Unit-3 Tree Data Structure

1. what is tree in data structure.?

In computer science, a tree is a non-linear data structure that is used to store hierarchical data. It consists of nodes connected by edges, where each node represents a value or an object, and each edge represents a relationship between the nodes.

A tree has a root node at the top, which has zero or more child nodes below it. Each child node can have its own children, creating a branching structure that resembles a tree. Nodes with no children are called leaf nodes.

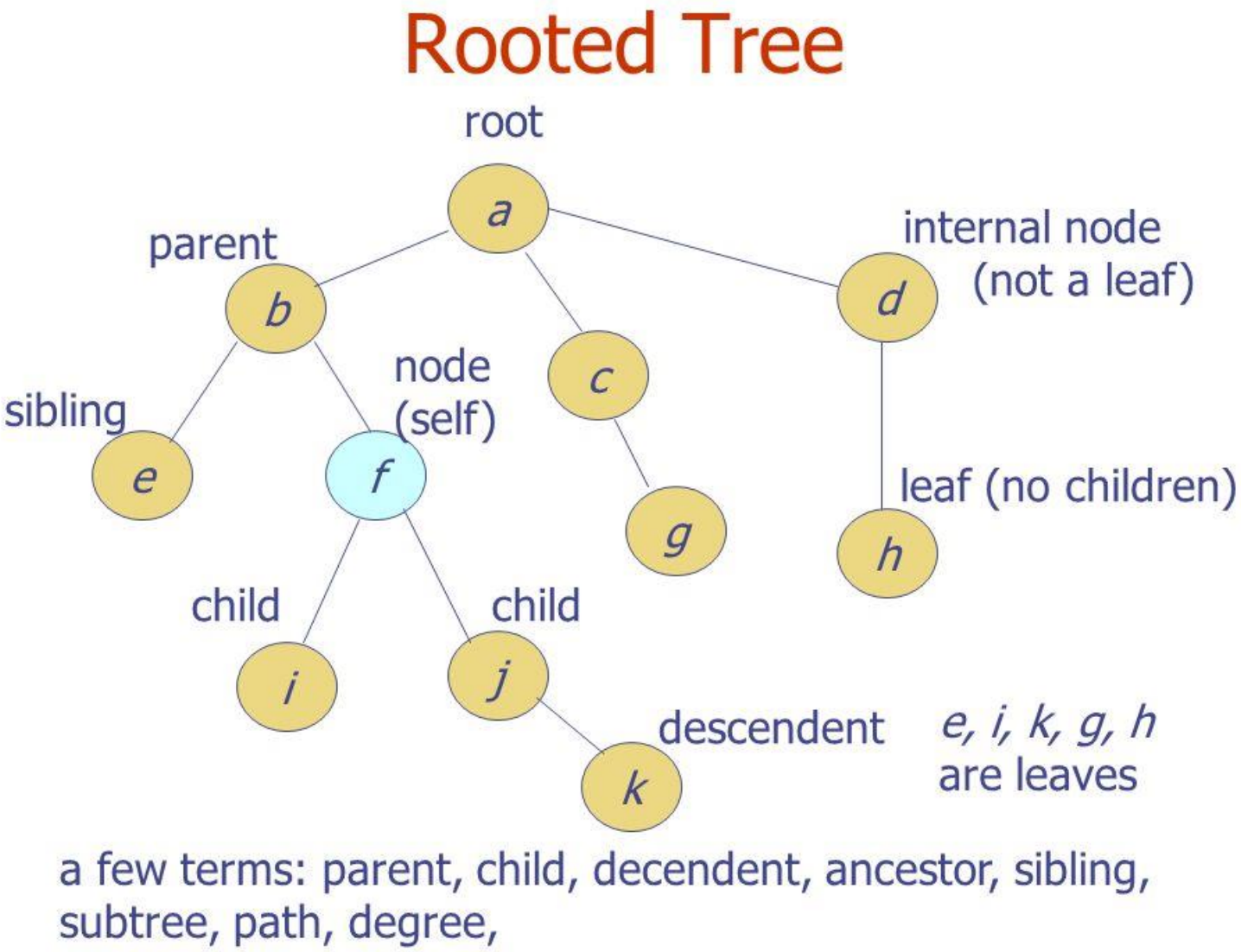
Trees are often used in algorithms and computer programs for organizing and accessing data efficiently. Some common examples of trees in computer science include binary trees, AVL trees, B-trees, and heaps.

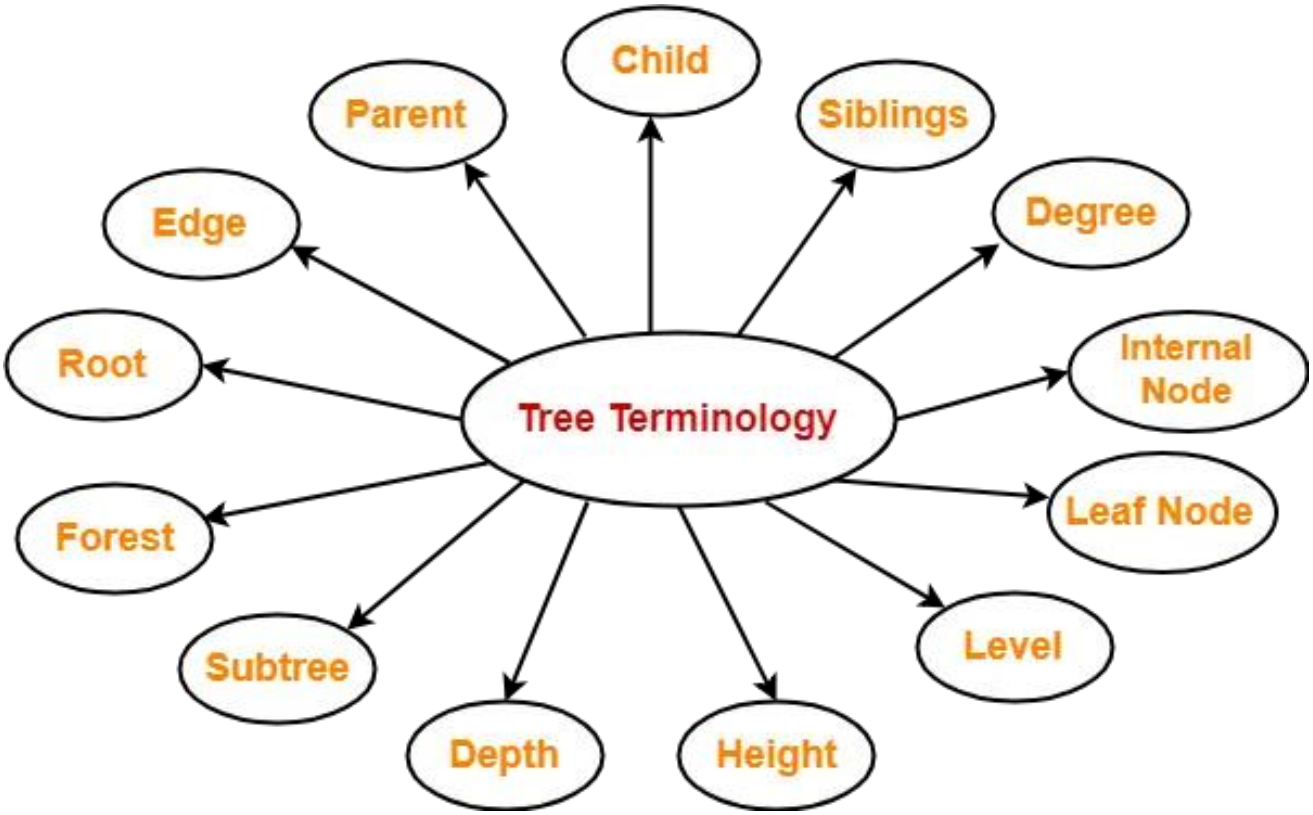
2. terminology and properties of a tree data structure.?

Here are some common terminology and properties of a tree data structure:

1. Node: An individual element of a tree that contains a value and a pointer to its child nodes, if any.
2. Root: The topmost node in a tree, which has no parent node.
3. Parent: A node that has one or more child nodes.
4. Child: A node that is directly connected to a parent node.
5. Siblings: Nodes that share the same parent node.
6. Leaf: A node that has no child nodes.
7. Height: The length of the longest path from the root node to any leaf node in the tree.
8. Depth: The length of the path from a node to the root node.
9. Degree: The number of children of a node.
10. Binary tree: A tree where each node has at most two child nodes.
11. Balanced tree: A tree where the height of the left and right subtrees of every node differ by at most one.
12. Full tree: A tree where every node has either 0 or 2 children.

13. Complete tree: A tree where all levels except possibly the last level are completely filled, and all nodes are as far left as possible on the last level.
14. Traversal: The process of visiting each node in a tree exactly once, in a particular order. Some common traversal algorithms include in-order, pre-order, and post-order traversal.
15. Binary search tree: A binary tree where the left child of a node contains a value less than the node's value, and the right child contains a value greater than the node's value. This property makes searching for values in the tree more efficient than in an unsorted binary tree.





3. Types of tree data structure.?

A. Binary search tree=: binary search tree is a non-linear data structure in which a node an be connected with number of nodes. It is a node-based data structure. A node can be represented in binary search tree with three fields one is data field and another two is left child and right child. A node can be connected with at most two-child node in a binary search tree so the node contains two pointers (left child and right child.)

In a binary search tree, the value for each node in left subtree is must less than to the value of root node and the value for each node of right subtree is must be grater to the root node.

A node can be create using user define data type known as struct shown as below.

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

Searching in binary search tree=:

Searching means to find/locate a specific node/element in a binary search tree.
The searching an element is easy because the element in binary search tree stored in an order, the steps to search an element in a binary search tree is listed as below-:

- ➔ First compare the searched element with the root node.
- ➔ If it is not matched, then check if whether the item is less to the root element if it is similar/smaller to the root node/element than move to the left subtree.
- ➔ If root is unmatched with target element than return the node location

➔ If the searched element is greater to the root node, then move to the right subtree.

➔ Repeat above procedure recursively until the match not found.

➔ If the element is not found or not present in tree, then return null.

Algorithm for search element in Binary search tree-:

```
Search(root, item) {
    Step-1:
    If(item==root->data) or (root==null)
        Return null;
    else if(item<root->data)
            return Search(root->left, item);
    else if(item>root)
        return Search(root->right, item);
    else
        return null;
Step-1:
            ENDIF;
```

Here's an example code in C to search for an element in a binary search tree:

```
#include <stdio.h>
#include <stdlib.h>

// Definition of a Binary Search Tree (BST) node
struct node {
    int data;
    struct node* left;
    struct node* right;
};

// Function to create a new BST node
struct node* create_node(int value) {
    struct node* new_node = (struct
node*)malloc(sizeof(struct node));
    new_node->data = value;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

// Function to insert a value into a BST
struct node* insert(struct node* root, int value) {
```

```
    if (root == NULL) {
        return create_node(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    }
    else {
        root->right = insert(root->right, value);
    }
    return root;
}

// Function to search for a value in a BST
struct node* search(struct node* root, int value) {
    if (root == NULL || root->data == value) {
        return root;
    }
    if (value < root->data) {
        return search(root->left, value);
    }
    else {
        return search(root->right, value);
```

```
    }
}

// Main function
int main() {
    struct node* root = NULL;
    int values[] = {5, 2, 8, 1, 3, 7, 9};
    int n = sizeof(values) / sizeof(values[0]);

    // Insert the values into the BST
    for (int i = 0; i < n; i++) {
        root = insert(root, values[i]);
    }

    // Search for a value in the BST
    int search_value = 7;
    struct node* result = search(root, search_value);

    // Check if the value was found
    if (result == NULL) {
        printf("%d was not found in the BST\n", search_value);
    }
```

```
    else {
        printf("%d was found in the BST\n", result->data);
    }

    return 0;
}
```

In this code, we define a struct node to represent a node in a binary search tree. We then create a function create_node to create a new node with a given value, and a function insert to insert a value into the BST.

The main function creates a BST with the values {5, 2, 8, 1, 3, 7, 9}, and then searches for the value 7 using the search function. The search function recursively searches for the value in the BST and returns the node if found, or NULL if not found.

Finally, the main function checks if the value was found and prints a message accordingly.

Insert element in binary search tree

To insert an element into a binary search tree, you can follow these steps:

- ➔ Start at the root node of the tree.
- ➔ Compare the value of the element you want to insert with the value of the current node.
- ➔ If the element is less than the current node's value, move to the left child of the node. If the left child is null, insert the new element as the left child of the current node. Otherwise, repeat step 2 with the left child as the current node.
- ➔ If the element is greater than the current node's value, move to the right child of the node. If the right child is null, insert the new element as the right child of the current node. Otherwise, repeat step 2 with the right child as the current node.
- ➔ If the element is equal to the current node's value, the element is already in the tree, so do nothing.

Here the Algorithm to insert an element in binary search tree