**Lab 4A**

# Binary Search Tree

In this lab session, we will learn and implement a special data structure: **Binary Search Tree**. Source code needs to be distributed in a single file named `StudentID_1.cpp`.

Consider that each node in the Binary Search Tree has the following basic structure:

```
struct Node
{
    int key;
    Node* left;
    Node* right;
};
```

## 1 Exercise 1

Implement these basic functions and operations on the Binary Search Tree as follows:

1. Create a new Node from a given value.

   `Node* newNode(int data)`

2. Insert a new value into a Binary Search Tree.

   `Node* insert(Node* root, int data)`

3. Search a Node with a given value from a Binary Search Tree. Return `NULL` if not found.

   `Node* search(Node* root, int data)`

4. Delete a Node with a given value from a Binary Search Tree.

   `Node* delete(Node* root, int data)`

5. Traversal in **Pre-order**, **In-order**, **Post-order** and **Level-order**.

   `void NLR(Node* root)`

   `void LNR(Node* root)`

   `void LRN(Node* root)`

   `void LevelOrder(Node* root)`

## 2 Exercise 2

With the Binary Search Tree implemented in Exercise 1, solve the following problems:

1. Calculate the height of a given Binary Search Tree.

   `int height(Node* root)`

2. Count the number of Node from a given Binary Search Tree.

   `int countNode(Node* root)`

3. Count the number of internal nodes (non-leaf nodes) from a given Binary Search Tree.

   `int countInternal(Node* root)`

4. Calculate the total value of all Nodes from a given Binary Search Tree:

   `int sumNode(Node* root)`

5. Count the number leaves from a given Binary Search Tree.

   `int countLeaf(Node* root)`

6. Count the number of Nodes from a given Binary Search Tree which key value is less than a given value.

   `int countLess(Node* root, int x)`

7. Count the number of Nodes from a given Binary Search Tree which key value is greater than a given value.

   `int countGreater(Node* root, int x)`

Test example in main function:

- Initialize an empty BST Tree.
- Insert 8, 6, 5, 7, 10, 9.
- Show the tree in Pre-order, In-order, Post-order.
- Remove 8. Show the tree in Level Order.
- Solve the problems in Exercise 2. Test with x = 7 (if the problem requires a key).

**Lab 4B**

# AVL Tree

In this lab session, we will learn and implement a self-balancing binary search tree: **AVL Tree**. Source code needs to be distributed in a single file named `StudentID_2.cpp`.

## 3 Exercise 1

Consider that each node in the AVL Tree has the following basic structure:

```
struct Node
{
    int key;
    Node* left;
    Node* right;
    int height;
};
```

Please implement these basic functions and operations on the AVL Tree as follows:

1. Create a new Node from a given value.

   `Node* newNode(int data)`

2. Insert a new value into a AVL Tree (do not add existing keys).

   `Node* insertNode(Node* root, int data)`

3. Search a Node with a given value from a AVL Tree. Return `NULL` if not found.

   `Node* searchNode(Node* root, int data)`

4. Delete a Node with a given value from a AVL Tree.

   `Node* deleteNode(Node* root, int data)`

5. Traversal in **Pre-order**, **In-order**, **Post-order** and **Level-order**.

   ```
   void NLR(Node* root)
   void LNR(Node* root)
   void LRN(Node* root)
   void LevelOrder(Node* root)
   ```

## 4 Exercise 2

With the AVL Tree implemented in Exercise 1, solve the following problems:

1. Check if the given AVL Tree is a full tree.

   `bool isFull(Node* root)`

2. Check if the given AVL Tree is a complete tree.

   `bool isComplete(Node* root)`

3. Check if the given AVL Tree is a perfect tree.

   `bool isPerfect(Node* root)`

4. Find the least common ancestor for any two given nodes in AVL Tree.

   `int findCommonAncestor(Node* root, int x, int y)`

5. Find all nodes with 2 child nodes, and the left child is a divisor of the right child. Print them to the console in ascending order.

   `void printSpecialNodes(Node* root)`

## Submission

Your source code must be contributed in the form of a compressed file and named your submission according to the format `StudentID.zip`. Here is a detail of the directory organization:

```
StudentID
├── StudentID_1.cpp
└── StudentID_2.cpp
```

The end.