Laboratory 7

Title of the Laboratory Exercise: Binary trees

1. Introduction and Purpose of Experiment

   Linear organization used on arrays, vectors, stacks and queues become inefficient in some applications. Then we choose the structures which provide non-linear organization. Binary tree is a non-linear data structure used in many applications. This experiment introduces binary search trees and its applications.

2. Aim and Objectives

   Aim

   - To develop Binary search tree ADT

   Objectives

   At the end of this lab, the student will be able to

   - Design binary tree ADT
   - Use binary tree ADT and illustrate binary tree traversals

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

<u>Implementation:</u>

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
     int data;
     struct node* left;
     struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
     struct node* node = (struct node*)
                                  malloc(sizeof(struct node));
     node->data = data;
     node->left = NULL;
     node->right = NULL;

     return(node);
```

```
}

/* Given a binary tree, print its nodes according to the
  "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
     if (node == NULL)
        return;

     // first recur on left subtree
     printPostorder(node->left);

     // then recur on right subtree
     printPostorder(node->right);

     // now deal with the node
     printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first recur on left child */
     printInorder(node->left);

     /* then print the data of node */
     printf("%d ", node->data);

     /* now recur on right child */
     printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first print data of node */
     printf("%d ", node->data);

     /* then recur on left sutree */
     printPreorder(node->left);

     /* now recur on right subtree */
     printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
     struct node *root  = newNode(1);
     root->left             = newNode(2);
     root->right           = newNode(3);
```

```
    root->left->left     = newNode(4);
    root->left->right   = newNode(5);

    printf("\nPreorder traversal of binary tree is \n");
    printPreorder(root);

    printf("\nInorder traversal of binary tree is \n");
    printInorder(root);

    printf("\nPostorder traversal of binary tree is \n");
    printPostorder(root);

    getchar();
    return 0;
}
```

5. Presentation of Results

```
Preorder traversal of binary tree is

1 2 4 5 3

Inorder traversal of binary tree is

4 2 5 1 3

Postorder traversal of binary tree is

4 5 2 3 1
```

6. Analysis and Discussions

Time Complexity: O(n)

Let us see different corner cases.

Complexity function T(n) — for all problem where tree traversal is involved — can be defined as:

$$T(n) = T(k) + T(n - k - 1) + c$$

Where k is the number of nodes on one side of root and n-k-1 on the other side.

Let’s do an analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.

$$T(n) = T(0) + T(n-1) + c$$

$$T(n) = 2T(0) + T(n-2) + 2c$$

$$T(n) = 3T(0) + T(n-3) + 3c$$

$$T(n) = 4T(0) + T(n-4) + 4c$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$

$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots.$$

$$T(n) = (n-1)T(0) + T(1) + (n-1)c$$

$$T(n) = nT(0) + (n)c$$

Value of $T(0)$ will be some constant say d. (traversing a empty tree will take some constants time)

$$T(n) = n(c+d)$$

$$T(n) = \Theta(n)\ (Theta\ of\ n)$$

Case 2: Both left and right subtrees have equal number of nodes.

$$T(n) = 2T(|\_n/2\_|) + c$$

7. Conclusions

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.