Ex : 6

Title: Design, Develop and Implement a menu driven Programin C for traversing a tree and search a given item.

Problem Description: Create a hierarchical data structure and perform efficient insertion and search.

Given a set of data items,

Create a hierarchical data structure of N nodes, with certain order.Implement all Traversal of the created data structure and output the nodes visited.

Iii) Perform efficient Search for an element (KEY) in the created data structure and report the appropriate message.

Method: Make use of binary search tree. You can use recursion or iterative techniques.

Theory Reference: Module 4

Explanation:

Hierarchical Data Structure: The program creates a binary search tree, where each node contains a data item, and links to its left and right children.

Efficient Insertion: Nodes are inserted into the tree while maintaining the properties of the BST, where left children are less than the parent node, and right children are greater.

Tree Traversals: The program supports three types of tree traversals:

- **Inorder Traversal**: Visits nodes in left-root-right order, yielding sorted output.
- **Preorder Traversal**: Visits nodes in root-left-right order.
- **Postorder Traversal**: Visits nodes in left-right-root order.

Search Functionality: Users can search for a specific key in the BST and receive feedback on whether the key exists.

Algorithm:

Step 1: Initialize

Define a self-referential node.

Define a structure BST with the following fields:

- data: an integer to store the value of the node.
- left: a pointer to the left child node.
- right: a pointer to the right child node.

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

Step 2. Insertion Function

- **Function**: insert(node *root, int key)
    1. If root is NULL, allocate memory for a new node, set its data to key, and set left and right to NULL.
    2. If key is less than root->data, recursively call insert on the left child.
    3. If key is greater than root->data, recursively call insert on the right child.
    4. Return the root.

3. Inorder Traversal Function

- **Function**: inorder(node *root)
    1. If root is not NULL, recursively call inorder on the left child.
    2. Print root->data.
    3. Recursively call inorder on the right child.

4. Preorder Traversal Function

- **Function**: preorder(node *root)
    1. If root is not NULL, print root->data.
    2. Recursively call preorder on the left child.
    3. Recursively call preorder on the right child.

5. Postorder Traversal Function

- **Function**: postorder(node *root)
    1. If root is not NULL, recursively call postorder on the left child.
    2. Recursively call postorder on the right child.
    3. Print root->data.

6. Search Function

- **Function**: search(node *root, int key)
    1. If root is NULL, print "key not found".
    2. If root->data equals key, print "key found".
    3. If key is less than root->data, recursively call search on the left child.
    4. If key is greater, recursively call search on the right child.

7. Main Function

- **Function**: main()
    1. Declare variables for the number of nodes, key, and user choice.
    2. Initialize root as NULL.
    3. Prompt the user to enter the number of nodes (n).
    4. Loop n times to read node values and insert them into the BST.
    5. Enter an infinite loop to display a menu for tree operations:
        - If the user chooses:
            - 1: Perform and print the result of the inorder traversal.
            - 2: Perform and print the result of the preorder traversal.
            - 3: Perform and print the result of the postorder traversal.
            - 4: Prompt for a key and search for it in the BST.
            - 5: Exit the program.
            - Any other input: Print "invalid choice".