**National University of Computer and Emerging Sciences**



**Laboratory Manual**

*for*

# Data Structures Lab

|  |  |
| --- | --- |
| Course Instructor | Mr. Uzair Naqvi |
| Lab Instructor | Mr. Durraiz Waseem |
| Semester | Fall 2024 |

# Department of Computer Science

FAST-NU, Lahore, Pakistan

**Objectives:**

In this lab, students will practice BST and AVL

**Problem 1: Binary Tree and BST Implementation:**

Implement the following Tree Node:

struct Node

{ int data;

Node\*left;

Node \*right;

};

class BST

{

Node\* root;

};

**Implement the following member functions for BST:**

1. A default **Constructor** that sets the root to nullptr.
2. Implement a function ‘**insert’**. It should insert the data while considering the insertion rules. If the data already exists in the BST, simply return false and true otherwise. bool insert(int v)
3. A **copy constructor** which uses recursion to deep copy another Binary Search Tree object.
4. **Print** the tree on the screen
5. A function “inorderPrint” prints the keys using in-order traversal. void **inorderPrint** () const
6. Use level order traversal for the printing of trees, level by level. void **levelorderPrint** () const
7. A function “search”. The function then uses recursion to return a pointer to the corresponding node. If the key does not exist, the function returns nullptr. Node\* **search**(int key)
8. Use **Postorder** LRV to implement the Destructor for BST.
9. **BST to Doubly Linked List**: Convert a BST to a doubly linked list (DLL) in-place. The left and right pointers of nodes in the BST should act as the prev and next pointers of the DLL, respectively. **Hint**: Use in-order traversal and keep track of the previous node.
10. **Count Pairs from Two BSTs that Add Up to a Target**: Given two BSTs, find the number of pairs (one from each tree) where the sum of the pair equals a given target. **Hint**: Traverse one BST and for each value, search for the complement in the other BST efficiently.
11. **Kth Largest Element in a BST (Without Extra Memory)**: Find the K-th largest element in a BST without using any additional memory (no arrays or lists allowed). **Hint**: Use reverse in-order traversal and keep a counter until the k-th largest element is reached.

**Problem 2: AVL Implementation:**

class AVL; class Node { friend class AVL; private:

int key; // data to be stored in a node

int height; // To store the height of each node with it

Node\* left;

Node\* right;

}; class AVL { private: Node\* root; int avlHeight (Node\*); int max (int a, int b);

int getBalance(Node\*); // get balance factor of a node

Node\* SingleRightRotation(Node\*); //LL case

Node\* SingleLeftRotation(Node\*); //RR case

Node\*LeftRightRotation(Node\*); //LR case

Node\*RightLeftRotation(Node\*); //RL case

Node\* insert (int data, Node\*); Node\* delete (int data, Node\*); void display\_inorder(Node\*);

public:

// driver functions for insert, delete, display

};

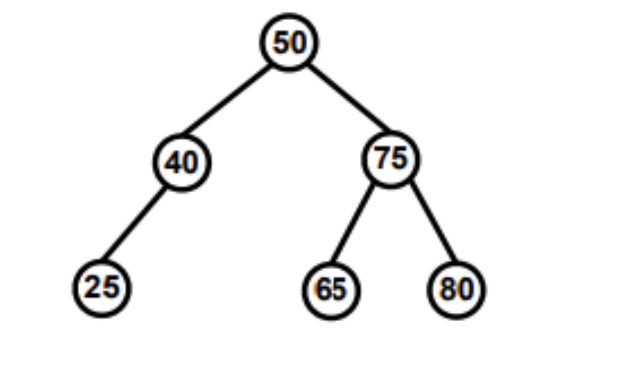
**Implement a template-based AVL tree and create the functions described below:**

1. Create a function ‘**insert()’** to insert a node into AVL keeping in view that the tree remains balanced after insertion. For this task, you need to compute the **Balancing Factor (BF)** of each node.
   * **Node \*insert(Node \*node, int key)**
   * **Int balanceFactor(Node \*n)**
2. Implement deletion of a given node from the AVL Tree. This function should delete the node with the key from the AVL tree rooted at node **‘n’**. Think about its logic now and don’t forget to balance your tree afterwards just like you did in insertion.
   * **Node\* delete (int data, Node\* n);**
3. Implement functions for four types of rotation namely **RR, LL, LR,** and **RL.**
   * **Node \*RotateRight(Node \*y)**
   * **Node \*RotateLeft(Node \*x)**
4. Create a bool function **‘isAVL()’** which takes a tree as an argument and tells whether the tree is an AVL or not.
5. A **‘search’** function to search an element in the tree **Bool search(int key)**
6. A **recursive** function ‘**findmin()’** that finds the minimum element using recursion.
7. A **‘print’** function to print the **inorder, preorder,** and **postorder** traversal.
8. **Print all possible paths from root to leaf.** Implement a public member function of the

AVLTree class which prints all the root-to-leaf paths of a given AVLTree. The prototype of your function can be

**● void printAllPaths ()**

For example, if we call this function on the following Tree:

For example, if we call this function on the following Tree:

It should print the following paths (in this order):

**50 -> 40 -> 25**

**50 -> 75 -> 65**

**50 -> 75 -> 80**

**Hint 1:** You will need to implement a private recursive helper function.

**● void printAllPathsRecursive(Node\*curr, int path[], int pathLength)**

**Hint 2**: You can assume that the maximum length of a root-to-leaf path will be 100.

**Note: Do NOT use any global or static variables in your implementation.**