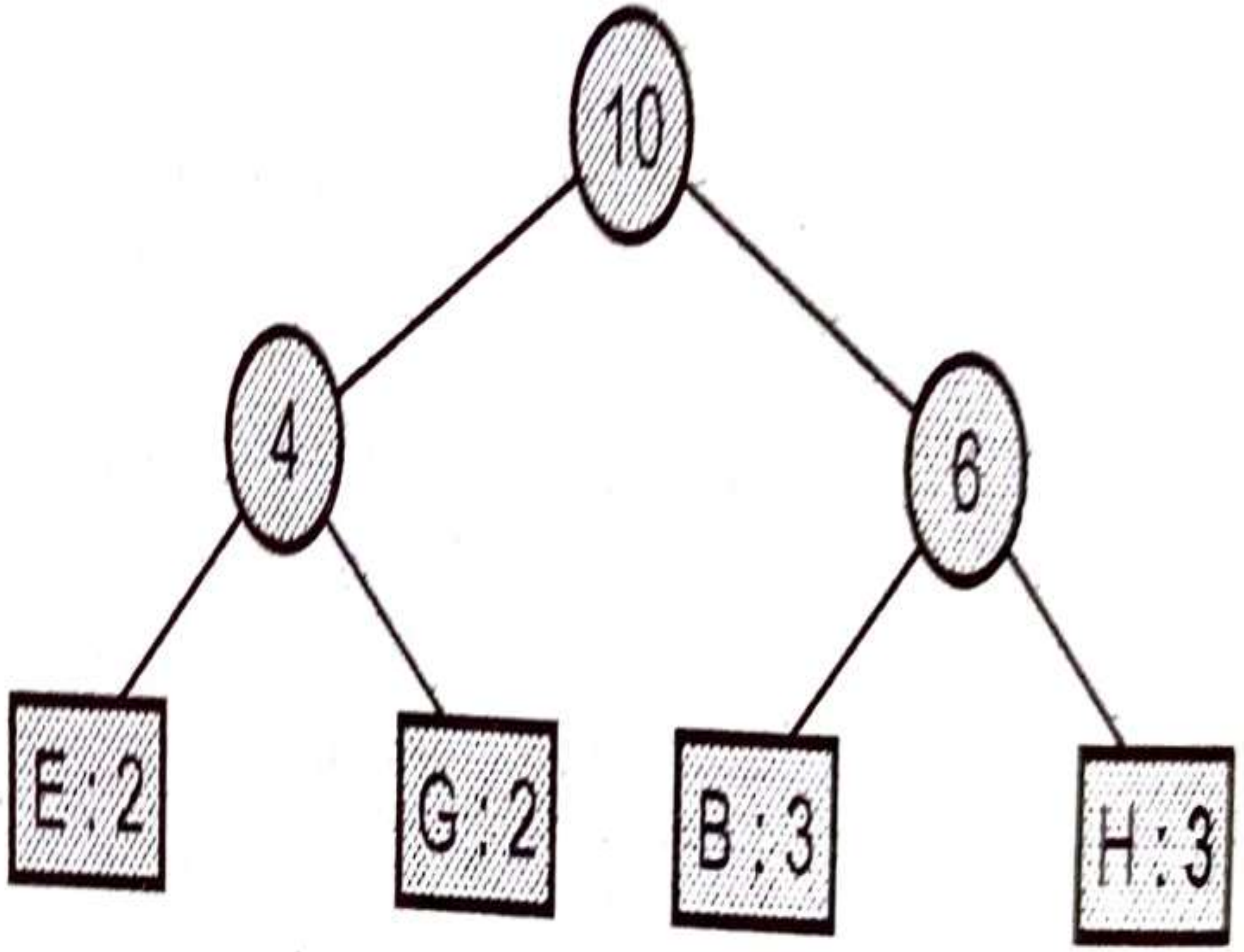
C	4
F	4
EG	4
BH	6
A	10
D	15



Huffman's Tree Example

Step 4 :

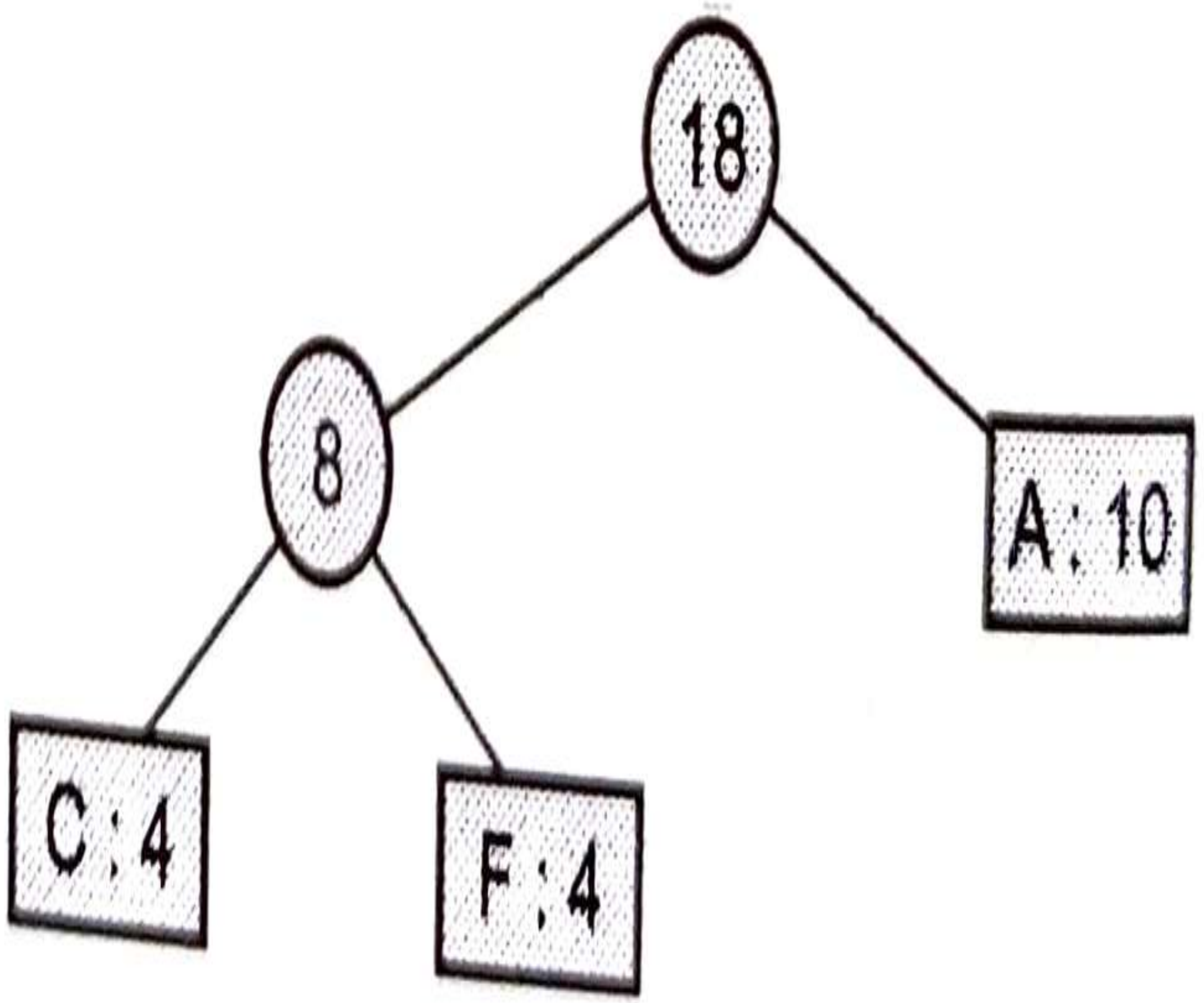
EG	4
BH	6
CF	8
A	10
D	15



Huffman's Tree Example

Step 5 :

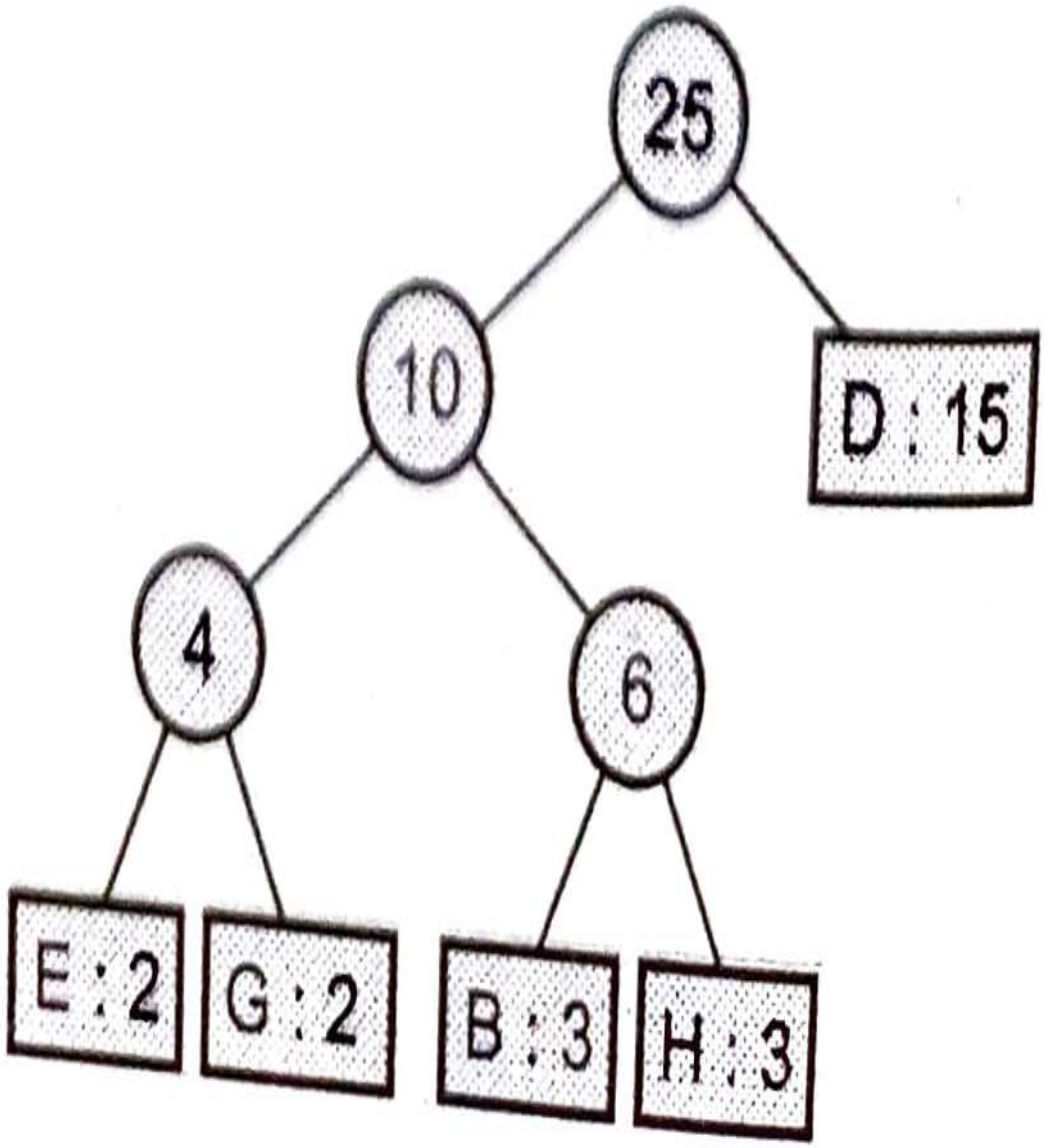
CF	8
A	10
EGBH	10
D	15



Huffman’s Tree Example

Step 6 :

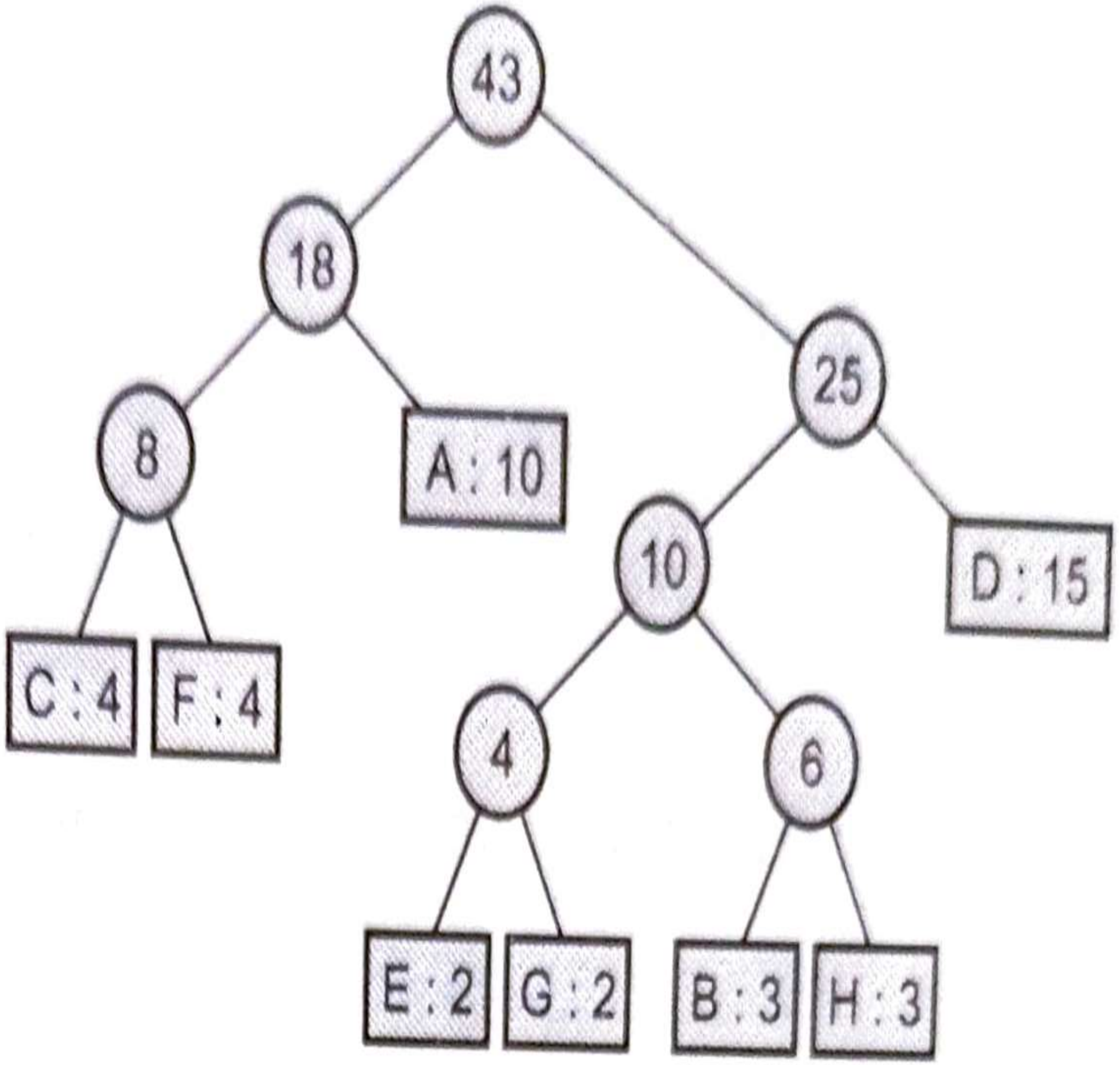
EGBH	10
D	15
CFA	18



Huffman's Tree Example

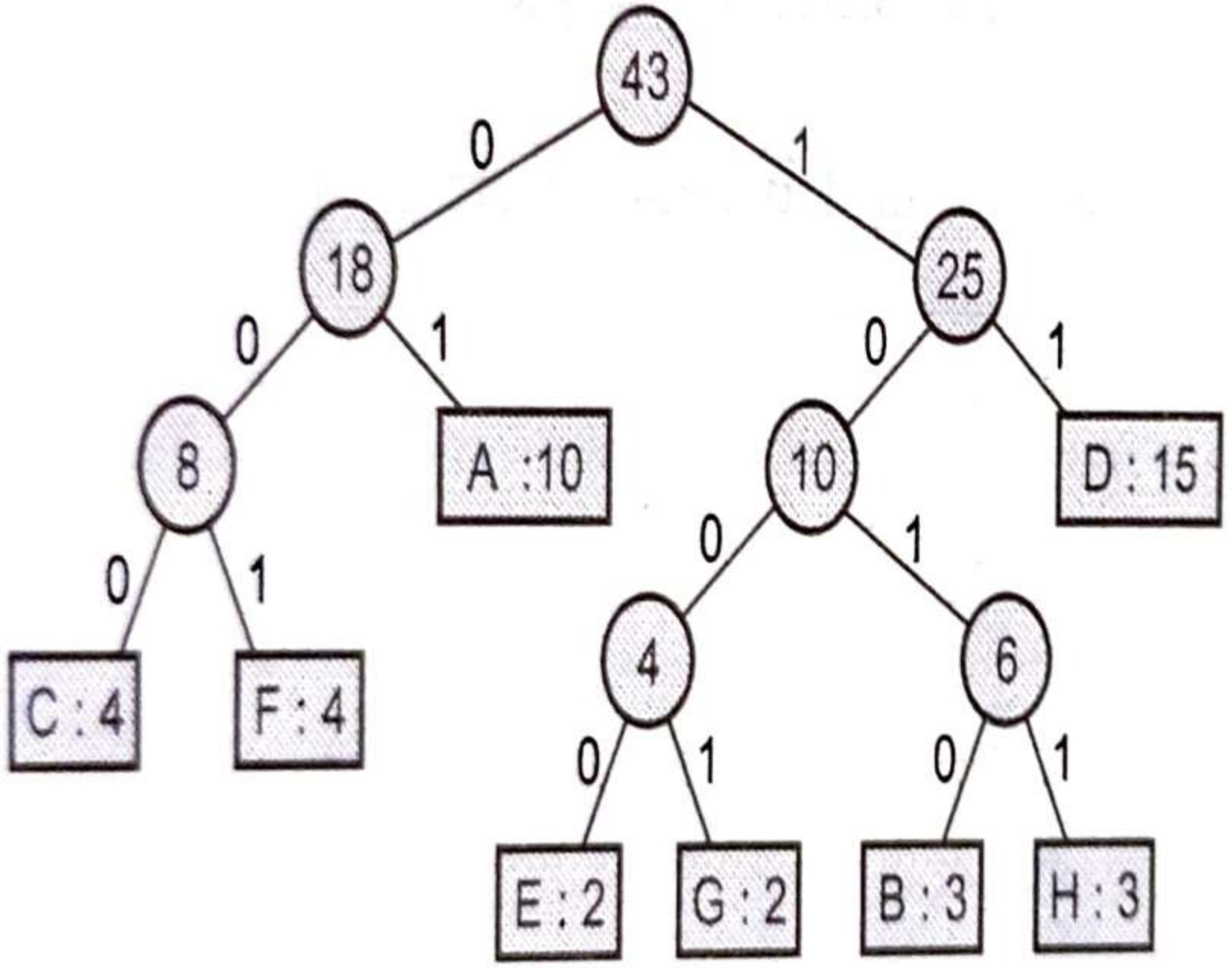
Step 7 : Huffman tree is thus obtained.

CFA	18
EGBHD	25



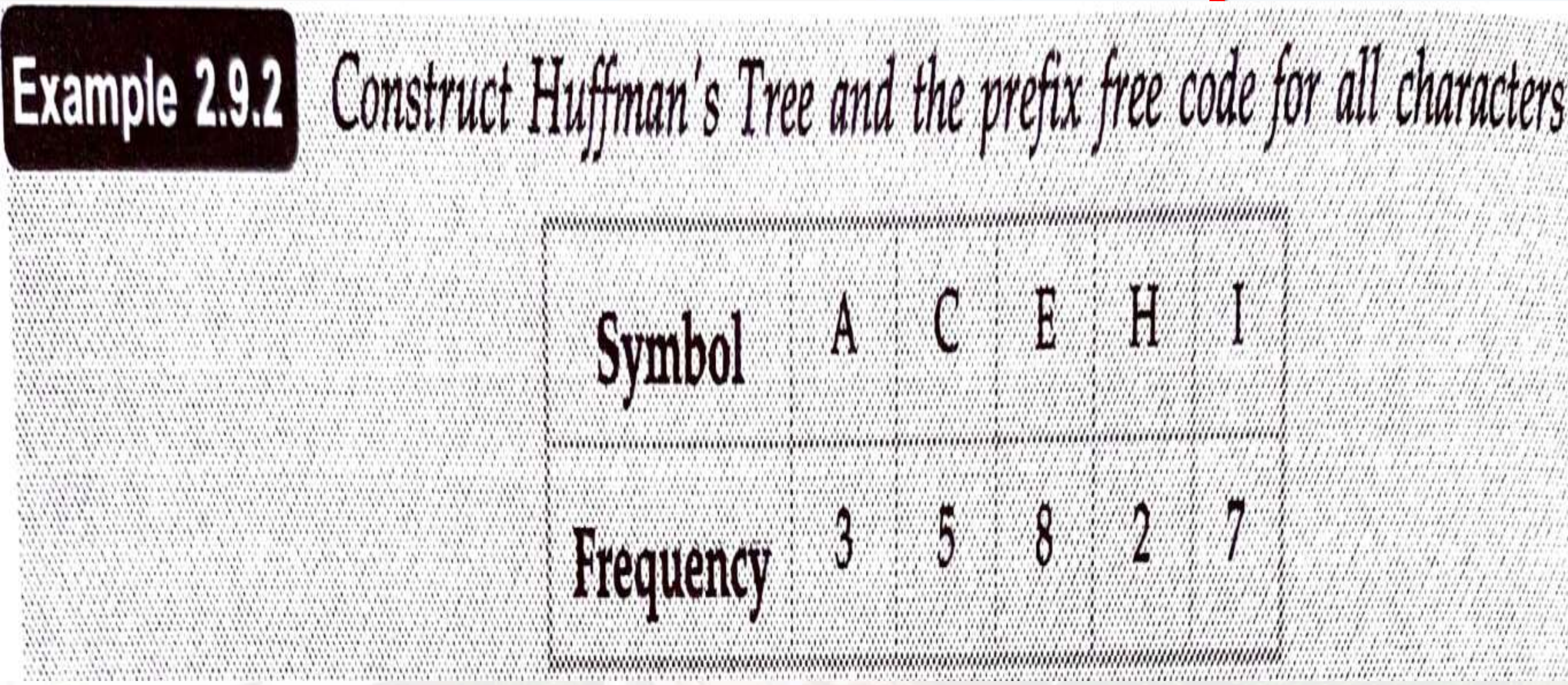
Huffman's Tree Example

Step 8 : The left branch is coded with 0 and right branch with 1.



Data	Weight	Code
A	10	01
B	3	1010
C	4	000
D	15	11
E	2	1000
F	4	001
G	2	1001
H	3	1011

Huffman's Tree Example

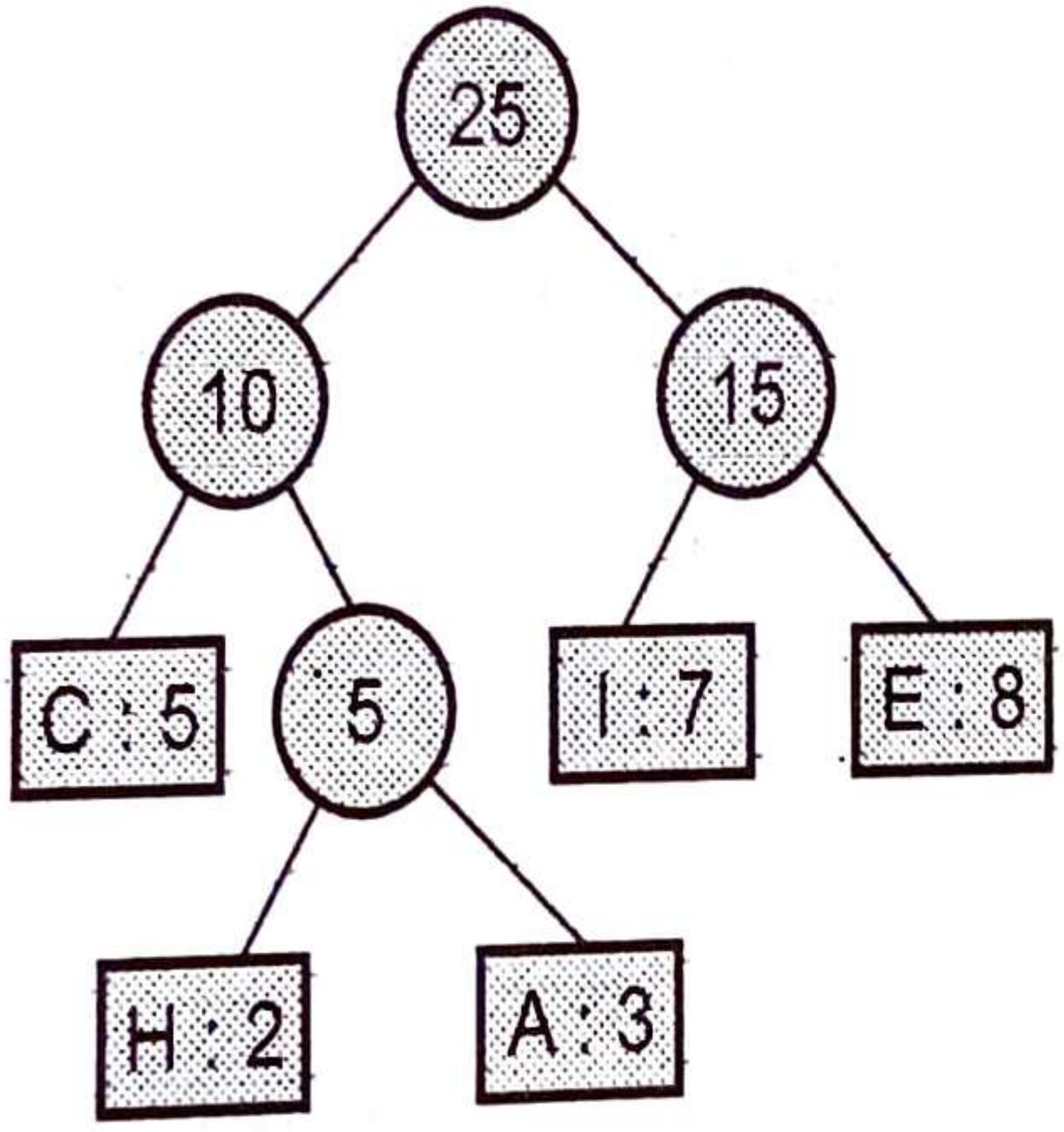


Example 2.9.2 *Construct Huffman's Tree and the prefix free code for all characters*

Symbol	A	C	E	H	I
Frequency	3	5	8	2	7

Huffman's Tree Example

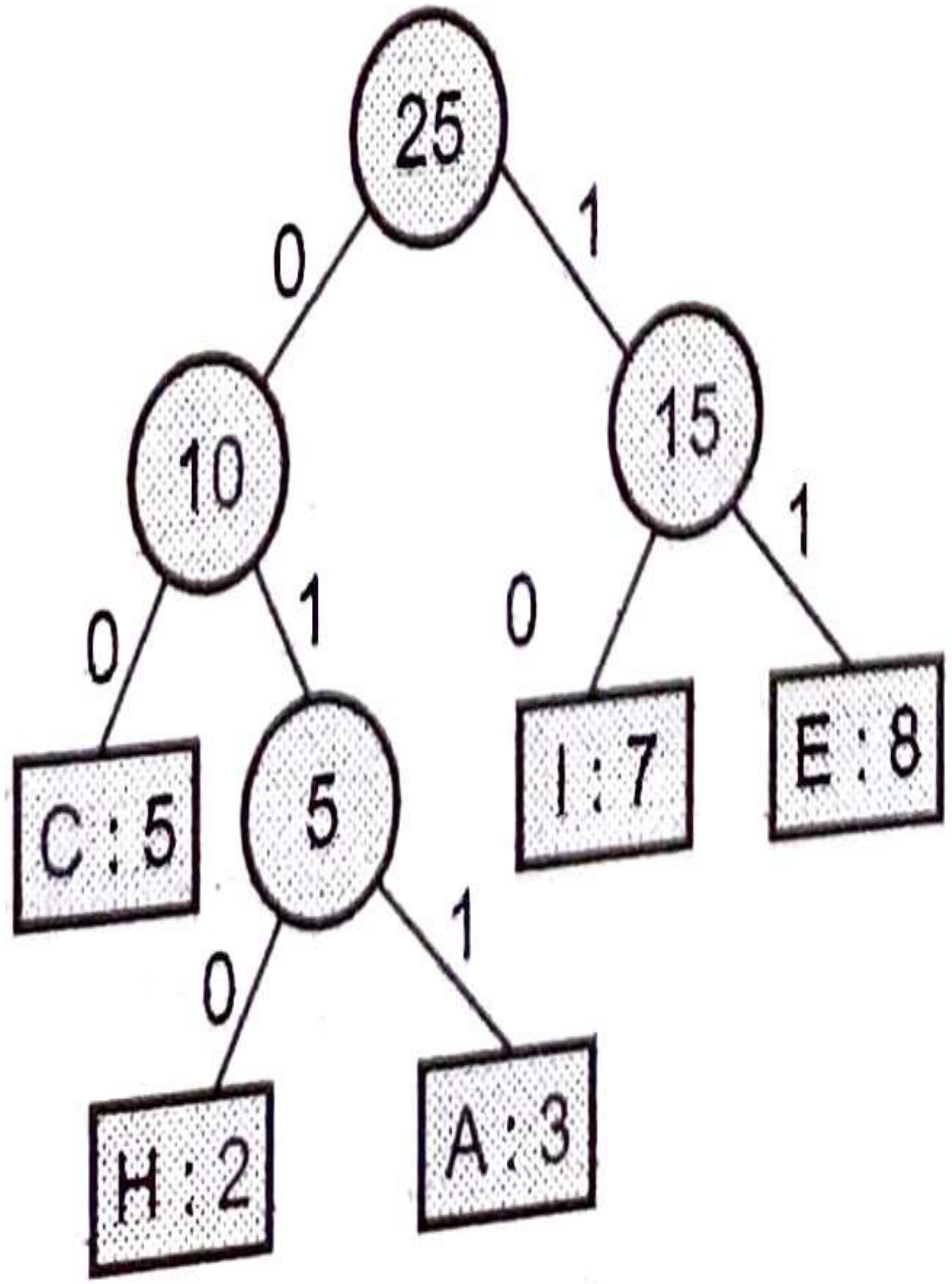
Combine two entries of the table to form parent node.



This is Huffman's tree

Huffman’s Tree Example

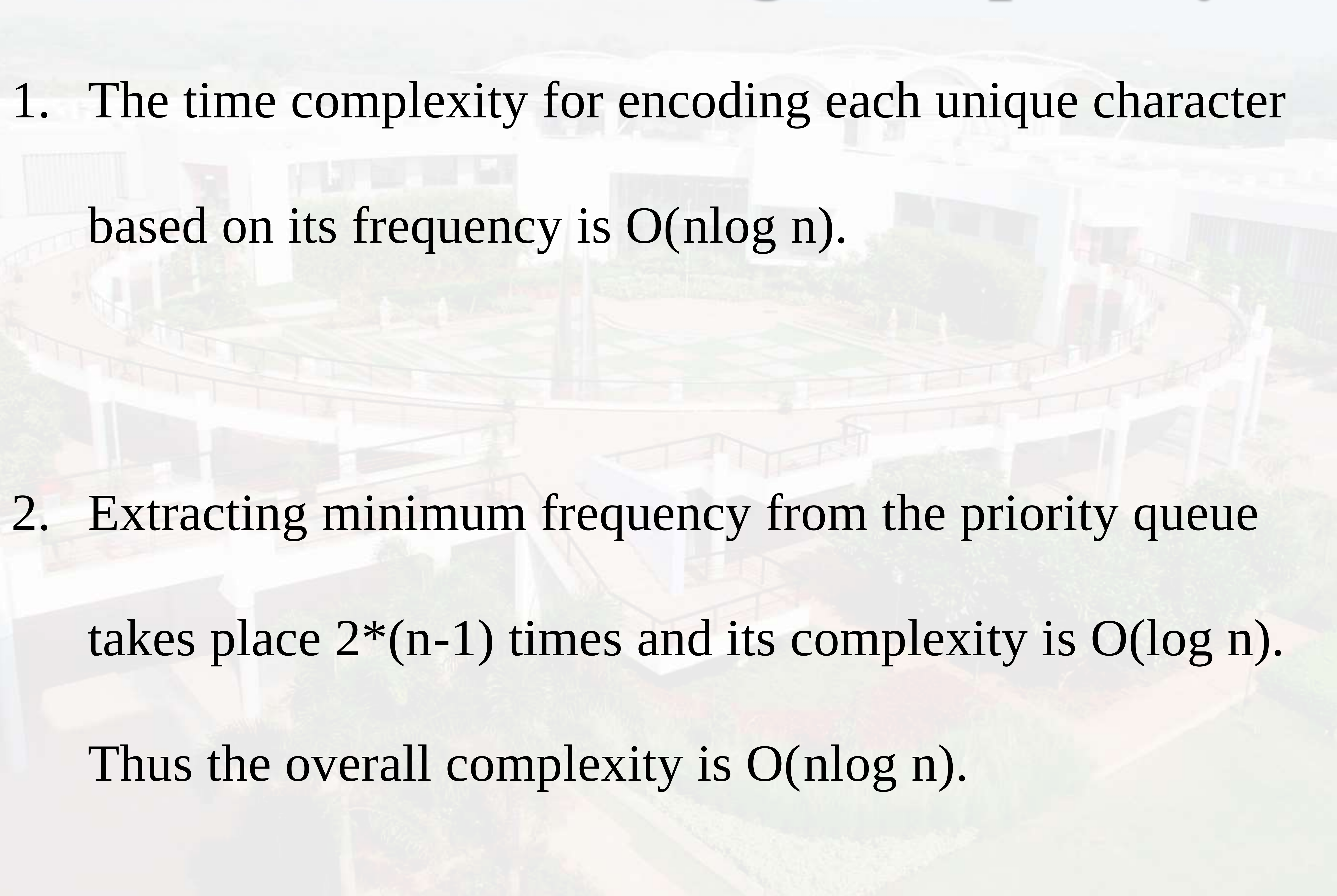
For encoding, the left branch is encoded as 0 and right branch as 1.



Huffman’s Coding Complexity

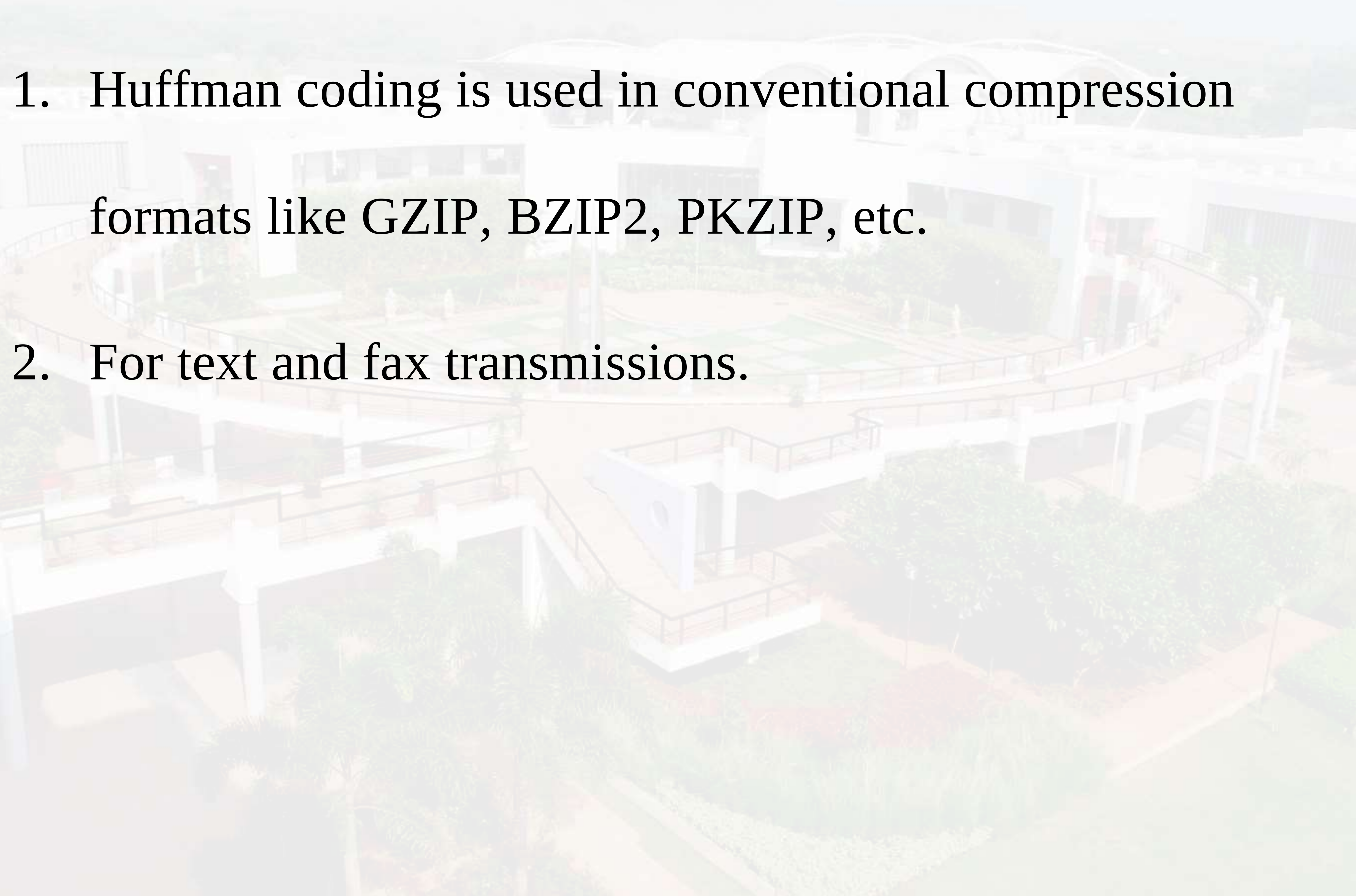
1. The time complexity for encoding each unique character based on its frequency is O(nlog n).

2. Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).



Applications of Huffman's Coding

1. Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc.
2. For text and fax transmissions.



Binary tree vs Binary search tree

Points	Binary Tree	Binary Search Tree
Definition	A Binary Tree is a non-linear data structure in which a node can have 0, 1 or 2 nodes. Individually, each node consists of a left pointer, right pointer and data element.	A Binary Search Tree is an organized binary tree with a structured organization of nodes. Each subtree must also be of that particular structure.
Structure	There is no required organization structure of the nodes in the tree.	The values of left subtree of a particular node should be lesser than that node and the right subtree values should be greater.

Binary tree vs Binary search tree

points	**Binary Tree**	**Binary Search Tree**
Operations Performed	The operations that can be performed are deletion, insertion and traversal	As these are sorted binary trees, they are used for fast and efficient binary search, insertion and deletion.
Types	There are several types. Most common ones are the Complete Binary Tree, Full Binary Tree, Extended Binary Tree	The most popular ones are AVL Trees, Splay Trees, Tango Trees, T-Trees.

Threaded Binary Tree

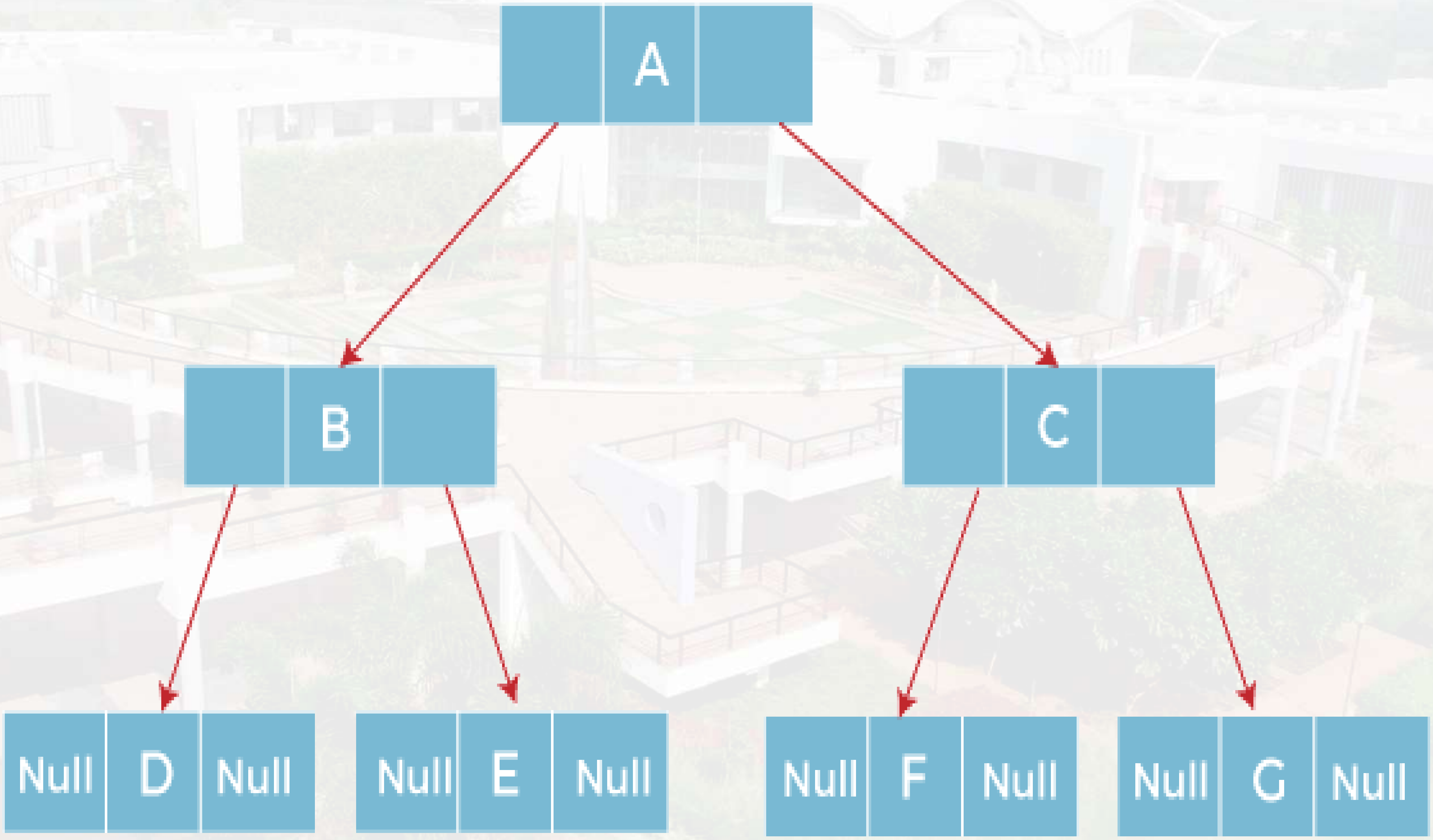
1. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion.

2. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists)

Threaded Binary Tree

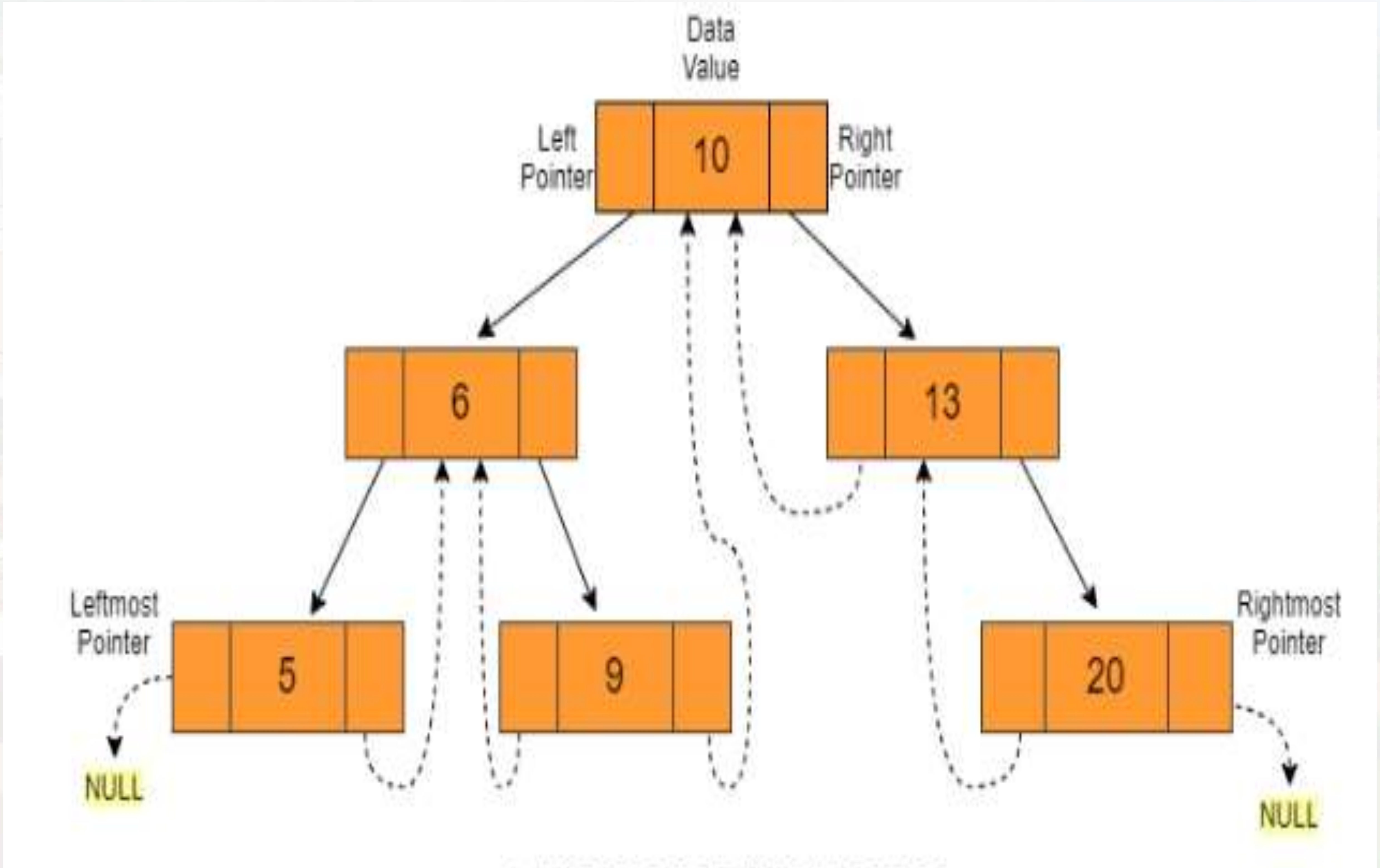
In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

Threaded Binary Tree



Threaded binary tree

Threaded Binary Tree



THREADED BINARY TREE

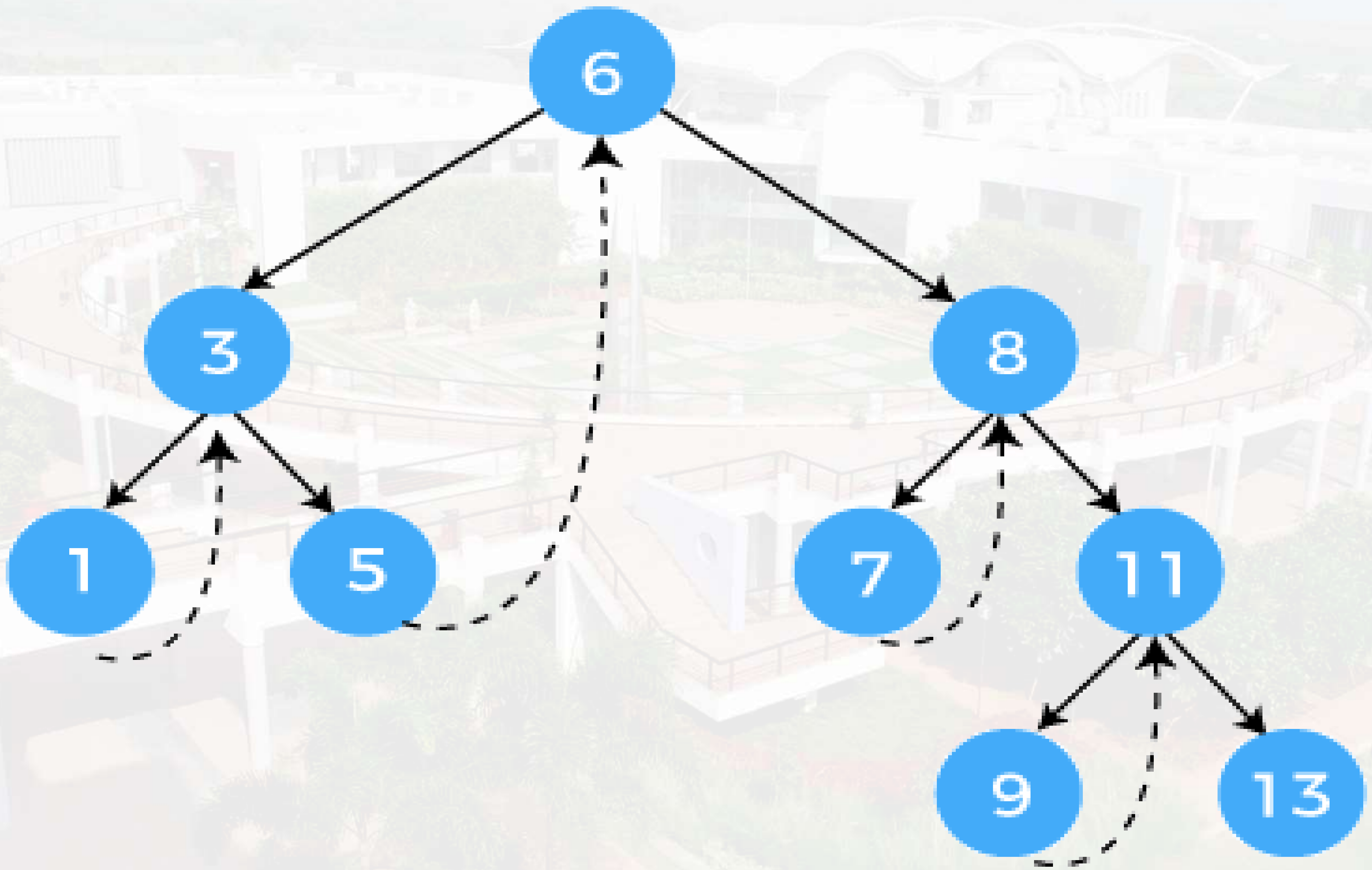
Threaded Binary Tree

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

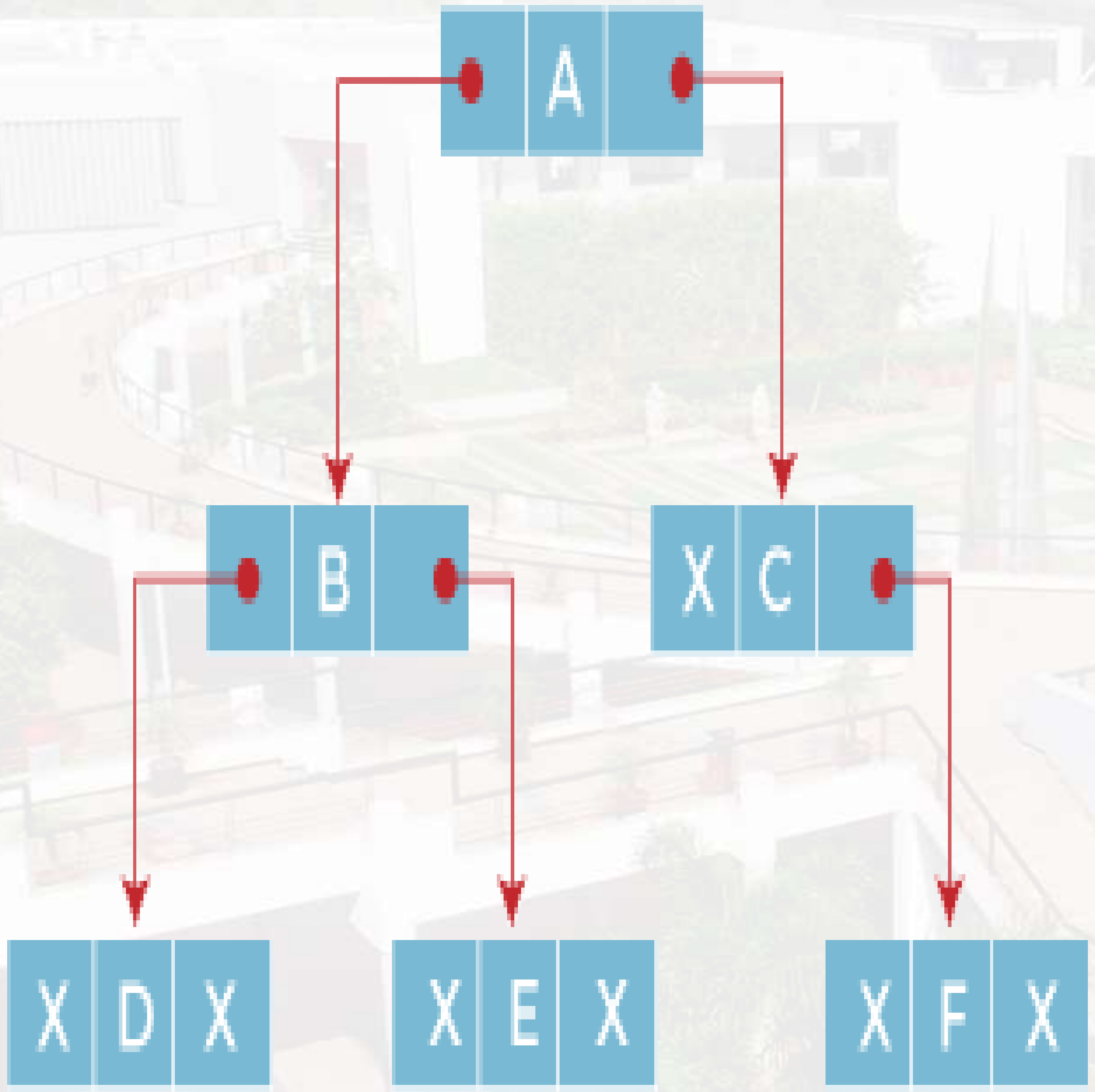
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

Threaded Binary Tree

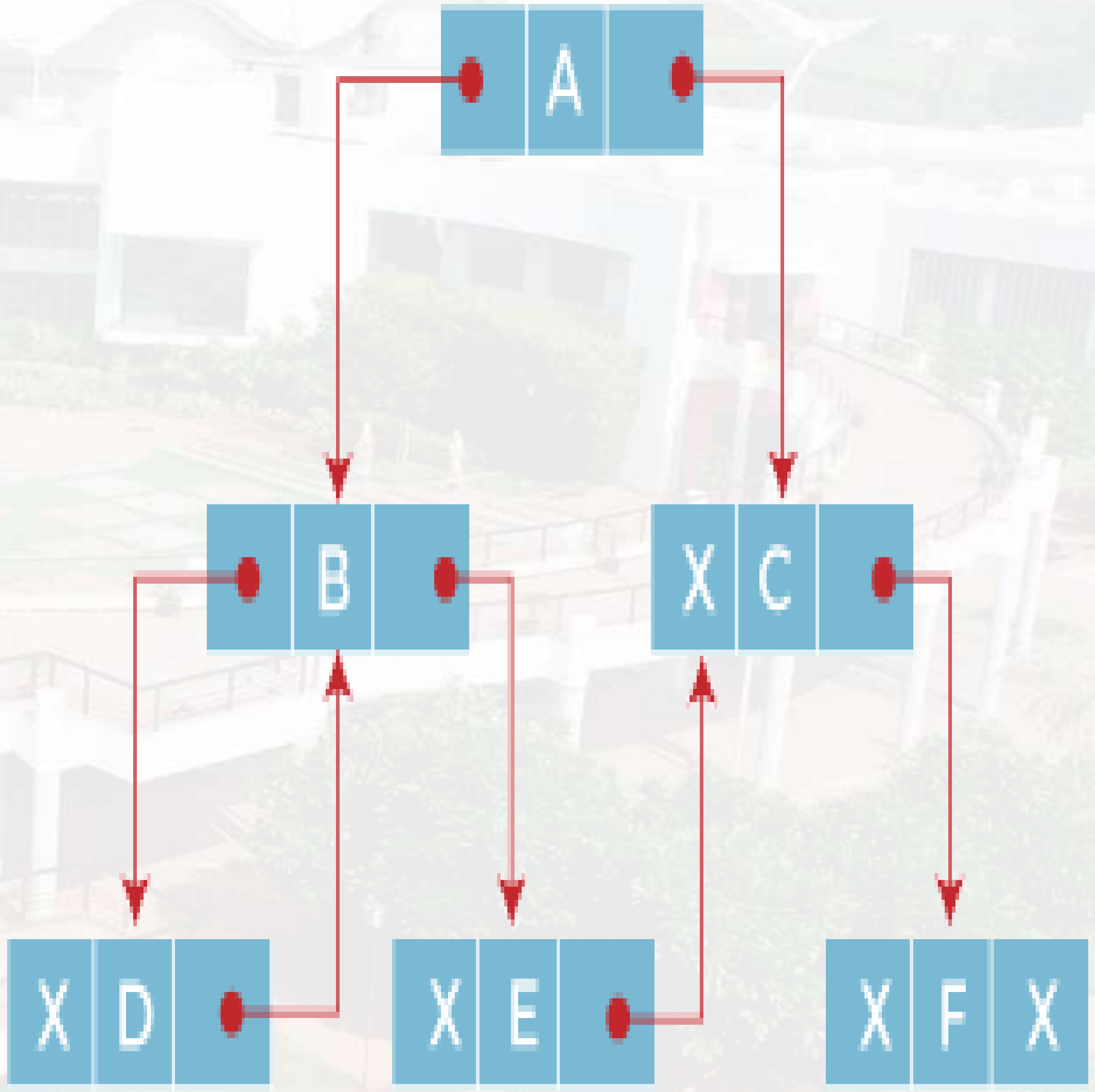


Single Threaded Binary Tree

Threaded Binary Tree

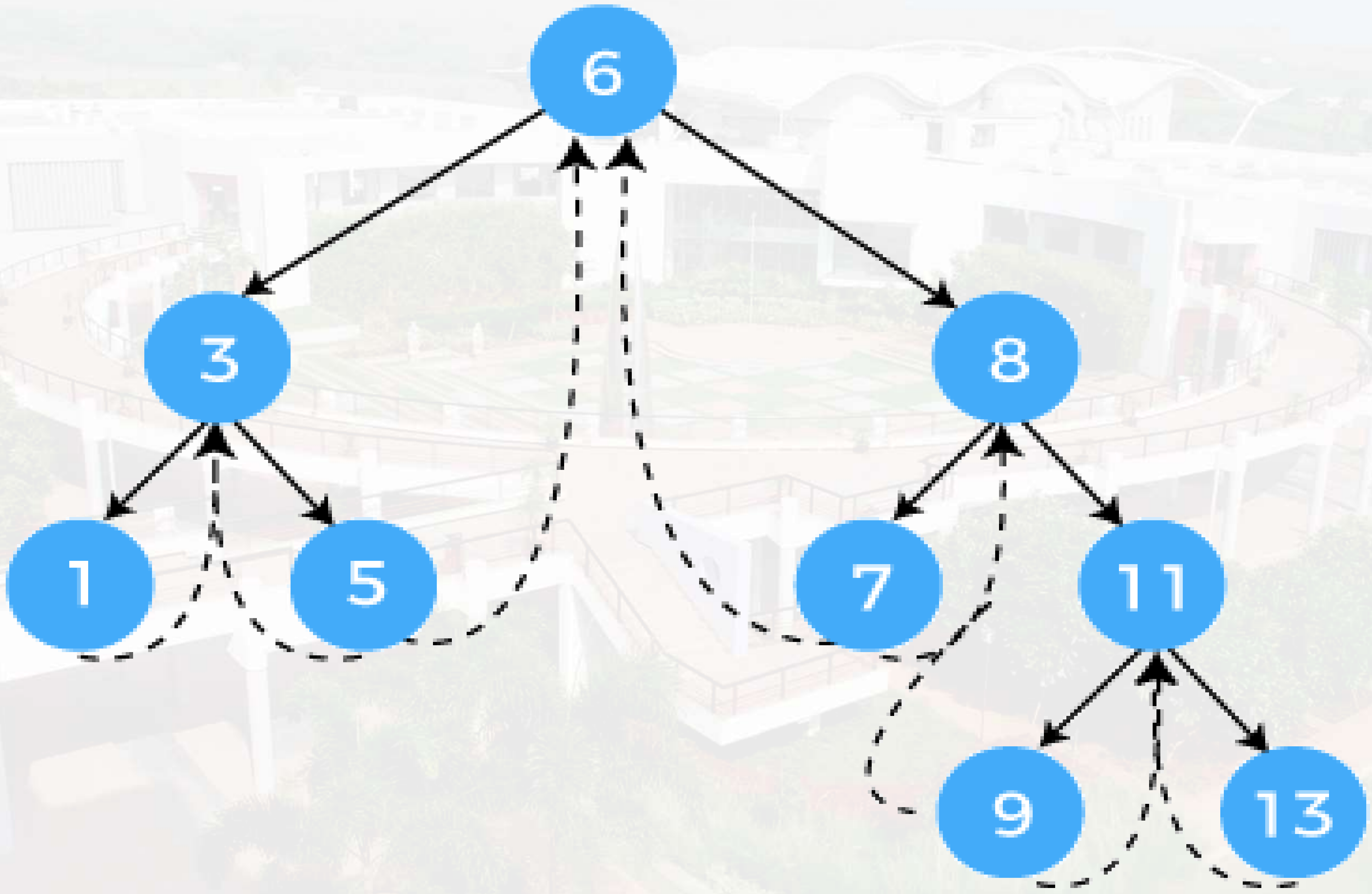


A binary tree (Inorder traversal - D, B, E, A, C, F)



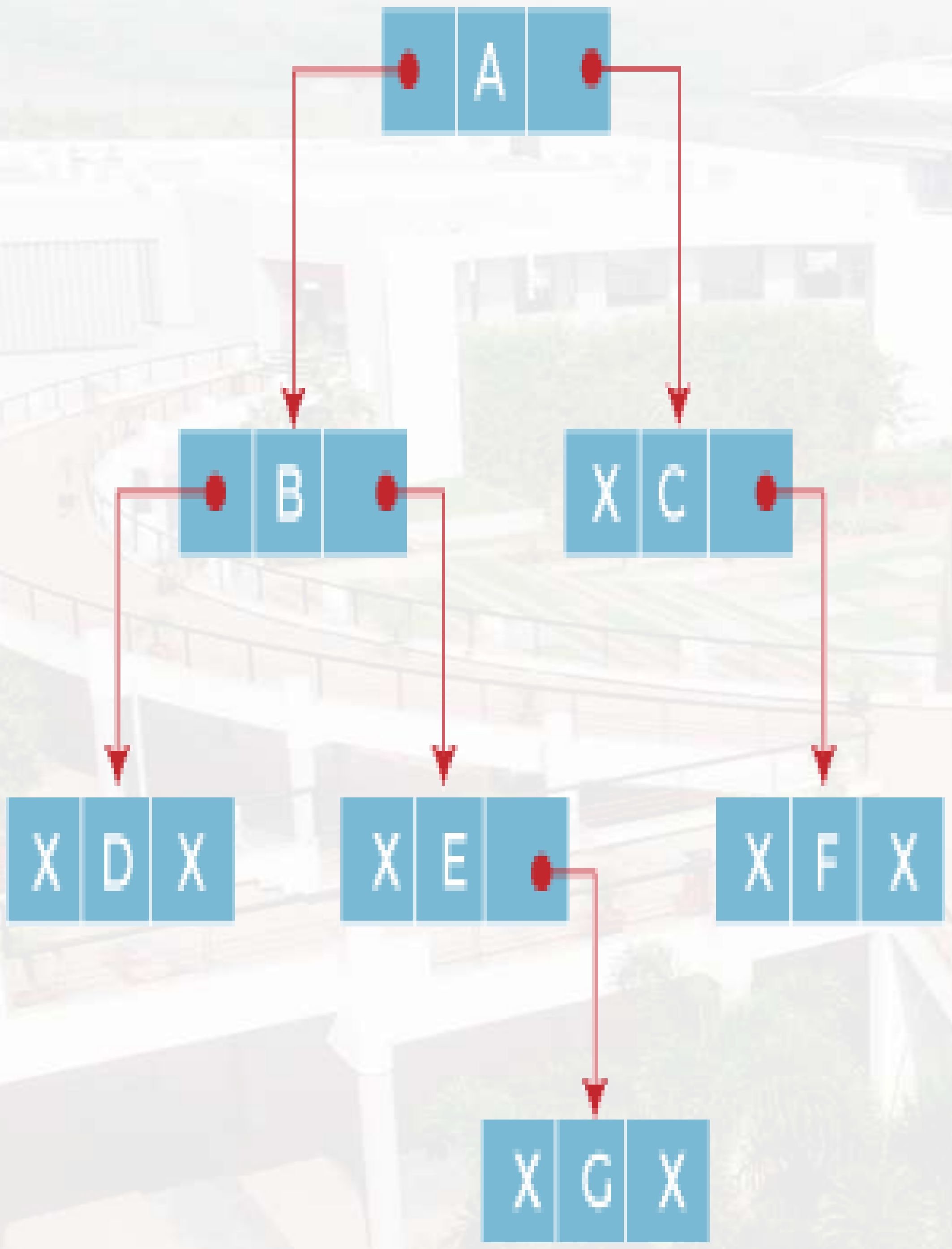
A right - threaded binary tree

Threaded Binary Tree

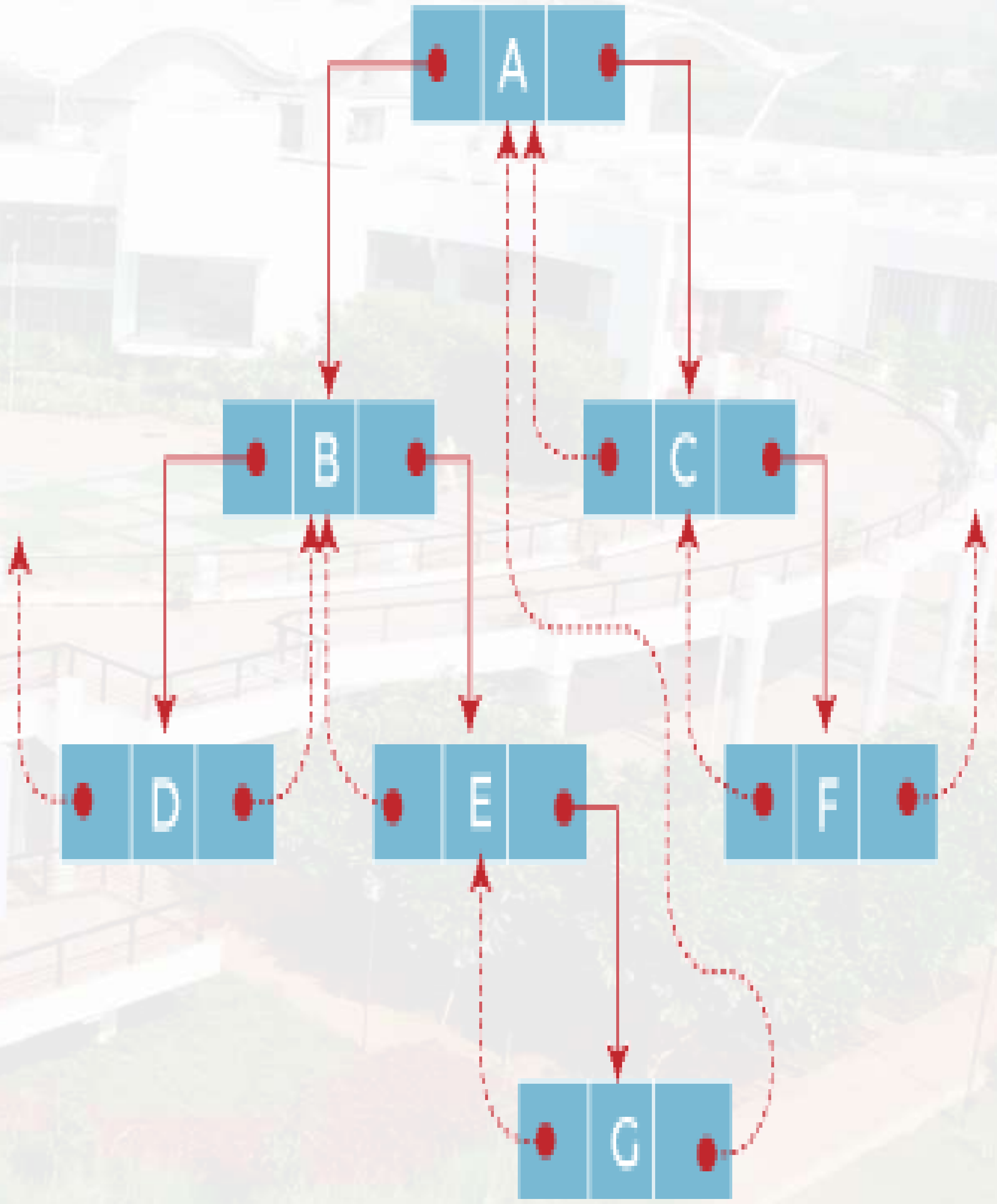


Double Threaded Binary Tree

Threaded Binary Tree

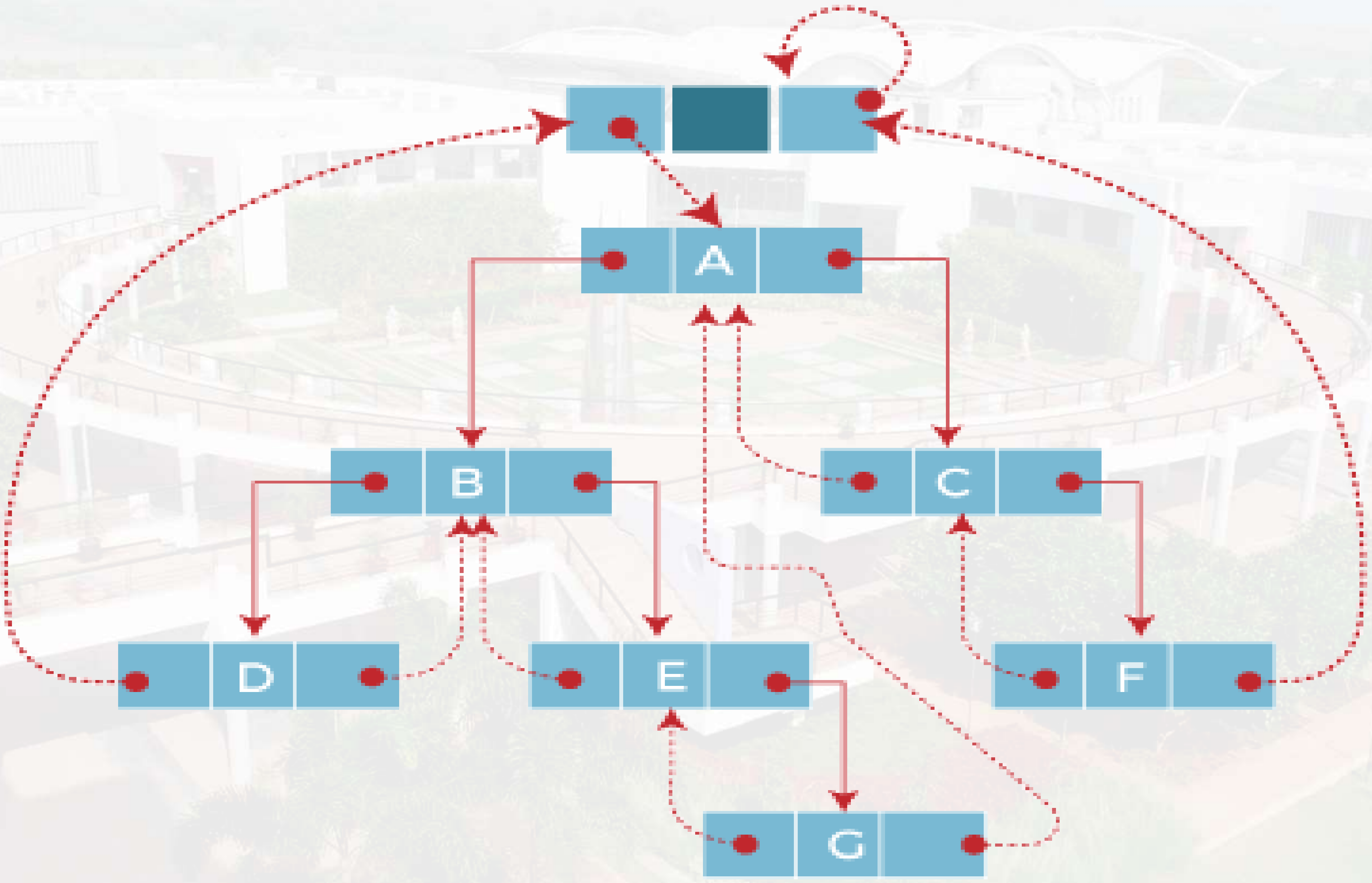


A binary tree (Inorder traversal - D, B, E, G, A, C, F)



A two - way threaded binary tree

Threaded Binary Tree



Two-way threaded - tree with header node

Insertion in Threaded Binary Tree

Insertion in Binary threaded tree is similar to insertion in binary tree but we will have to adjust the threads after insertion of each element.

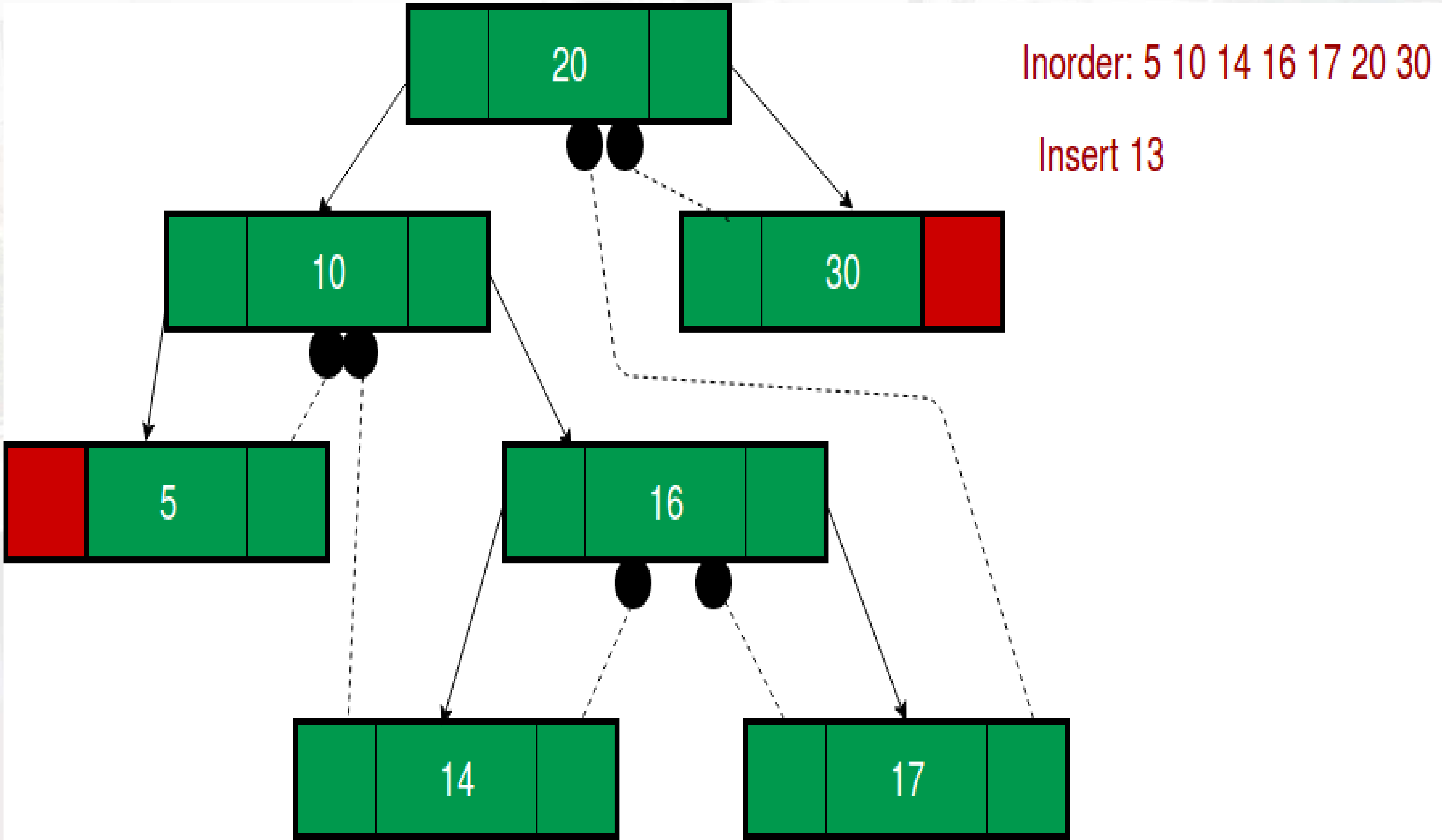
Case 1: Insertion in empty tree

Case 2: When new node inserted as the left child

Case 3: When new node is inserted as the right child

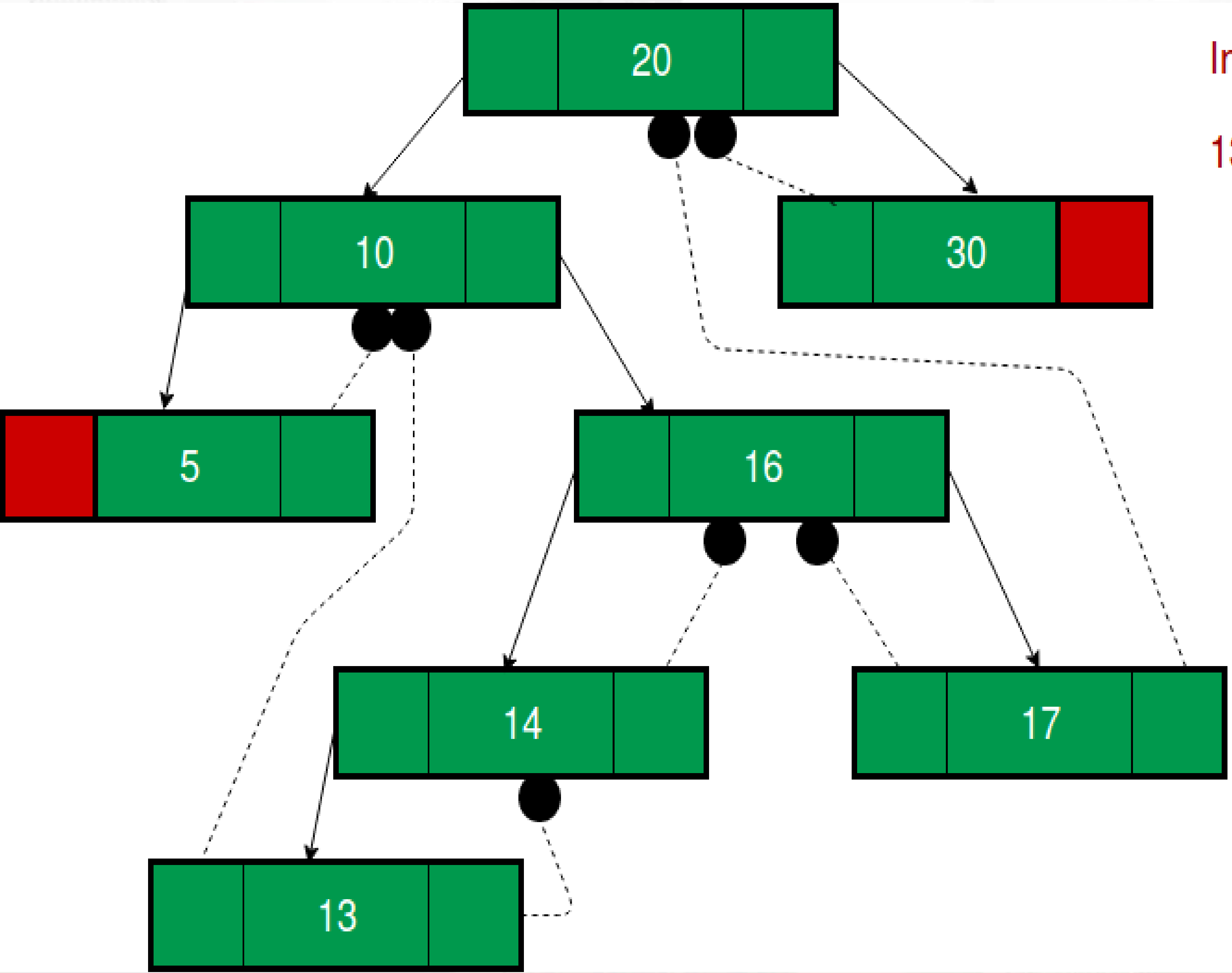
Insertion in Threaded Binary Tree

Following example show a node being inserted as left child of its parent.



Insertion in Threaded Binary Tree

After insertion of 13 element

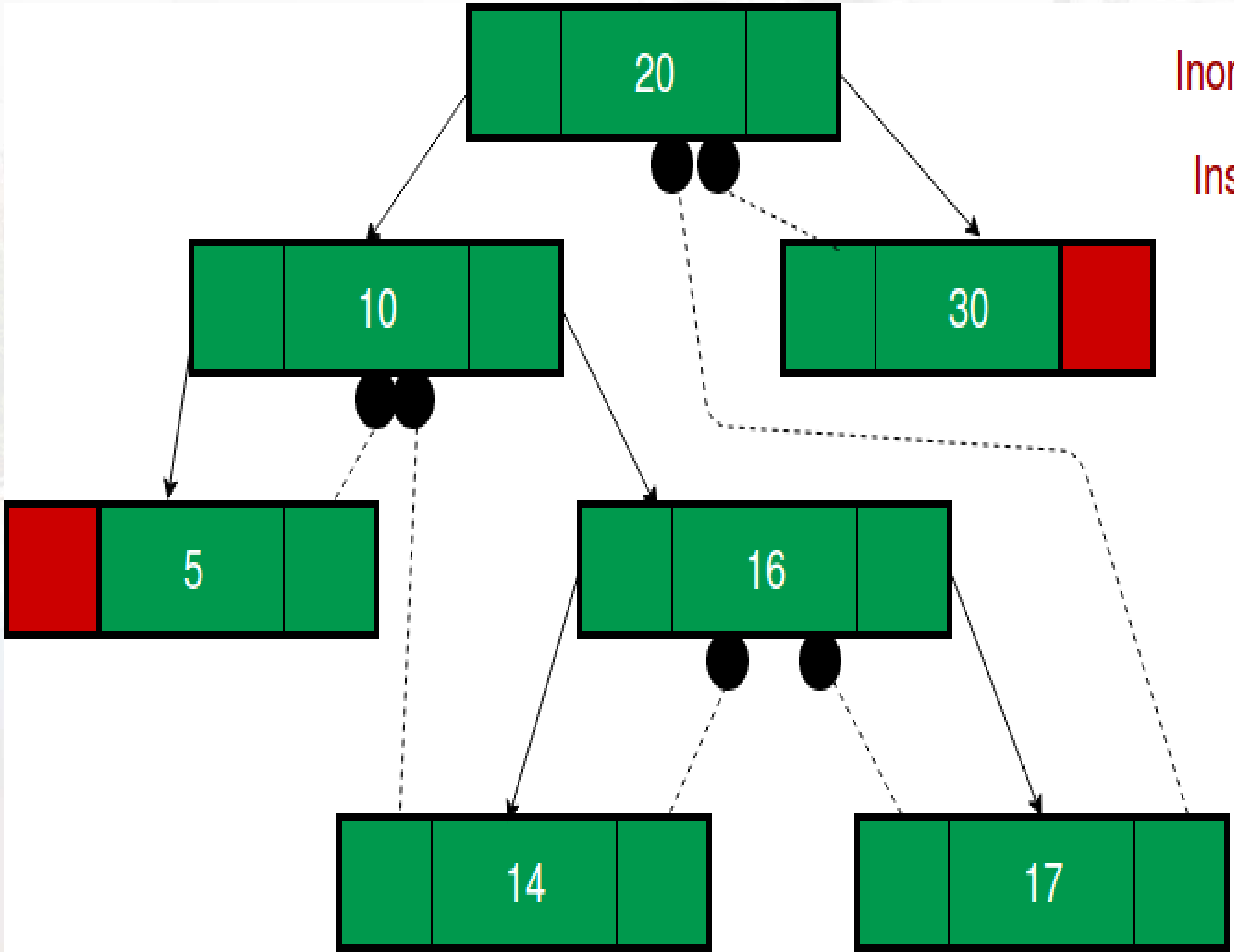


Inorder: 5 10 13 14 16 17 20 30

13 inserted as left child of 14

Insertion in Threaded Binary Tree

Following example shows a node being inserted as right child of its parent.

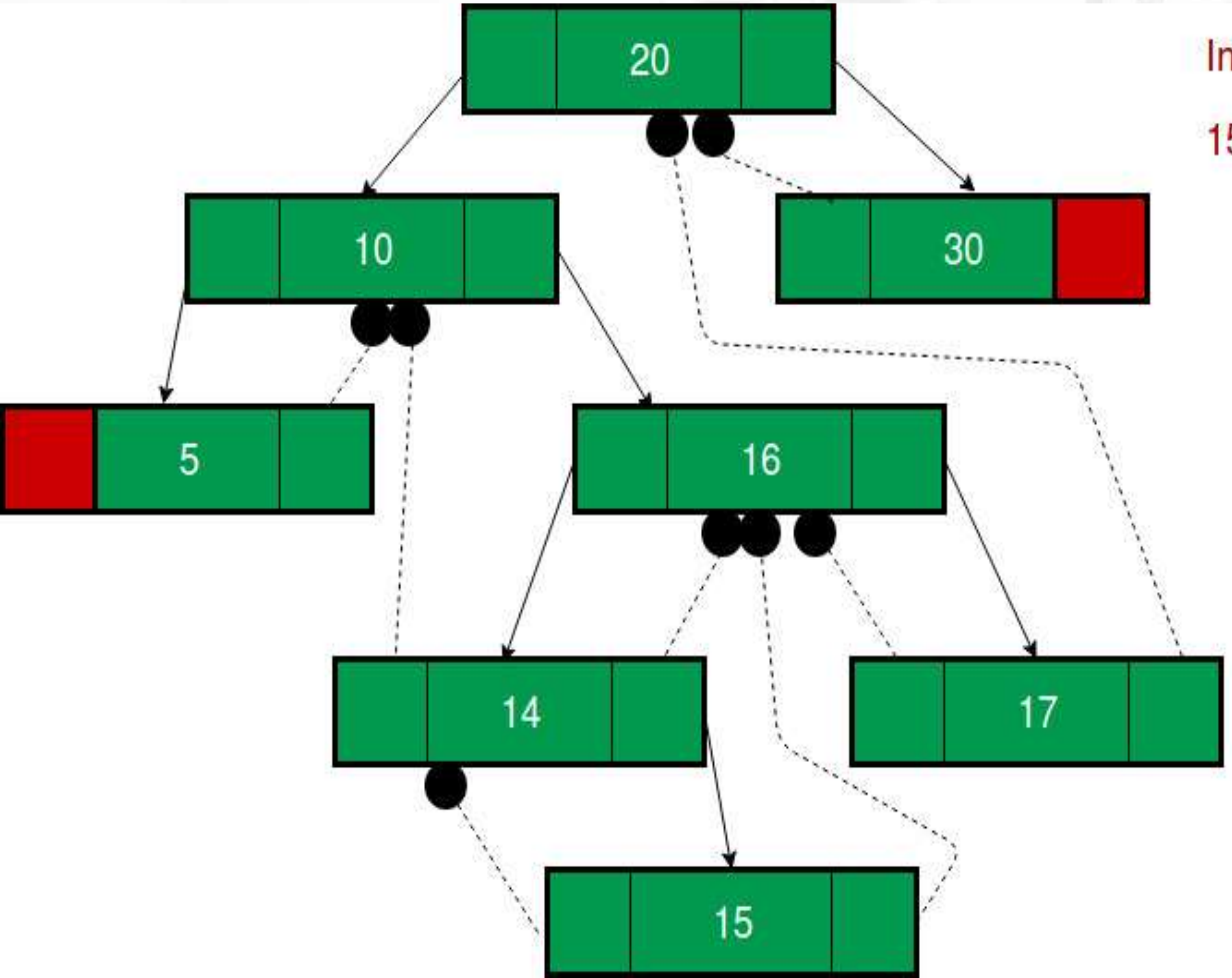


Inorder: 5 10 14 16 17 20 30

Insert 15

Insertion in Threaded Binary Tree

After 15 inserted



Inorder: 5 10 14 16 17 20 30

15 inserted as right child of 14

Deletion in Threaded Binary Tree

In deletion, first the key to be deleted is searched, and then there are different cases for deleting the Node in which key is found.

Case A: Leaf Node need to be deleted

Case B: Node to be deleted has only one child

Case C: Node to be deleted has two children

Deletion in Threaded Binary Tree

Case A: Leaf Node need to be deleted

In BST, for deleting a leaf Node the left or right pointer of parent was set to NULL. Here instead of setting the pointer to NULL it is made a thread.

If the leaf Node is to be deleted is left childs of its parent then after deletion, left pointer of parent should become a thread pointing to its predecessor of the parent Node after deletion.

Deletion in Threaded Binary Tree

Case B: Node to be deleted has only one child

After deleting the Node as in a BST, the inorder successor and inorder predecessor of the Node are found out.

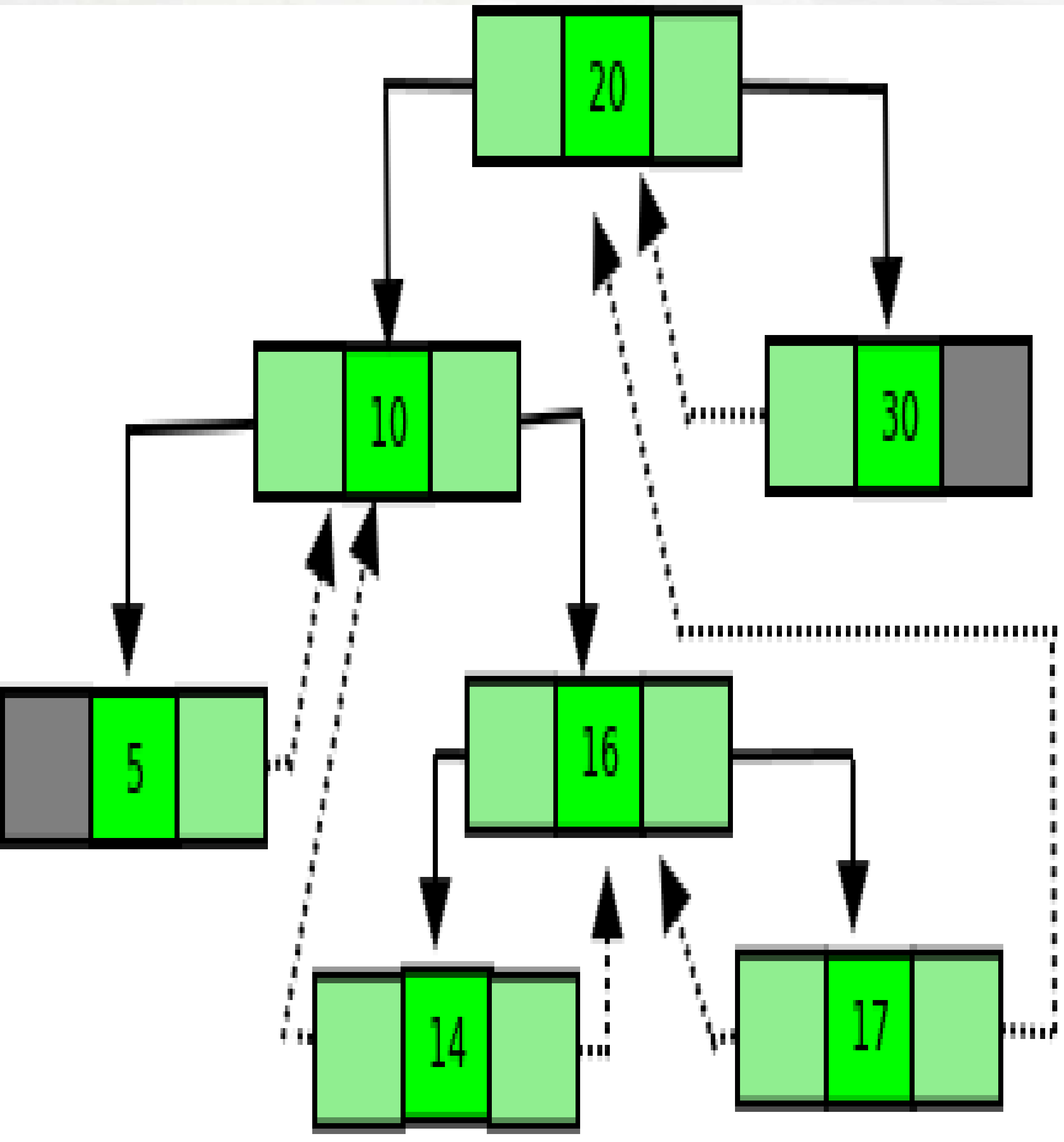
Deletion in Threaded Binary Tree

Case C: Node to be deleted has two children :

We find inorder successor of Node ptr (Node to be deleted) and then copy the information of this successor into Node ptr. After this inorder successor Node is deleted using either Case A or Case B

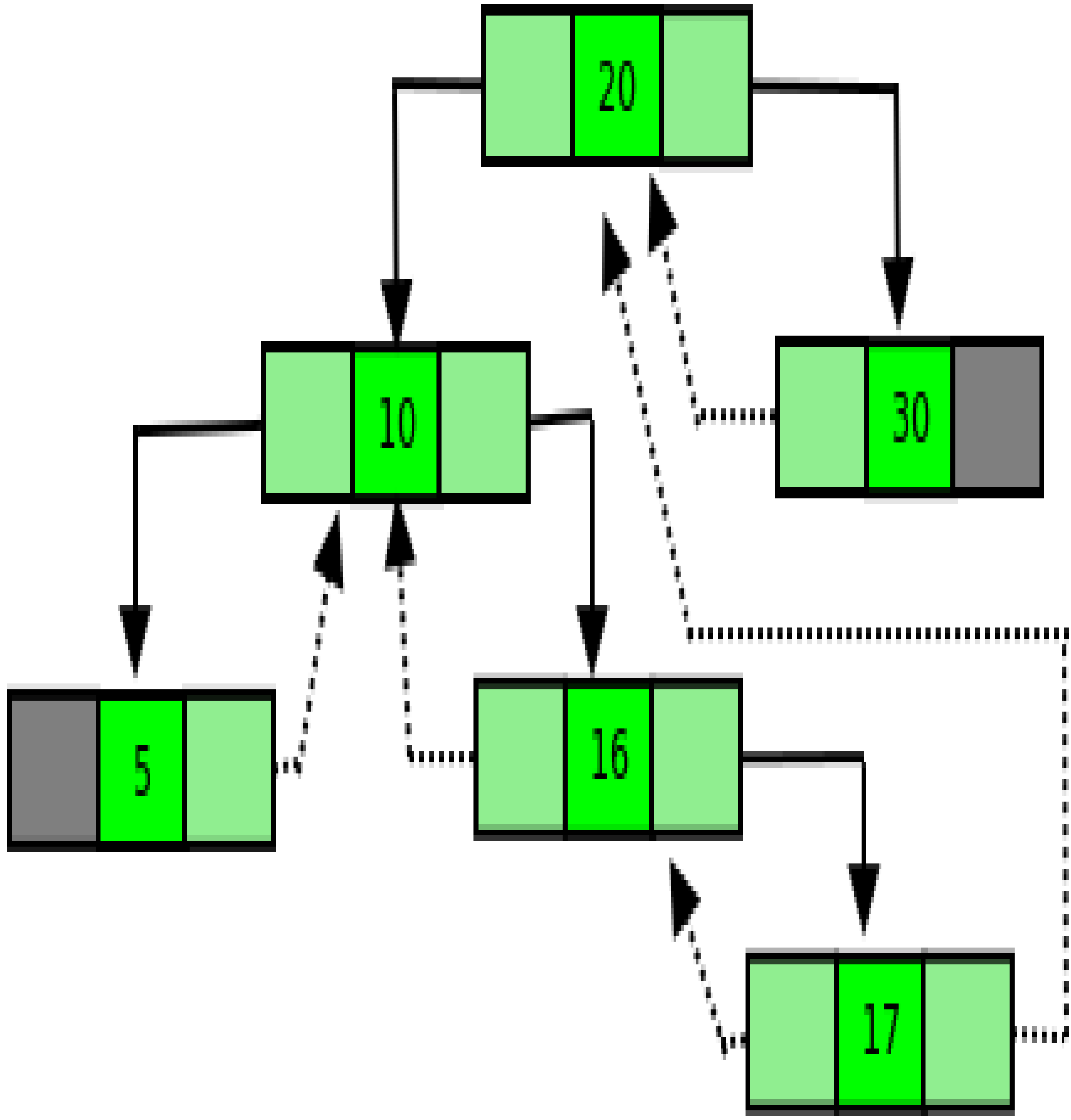
Deletion in Threaded Binary Tree

Case A: Leaf Node need to be deleted



Delete 14

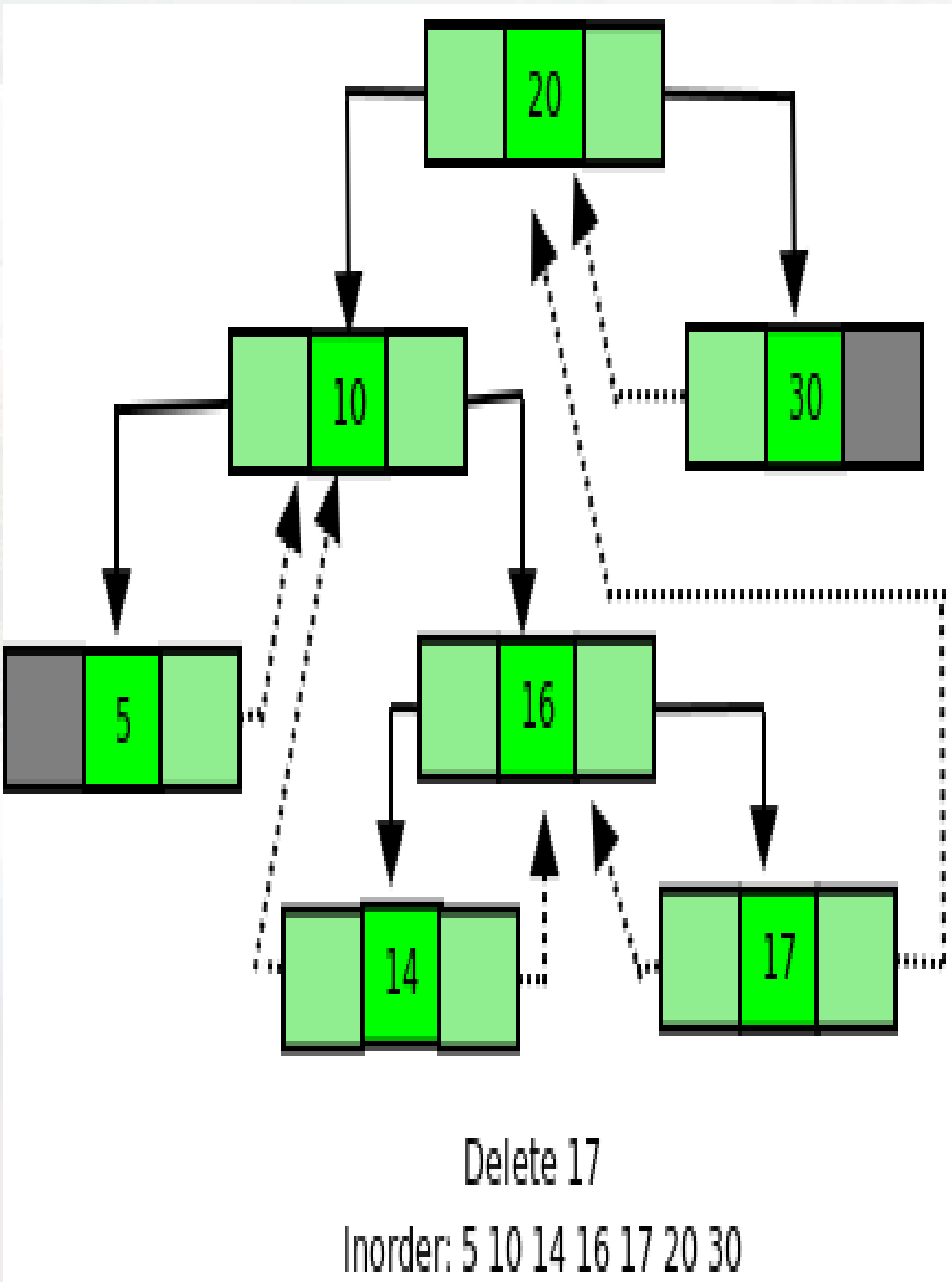
Inorder: 5 10 14 16 17 20 30

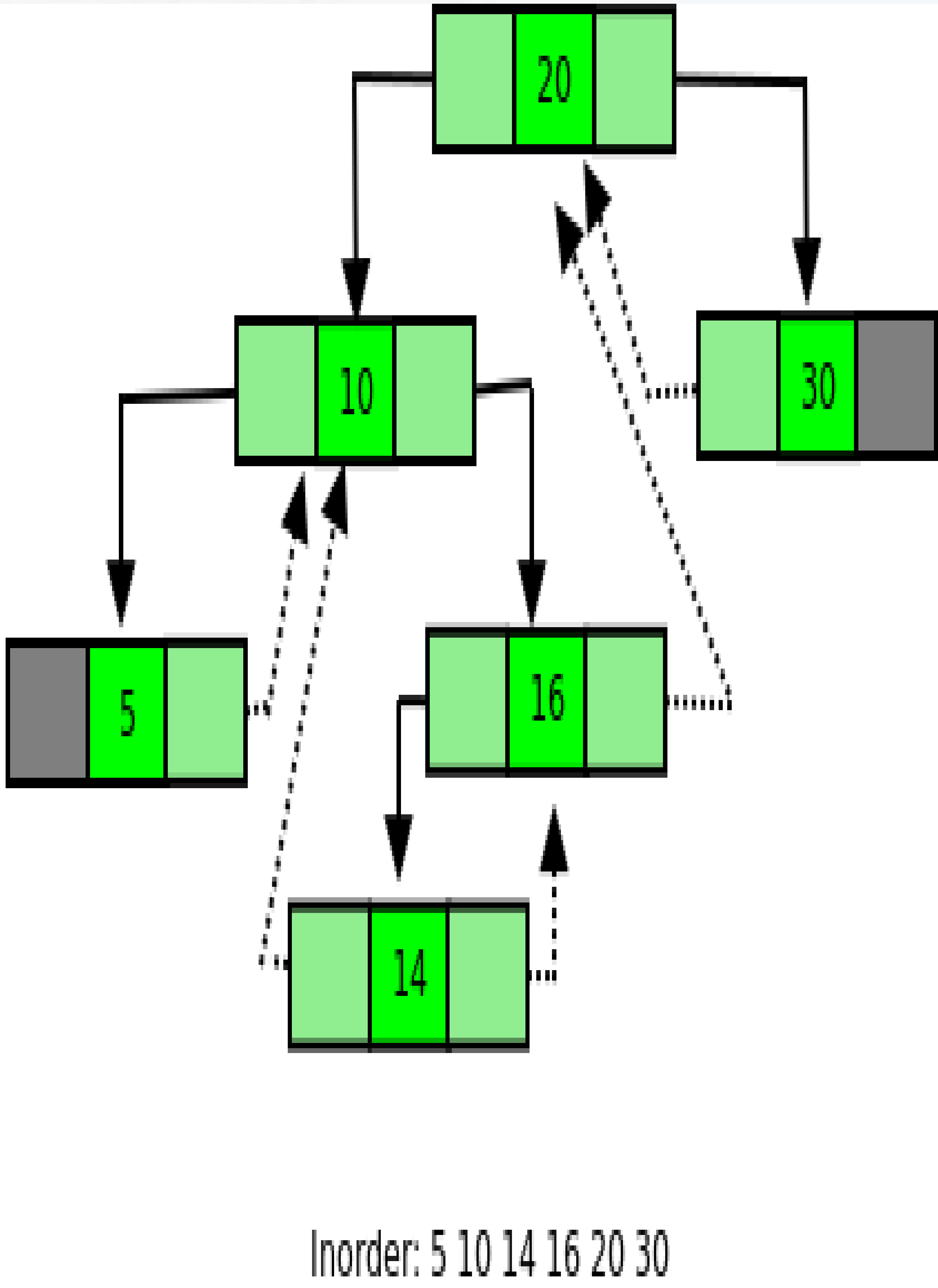


Inorder: 5 10 16 17 20 30

Deletion in Threaded Binary Tree

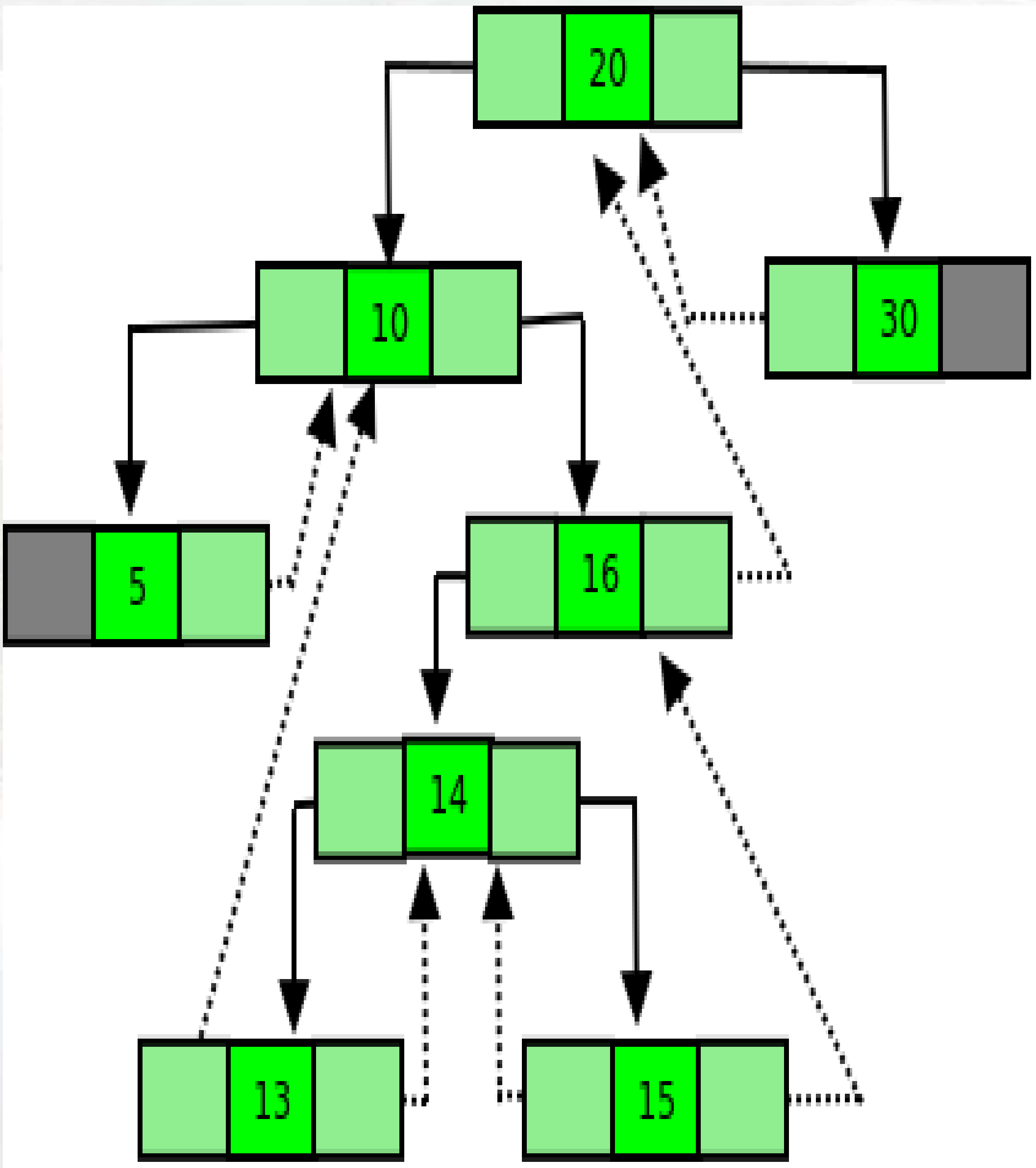
Case A: Leaf Node need to be deleted





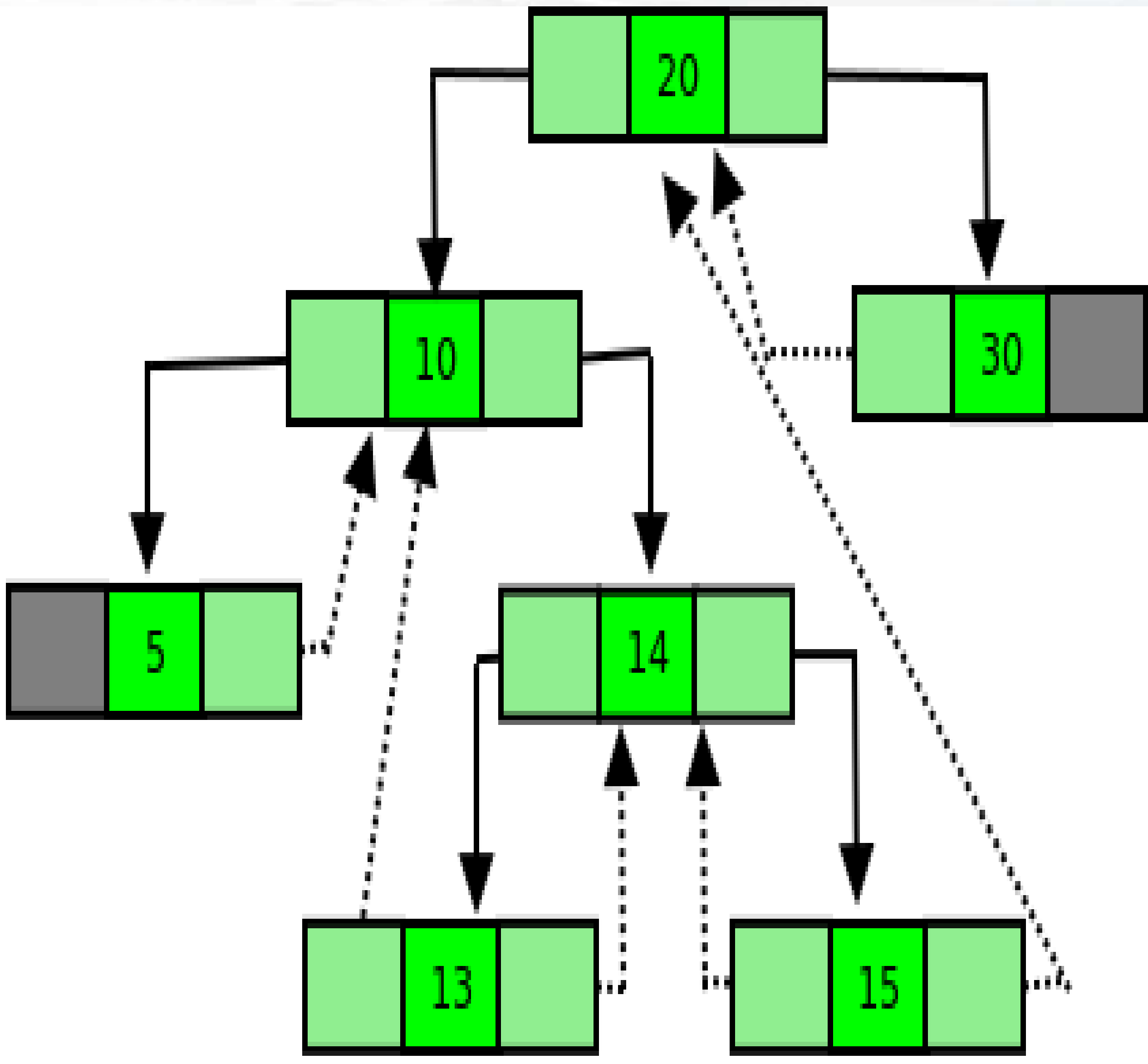
Deletion in Threaded Binary Tree

Case B: Node to be deleted has only one child



Delete 16

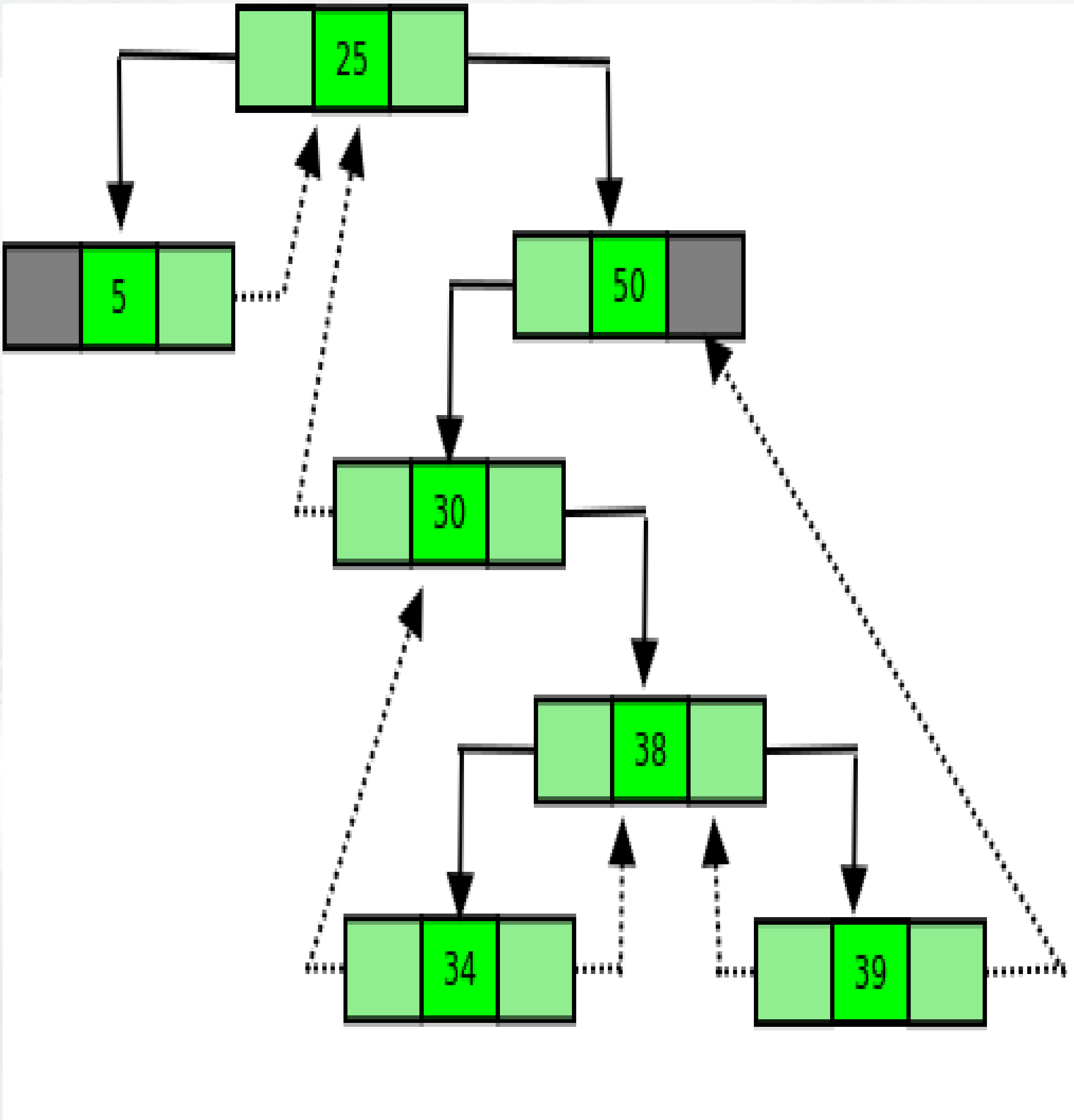
Inorder: 5 10 14 16 17 20 30



Inorder: 5 10 13 14 15 20 30

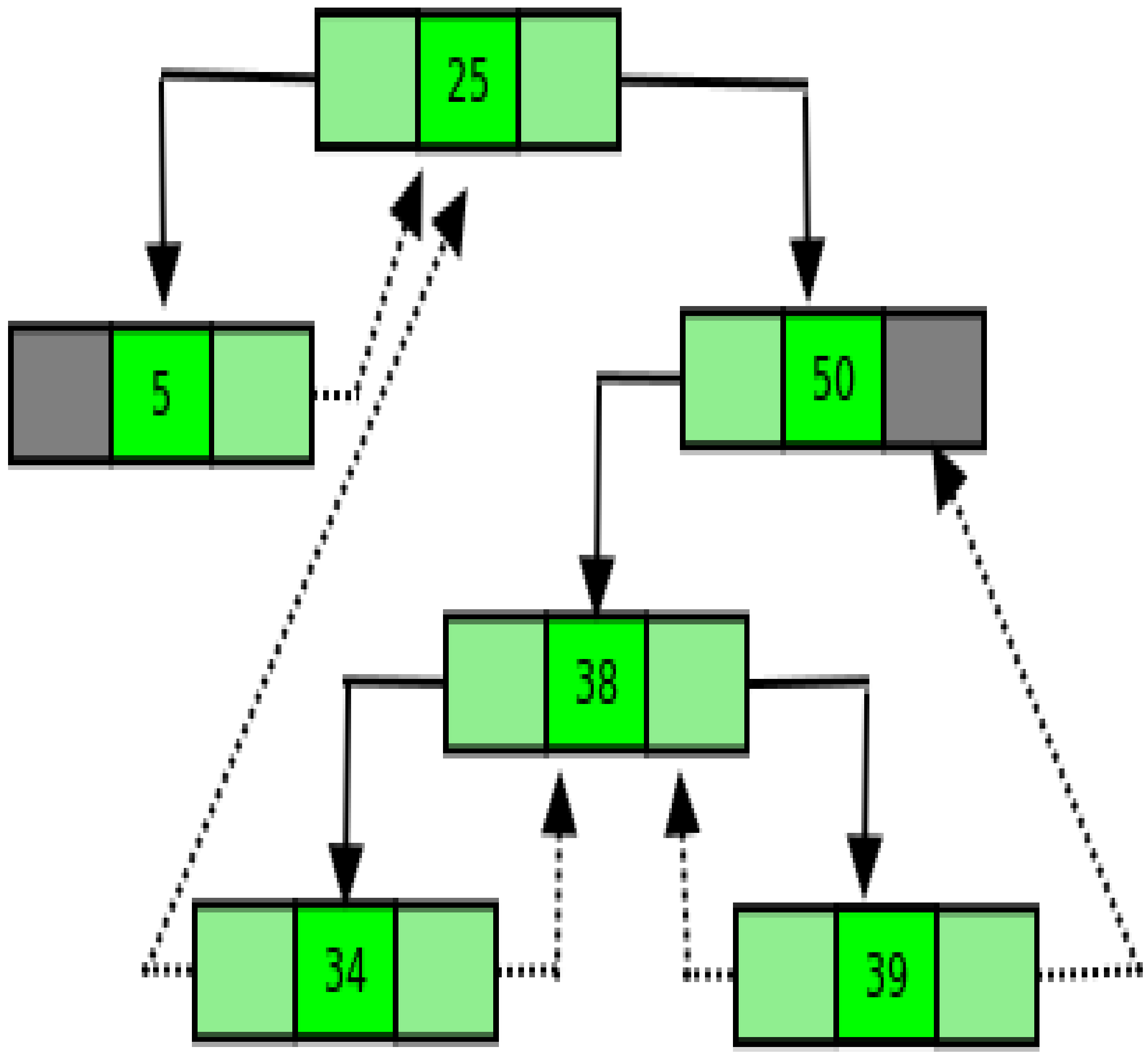
Deletion in Threaded Binary Tree

Case B: Node to be deleted has only one child



Delete 30

Inorder: 5 25 30 34 38 39 50



Inorder: 5 25 34 38 39 50

Advantages Threaded Binary Tree

1. In this Tree it enables linear traversal of elements.
2. It eliminates the use of stack as it perform linear traversal, so save memory.
3. Enables to find parent node without explicit use of parent pointer
4. Threaded tree give forward and backward traversal of nodes by in-order fashion
5. Nodes contain pointers to in-order predecessor and successor.
6. In threaded binary tree there is no NULL pointer present. Hence memory wastage in occupying NULL links is avoided.
7. There is no need of stack while traversing the tree, because using thread links we can reach to previously visited nodes.

Disadvantages Threaded Binary Tree

1. This makes the **Tree more complex** .
2. Insertion and deletion **operation becomes more difficult.**
3. **Tree traversal algorithm becomes difficult**.
4. Memory required to store a node increases. Each node has to store the information whether the links is normal links or threaded links.

Disadvantages Threaded Binary Tree

5. When implemented, the threaded binary tree needs to maintain the for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.

6. Insertion into and deletion from a threaded binary tree are **more time consuming** since both threads and ordinary links need to be maintained.

Applications Threaded Binary Tree

1. **Expression parsing**: Easy evaluation of mathematical expressions through the use of expression parsing.
2. **Database indexing**: Finding information quickly in indexed databases.
3. **Threaded in-order traversal**: Ordered threads implementing quick and responsive user interfaces are crucial.
4. **Symbol table management:** In a compiler or interpreter, threaded binary trees can be used to store and manage symbol tables for variables and functions.

Applications Threaded Binary Tree

5. Navigation of hierarchical data: In certain applications, threaded binary trees can be used to navigate hierarchical data structures, such as file systems or web site directories.

6. Fast Searching and Retrieval: Threaded binary trees enable faster navigation, improving the performance of search operations

7. Threaded Tree-based Iterators: Threaded trees are useful for implementing efficient iterators for various tree traversal orders

8. Binary Search Tree Operations: Threaded trees enhance efficiency in operations like finding minimum/maximum elements or predecessors/successor

References

1. https://www.javatpoint.com/depth-first-search-algorithm
2. https://www.javatpoint.com/breadth-first-search-algorithm
3. https://www.javatpoint.com/threaded-binary-tree
4. https://www.codingninjas.com/studio/library/understanding-threaded-binary-trees

THANK YOU!!!

My Blog : https://anandgharu.wordpress.com/

Email : gharu.anand@gmail.com