Yash Mahajan SE IT B 04 191061

Experiment - 4

Aim : Write a C program to implement AVL tree.

Objectives: After study of this experiment, the student will be able to
- To use basic principles of programming as applied to complex data structures
- To implement the tree through programming.

Outcomes :- After study of this experiment, the student will be able to
- Formulate and Solve problems of AVL trees and it's operations
- Understand the concepts and apply the methods in basic trees.

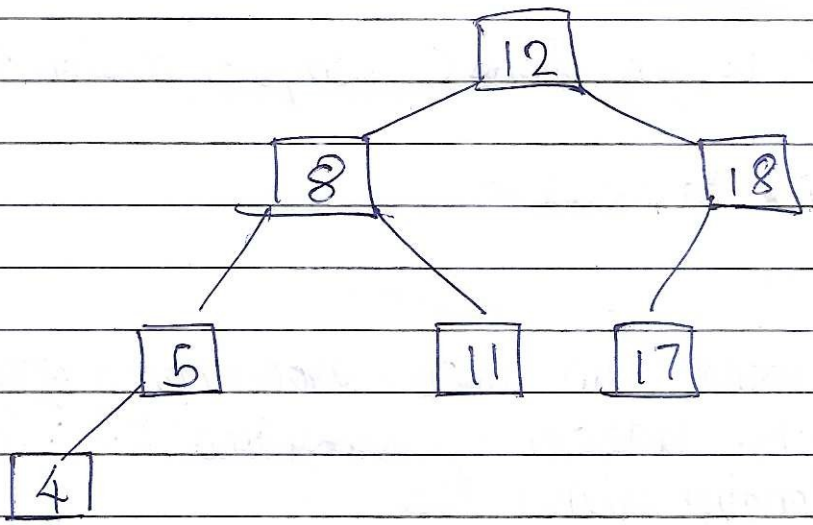
Prerequisite : Types of trees, Binary tree.

Requirements :- PC and Turbo C compiler version 3.0.

Pre-Experiment Exercise :-

AVL Tree :-
- In computer science an AVL tree (named after inventors G.M. Adelson - Velsky and E.M. Landis) is a self balancing binary tree.
- In AVL tree, the heights of the two child sub-trees of any node differ by at most one; if at any time they differ by by more than one,

rebalancing is done to restore this property.
- The structure of an AVL tree is the same as that of a binary tree but with a little difference. In it's structure, it stores an additional variable called the balance factor.
- Thus, every node has a balance factor associated with it.
- The balance factor of a node is calculated by subtracting the height of it's right sub-tree from the height of it's left sub-tree.
- Balance factor = Height (left sub tree) - Height (right sub-tree)
- A binary search tree in which every node has a balance factor of -1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.



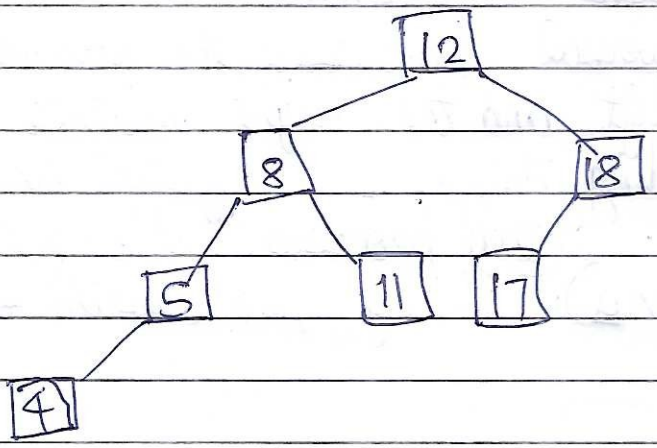
Page :
Date : / /

Yash Mahajan SE IT B 04

Operations on AVL Tree :-

- Traversing in an AVL Tree -

Traversing on AVL Tree is similar to that in a binary search tree. There are 3 ways of traversal :-



i) Inorder Traversal (LNR)

The inorder ~~travel~~ traversal processes the left ~~node~~ sub-tree first, then the root and finally the right sub-tree. The prefix 'in' ~~is~~ means that the root is processed in between the sub trees. The implementation of this algorithm is done recursively.

Inorder traversal of the given tree is ~~4,5~~.

4, 5, 8, 11, 12, 17, 18.

2) Preorder Traversal (NLR)

The preorder traversal processes the root node first, followed by the left and the right sub tree. The prefix 'pre' means that the root node is

processed before the sub trees. Implementation of this algorithm is also done recursively.

The preorder traversal of the given tree is

12, 8, 5, 4, 11, 18, 17.

3) Postorder traversal :- (LRN)

The postorder traversal processes the root node after (post) the left and the right sub-tree. It starts by finding the far left leaf node and processing it. It then processes it's right sibling (if any). Finally processes the root node.

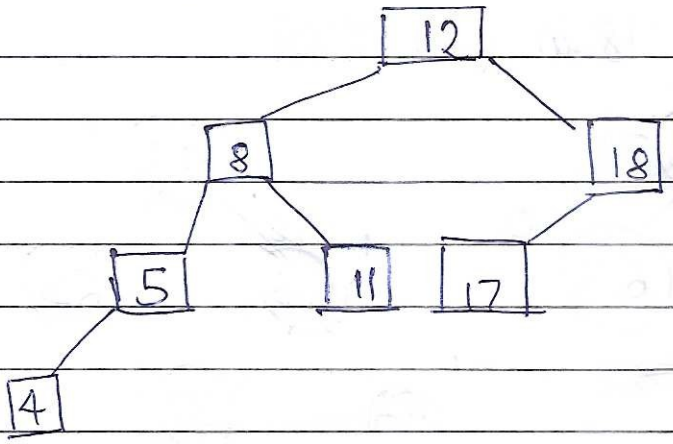
The postorder traversal of given tree is.

4, 5, 11, 8, 17, 18, 12

- Searching for a node in AVL tree

Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree. It starts by comparing the value to be searched to the root node. If the value is equal to the root node, required message is printed ; If the value to be searched is less than the root node then the function is recursively called on the left sub tree. If

the value to be searched is greater than root node then the function is recursively called on the ~~left~~ right sub tree.

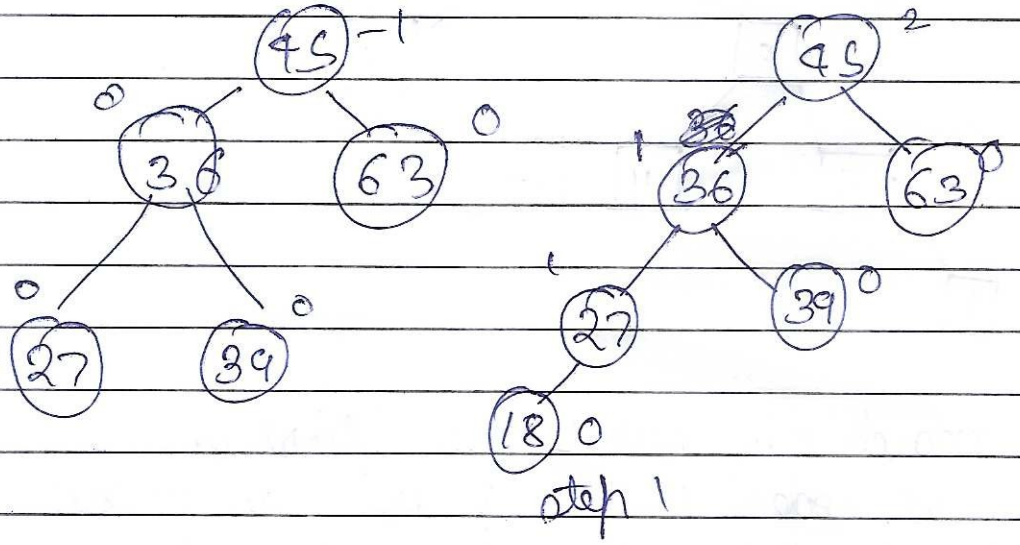


To search 11 in the AVL tree first compare it with root node 12. 11 is less than 12 so call the function on the left sub-tree. Now 11 is greater than 8, so call the function ~~function~~ on the right sub tree. Now 11 is equal to 11. So stop the function.

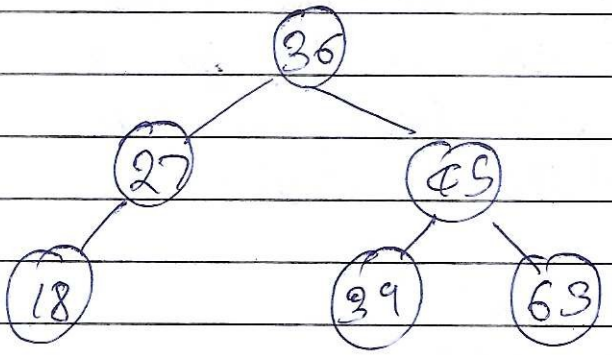
• Inserting a new node in an AVL tree.
Insertion in an AVL tree is also done in the same way as it is done in a binary ~~tree~~ search tree. In the AVL tree, the new node is always inserted at the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree. However if insertion of new node does not disturb the balance factor of the tree, then rotations are not required.

LL Rotation :- The new node is inserted at the left of the left sub tree of the critical node.

Eg Inserting 18 in given tree



step 1



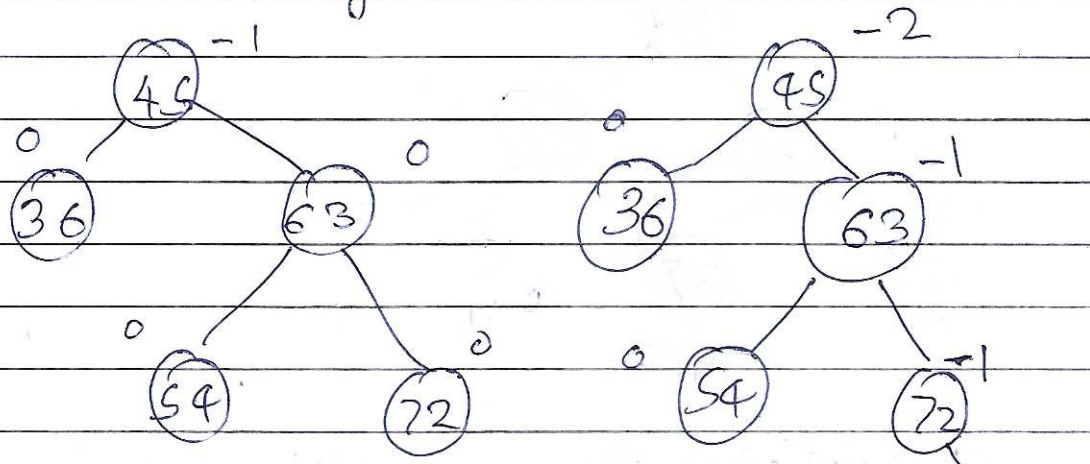
step 2.

RR Rotation - The new node is inserted in right sub tree of the right sub tree of the Critical node.

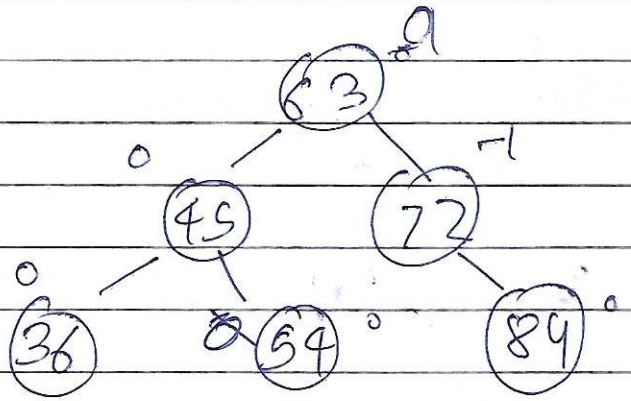
Yash Mahajan SE ITB 04

Page :
Date : / /

Eg Insert 89 in given tree

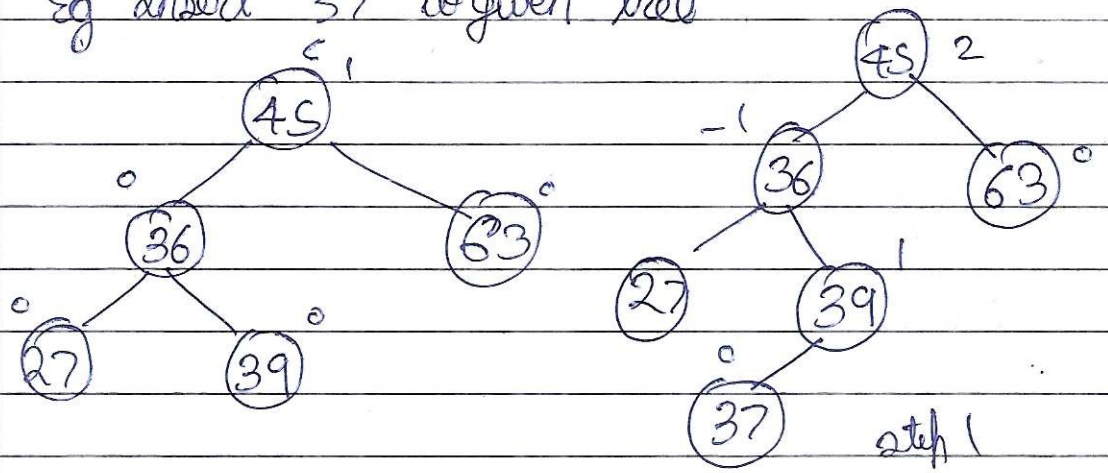


Step 1



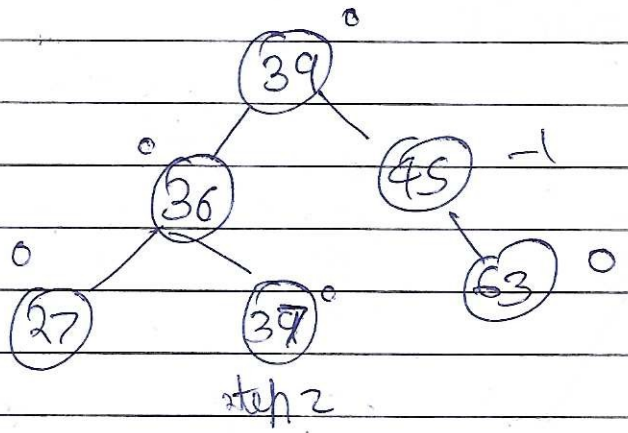
LR Rotation :- The new node is inserted in the right of the subtree of the left subtree of the critical node.

Eg Insert 37 to given tree



Step 1

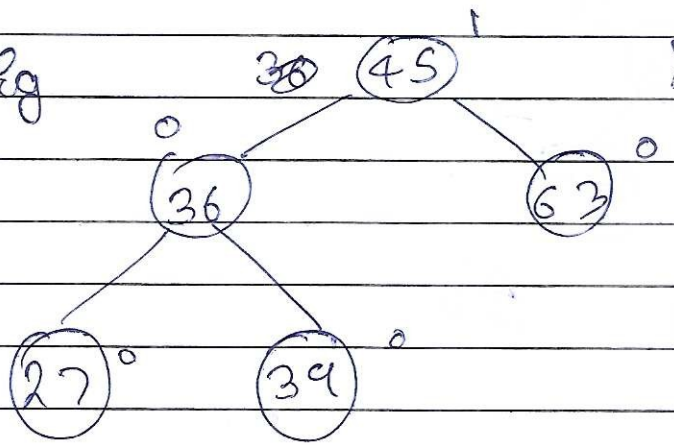
Page :
Date : 11



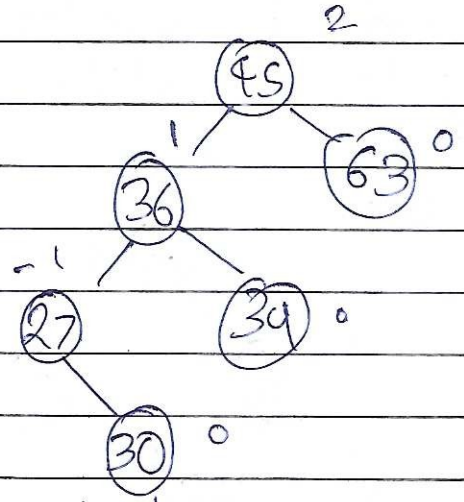
step 2

RR Rotation :- The new node is entered in the left sub-tree of the right subtree of the critical node.

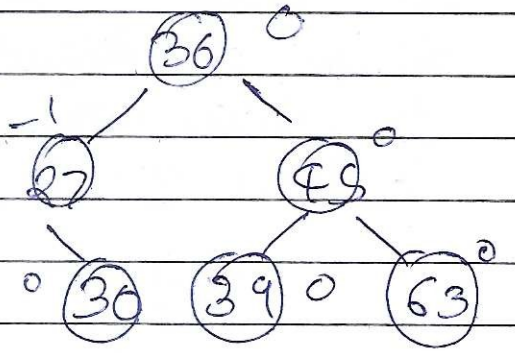
Eg



Insert 30



step 1



Yash Mahajan SE IT B 04

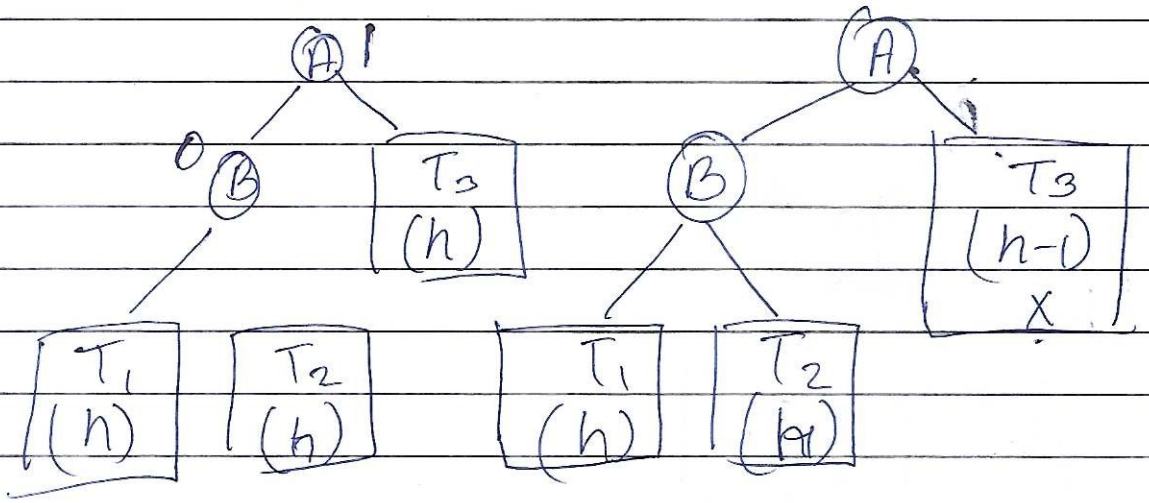
- Deletion operation in AVL trees.

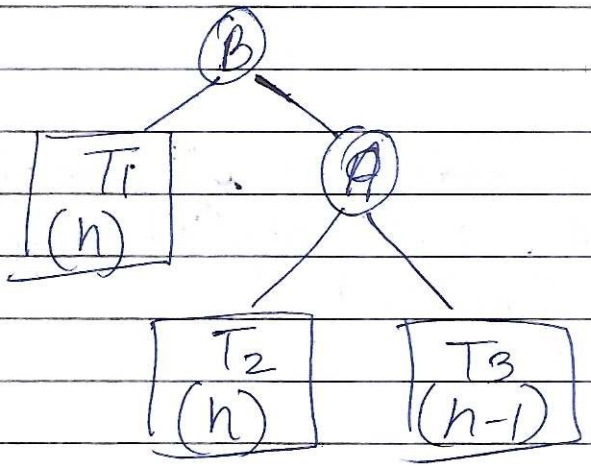
Deletion of a node is similar to that of binary search tree. But it goes one step ahead. Deletion may disturb the balance of AVL tree, so rotation is also performed.

~~RO~~ RO rotation.

~~The critical node R0 to move~~

Let B be the ~~next~~ node in the left sub-tree of A. RO rotation is applied when balance factor of B is 0.

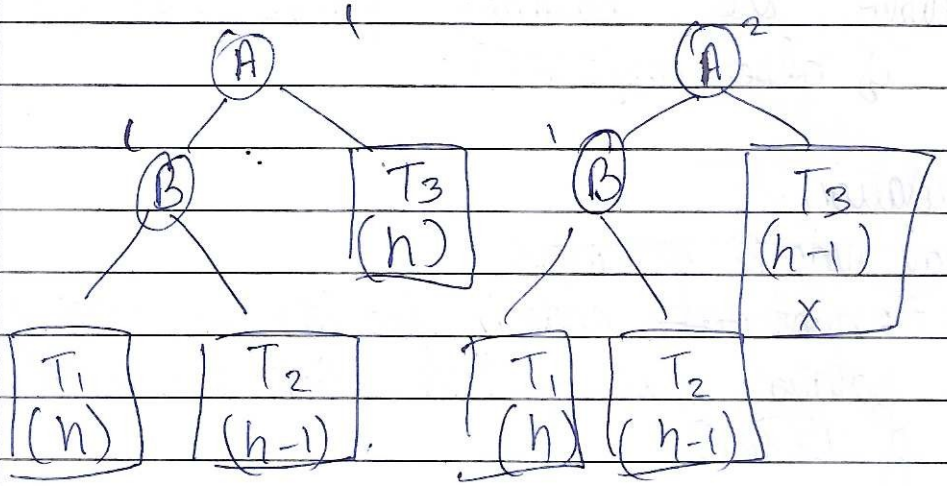


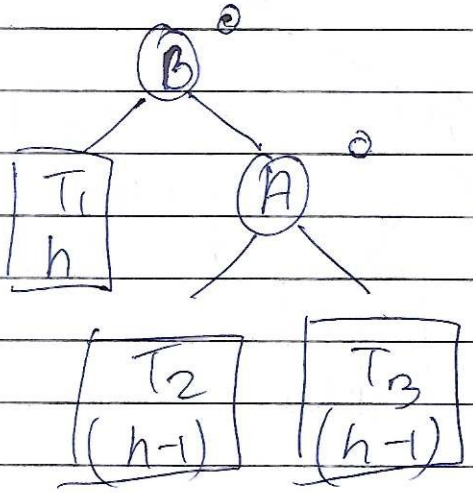


Ash Mahajan SE IT B 09

R1 Rotation -

Let B be the root node of the left sub-tree of A (critical node). R 1 rotation is applied when balance factor of B is 1



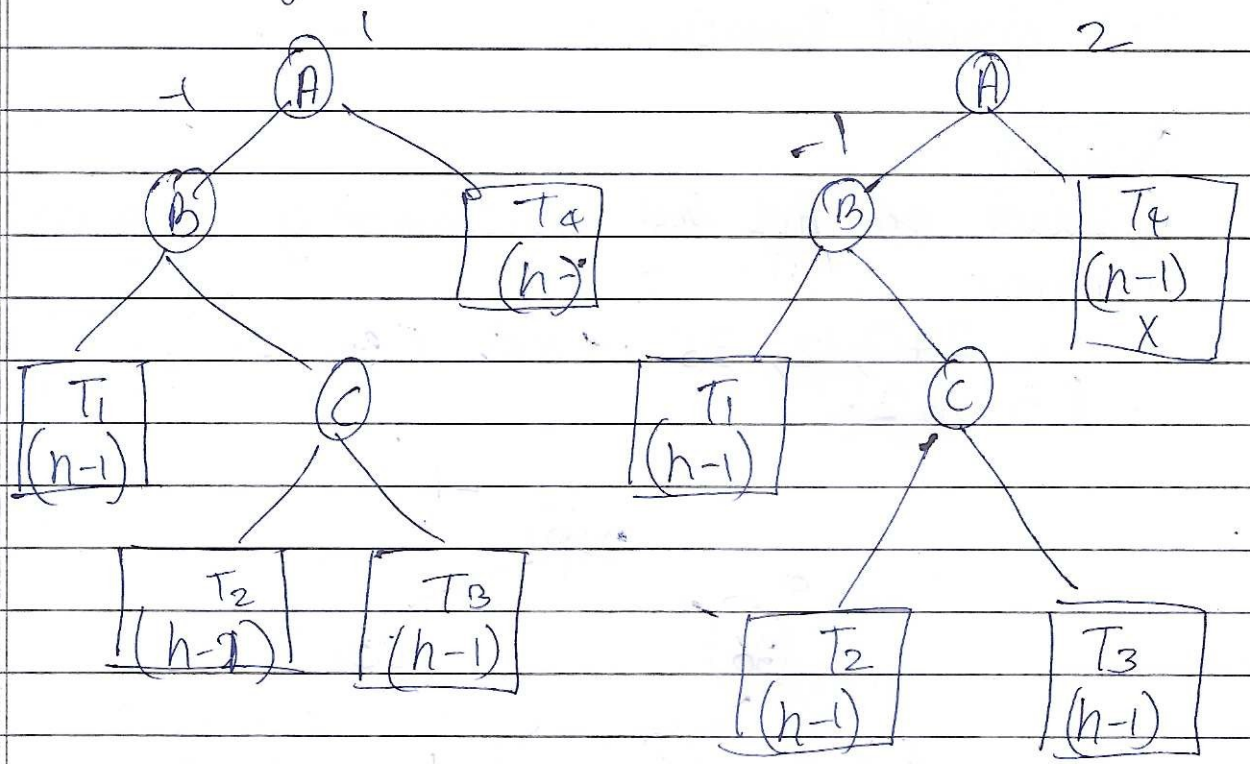


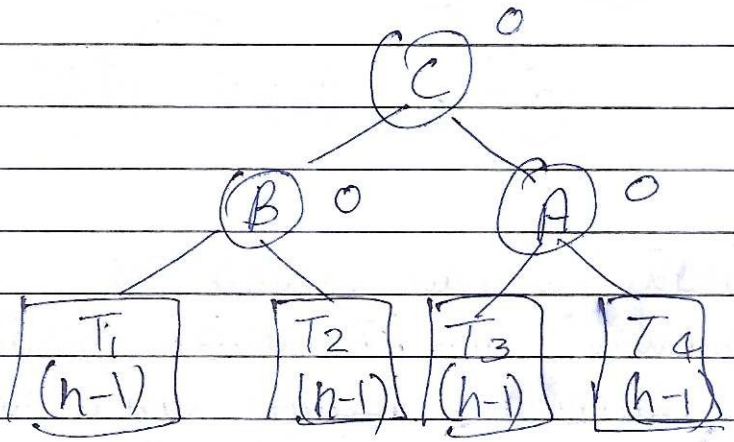
R-1 Rotation.

Let B be the root node of the left or right sub-tree of A. R-1 is applied only if balance factor of B is -1

Yash Mahajan SE IT B 04

Page :
Date : / /





Laboratory Exercise

A. Procedure

Write a C program to implement a AVL tree and show all the following operations in switch case,
i) Insertion
ii) Display Tree

```c
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
// An AVL tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};
// A utility function to get maximum of two integers
int max(int a, int b);
// A utility function to get the height of the tree
int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}
// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}
/* Helper function that allocates a new node with the given key and
NULL left and right pointers. */
struct Node* newNode(int key)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
```

```
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}
// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;
    // Perform rotation
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;
    // Return new root
    return x;
}
// A utility function to left rotate subtree rooted with x
// See the diagram given above.

struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;
    // Perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    // Return new root
    return y;
}
// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
```

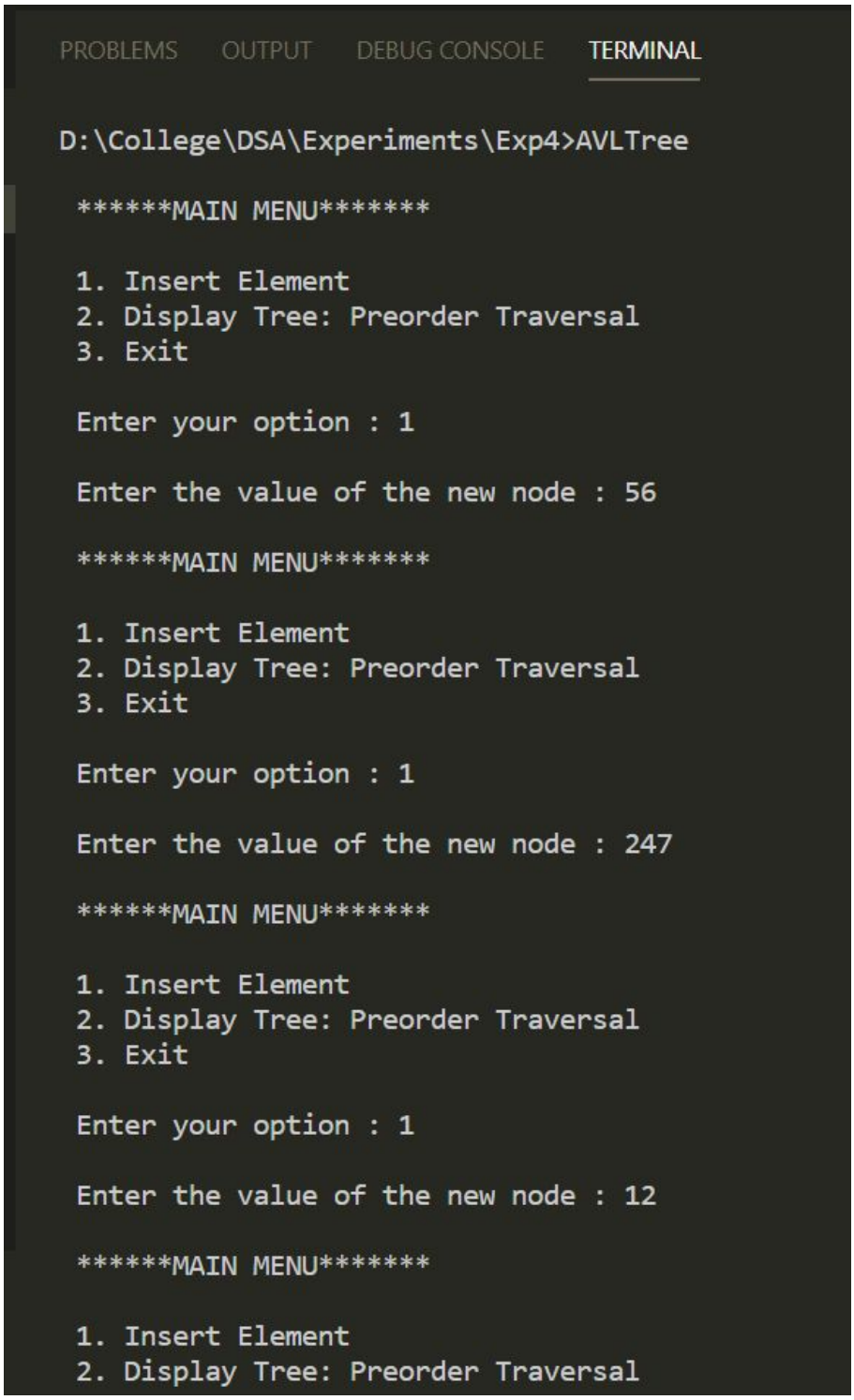
```
        return 0;
    return height(N->left) - height(N->right);
}
// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    else // Equal keys are not allowed in BST
        return node;
    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
    height(node->right));
    /* 3. Get the balance factor of this ancestor
    node to check whether this node became
    unbalanced */
    int balance = getBalance(node);
    // If this node becomes unbalanced, then
    // there are 4 cases
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if (balance < -1 && key < node->right->key)
```

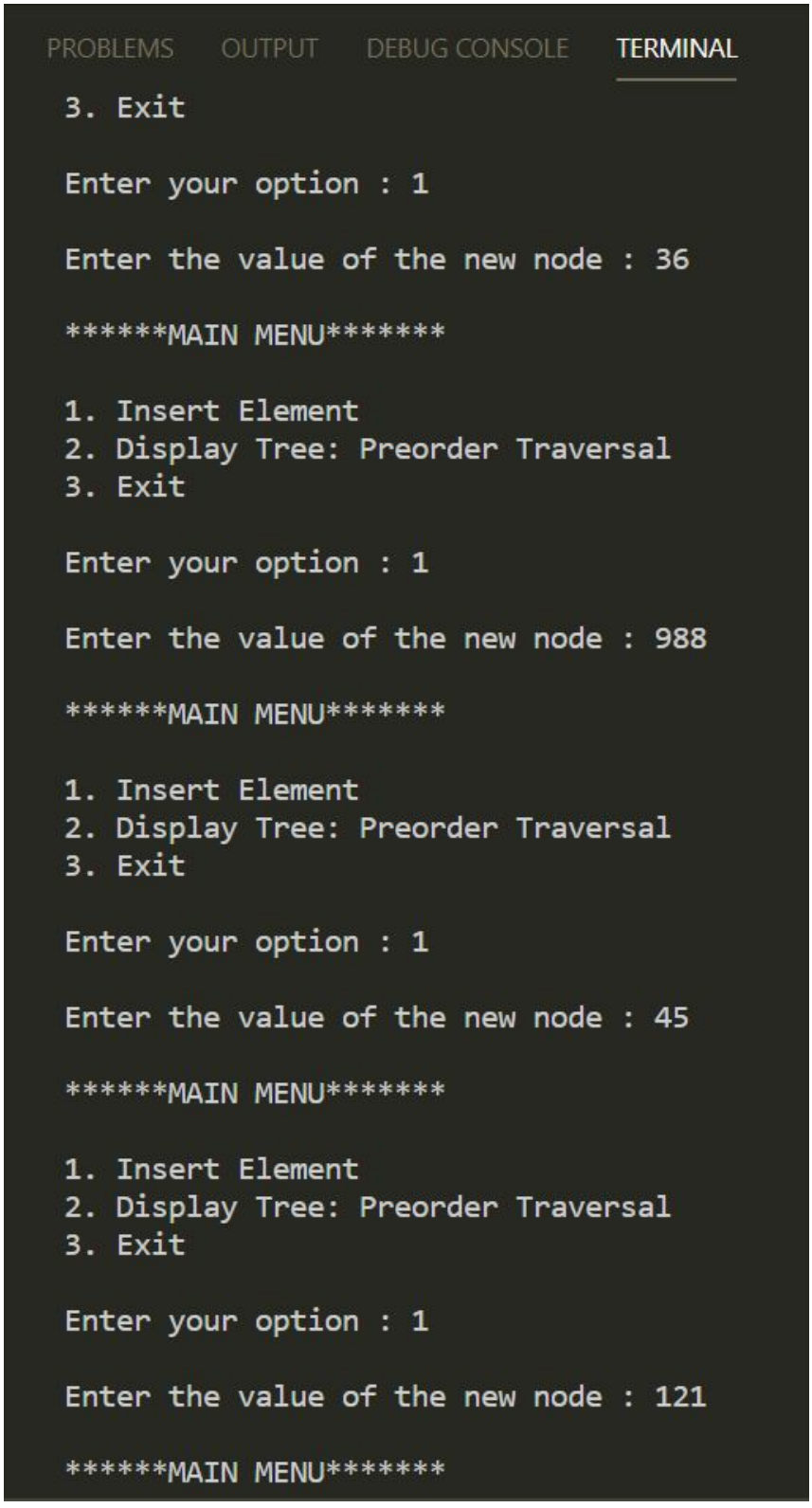
```
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}
// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
/* Driver program to test above function*/
int main()
{
    int option,val;
    struct Node *root = NULL;

    do
    {
        printf("\n ******MAIN MENU******* \n");
        printf("\n 1. Insert Element");
        printf("\n 2. Display Tree: Preorder Traversal");
        printf("\n 3. Exit ");
        printf("\n\n Enter your option : ");
        scanf("%d", &option);
        switch(option)
        {
            case 1:
                printf("\n Enter the value of the new node : ");
                scanf("%d", &val);
                root = insert(root, val);
```

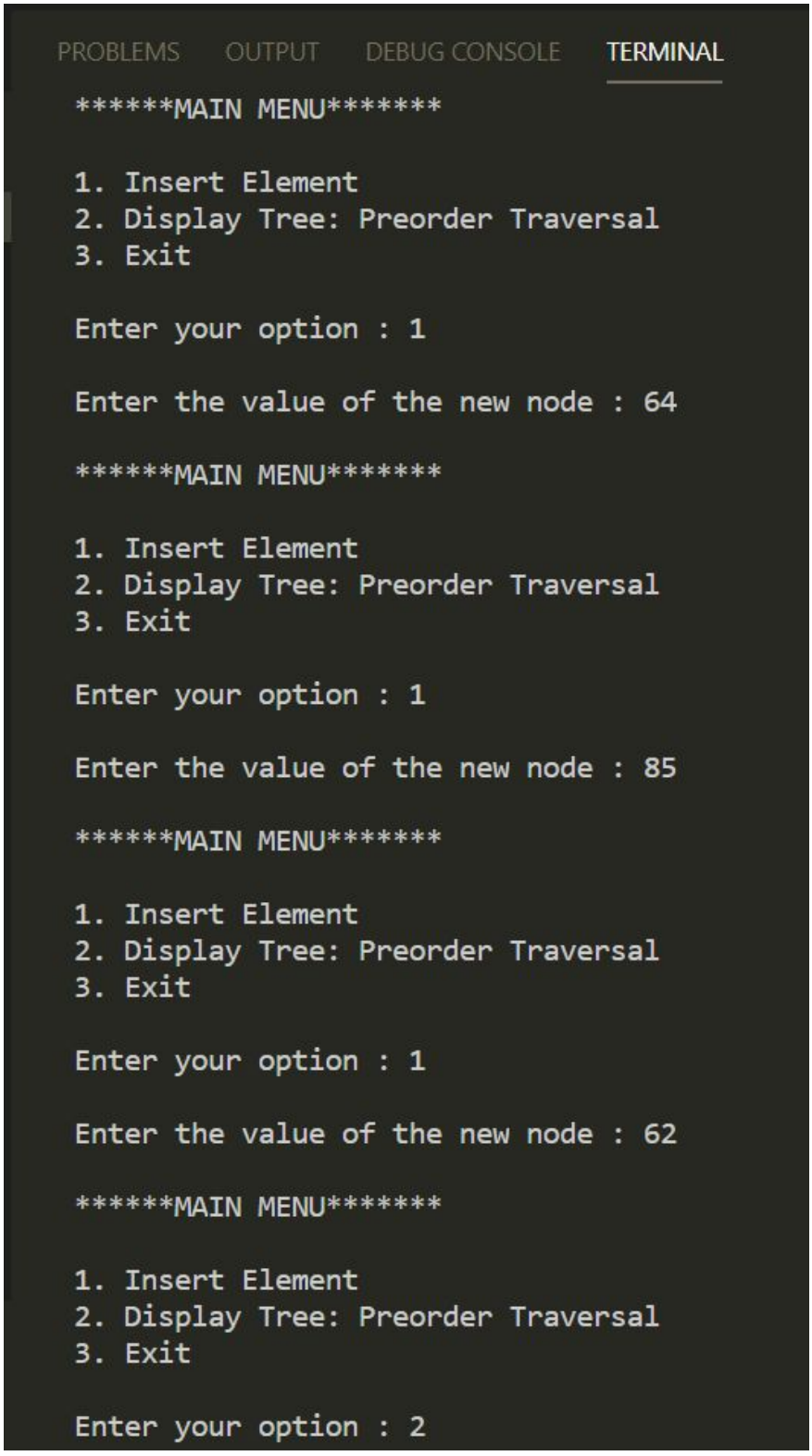
```
            break;

            case 2:
                printf("Preorder traversal of the constructed AVL tree is
\n");
                preOrder(root);
            break;
        }
    }while(option!=3);

    return 0;
}
```

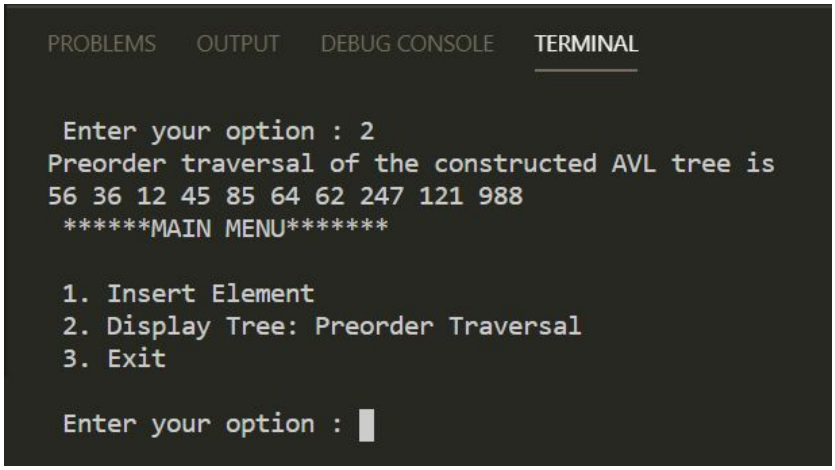
Result/Observation/Program code:

Observe the output for the above code and print it.




PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
3. Exit
Enter your option : 1
Enter the value of the new node : 36
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 1
Enter the value of the new node : 988
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 1
Enter the value of the new node : 45
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 1
Enter the value of the new node : 121
******MAIN MENU*******


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 1
Enter the value of the new node : 64
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 1
Enter the value of the new node : 85
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 1
Enter the value of the new node : 62
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option : 2


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Enter your option : 2
Preorder traversal of the constructed AVL tree is
56 36 12 45 85 64 62 247 121 988
******MAIN MENU*******
1. Insert Element
2. Display Tree: Preorder Traversal
3. Exit
Enter your option :

Yash Mahajan SE IT B 04.

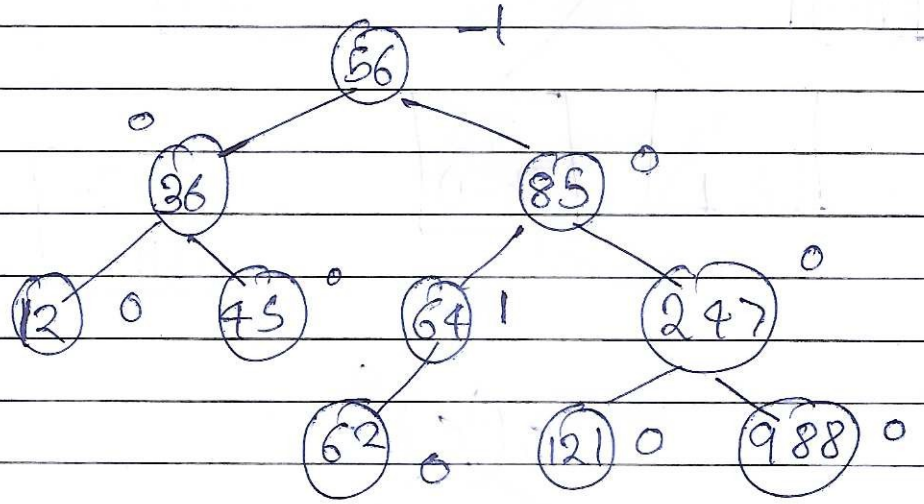
Page:
Date: / /

Post Experiment Exercise :-

Question :-

Construct an AVL tree corresponding to given numbers.

56, 247, 12, 36, 988, 45, 121, 64, 85, 62.



Conclusion :-

In this experiment we have studied the AVL Tree data structure and its various operations like traversal, searching, insertion and deletion. We have performed programs to implement AVL trees and also perform traversal and insertion operation.

In most BST operations take O(h) time where h is height. It can also become O(n) is the tree is skewed. AVL trees balance themselves after every operation. Hence their height remains Logn. Thus operations in AVL tree take O(Logn) time.