12. IMPLEMENTATION OF BINARY TREE

Preamble

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children. The binary tree has following properties.

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2h+1 -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is log2(n+1) - 1.

Steps – Creation of Binary Tree

Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50

First, we have to insert 45 into the tree as the root of the tree.

Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.

Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Steps – Searching the Binary Tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

- First, compare the element to be searched with the root element of the tree.
- If root is matched with the target element, then return the node's location.
- If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
- If it is larger than the root element, then move to the right subtree.
- Repeat the above procedure recursively until the match is found.
- If the element is not found or not present in the tree, then return NULL.

Steps – Deletion in Binary Tree

- Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.
- Replace the deepest rightmost node's data with the node to be deleted.
- Then delete the deepest rightmost node.

Implementation in C

```
#include <stdio.h>

// Node structure to define the structure of the node
typedef struct Node
{
	int data;
	struct Node *left, *right;
} Node;

// Function to create a new node
Node* newNode(int val)
{
	Node* temp = (Node*)malloc(sizeof(Node));
	temp->data = val;
	temp->left = temp->right = NULL;
	return temp;
}

// Function to insert nodes
Node* insert(Node* root, int data)
{
	// If tree is empty, new node becomes the root
	if (root == NULL)
	{
		root = newNode(data);
		return root;
```

```
    }

    // Queue to traverse the tree and find the position to insert the node
    Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;
    while (front != rear)
    {
        Node* temp = queue[front++];
        // Insert node as the left child of the parent node
        if (temp->left == NULL)
        {
            temp->left = newNode(data);
            break;
        }
        // If the left child is not null, push it to the queue
        else
            queue[rear++] = temp->left;
            // Insert node as the right child of parent node
        if (temp->right == NULL)
        {
            temp->right = newNode(data);
            break;
        }
        // If the right child is not null, push it to the queue
        else
            queue[rear++] = temp->right;
    }
    return root;
}

/* Function to delete the given deepest node (d_node) in binary tree */
```

```
void deletDeepest(Node* root, Node* d_node)
{
      Node* queue[100];
      int front = 0, rear = 0;
      queue[rear++] = root;
      // Do level order traversal until last node
      Node* temp;
      while (front != rear)
      {
            temp = queue[front++];
            if (temp == d_node)
            {
                  temp = NULL;
                  free(d_node);
                  return;
            }
            if (temp->right)
            {
                  if (temp->right == d_node)
                  {
                        temp->right = NULL;
                        free(d_node);
                        return;
                  }
            else
                  queue[rear++] = temp->right;
            }
            if (temp->left)
            {
                  if (temp->left == d_node)
                  {
                        temp->left = NULL;
```

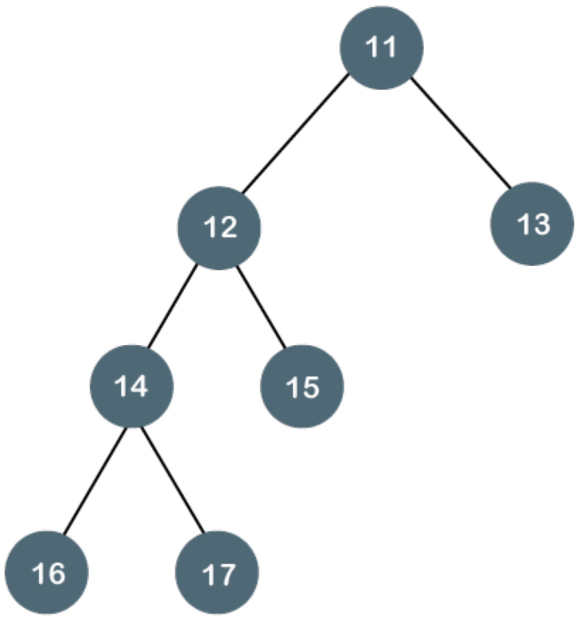
```
                        free(d_node);
                        return;
                }
                else
                        queue[rear++] = temp->left;
            }
        }
}

/* Function to delete element in binary tree */
Node* deletion(Node* root, int key)
{
        if (!root)
            return NULL;
        if (root->left == NULL && root->right == NULL)
        {
            if (root->data == key)
                return NULL;
            else
                return root;
        }
        Node* queue[100];
        int front = 0, rear = 0;
        queue[rear++] = root;
        Node* temp;
        Node* key_node = NULL;
        // Do level order traversal to find deepest node (temp) and node to be
        deleted (key_node)
        while (front != rear)
        {
            temp = queue[front++];
            if (temp->data == key)
```

```
                key_node = temp;
            if (temp->left)
                queue[rear++] = temp->left;
            if (temp->right)
                queue[rear++] = temp->right;
        }
        if (key_node != NULL)
        {
            int x = temp->data;
            key_node->data = x;
            deletDeepest(root, temp);
        }
        return root;
}

// Inorder tree traversal (Left - Root - Right)
void inorderTraversal(Node* root)
{
        if (!root)
            return;
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
}

// Preorder tree traversal (Root - Left - Right)
void preorderTraversal(Node* root)
{
        if (!root)
            return;
        printf("%d ", root->data);
        preorderTraversal(root->left);
```

```
        preorderTraversal(root->right);
}

// Postorder tree traversal (Left - Right - Root)
void postorderTraversal(Node* root)
{
        if (root == NULL)
                return;
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
}

// Function for Level order tree traversal
void levelorderTraversal(Node* root)
{
        if (root == NULL)
                return;
        // Queue for level order traversal
        Node* queue[100];
        int front = 0, rear = 0;
        queue[rear++] = root;
        while (front != rear)
        {
                Node* temp = queue[front++];
                printf("%d ", temp->data);
                // Push left child in the queue
                if (temp->left)
                        queue[rear++] = temp->left;
                // Push right child in the queue
                if (temp->right)
                        queue[rear++] = temp->right;
```

```
    }
}

/* Driver function to check the above algorithm. */
int main()
{
    Node* root = NULL;
    // Insertion of nodes
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    printf("Preorder traversal: ");
    preorderTraversal(root);
    printf("\nInorder traversal: ");
    inorderTraversal(root);
    printf("\nPostorder traversal: ");
    postorderTraversal(root);
    printf("\nLevel order traversal: ");
    levelorderTraversal(root);
    // Delete the node with data = 20
    root = deletion(root, 20);
    printf("\nInorder traversal after deletion: ");
    inorderTraversal(root);
    return 0;
}
```

Sample Input and Output


11
12
13
14
15
16
17