**Binary Trees**

**3A. Binary Search Tree:** Implement a program to construct a Binary Search Tree and perform insert, delete, and search operations. [This is implemented using a linked list]

**Algorithm:**

Step 1: Create class Node to store the data and its children.

Step 2: To insert a node N in the tree, if the root is NULL, initialize the root with N, else recursively try to insert the node N to its left child if its less than the root, or the right child if it is greater than the root.

Step 3: To remove a node N in the tree, traverse the tree (search left child if lesser, right child if greater). When the node is found, if it does not have any children, simply remove the node. Else, find the minimum element in the right subtree and put that minimum node in place of the current node. Then delete the repeated minimum node in the right subtree.

Step 4: To search for a node N, recursively traverse the tree – explore right subtree if N is greater than current node, else the left subtree if N is less than the current node. When then node is found, return the node. If a null pointer is reached, return NULL.

**Program:** [In next page]







    BST() {

        // default constructor

        root = NULL;

    }

    void insert(int x) {

        // insert into tree

        root = insert(x, root);

    }

    void remove(int x) {

        // remove fromm tree

        root = remove(x, root);

    }

    void display() {

        // display tree inorder

        inorder(root);

        cout << endl;

    }

    void search(int x) {

        // search from tree

        node\* result = find(root, x);

        if (result == NULL) {

            cout << "Not found!\n";

        } else {

            cout << "Found\n";

        }

    }

};

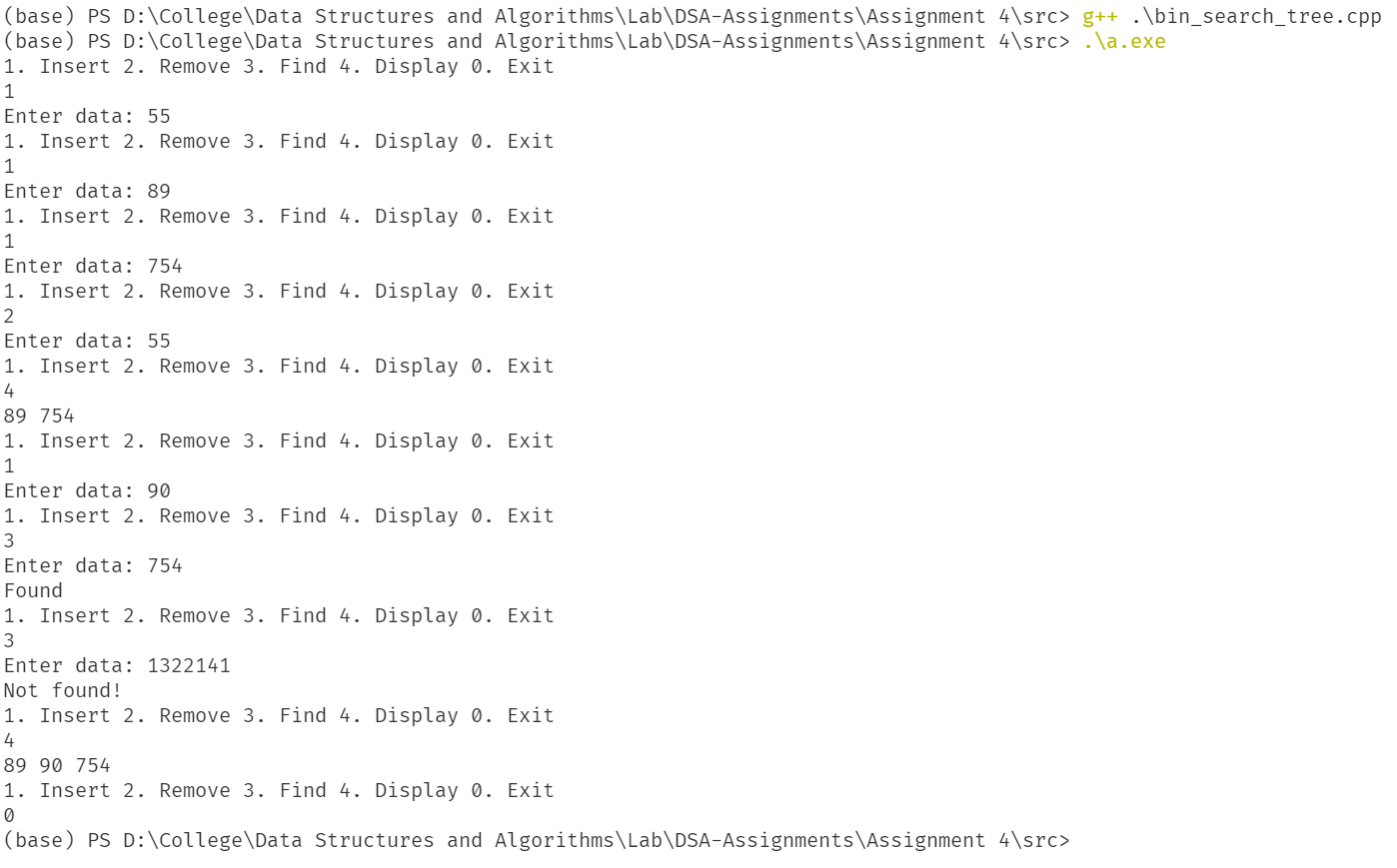
void menu() {

    cout << "1. Insert 2. Remove 3. Find 4. Display 0. Exit" << endl;

}



**Output:**

****

**Results:**

Thus, the binary search tree with insert, delete, find and display operations has been implemented.

**4B. Min and Max in BST:** Find the minimum and maximum elements in a binary search Tree

**Algorithm:**

Step 1: Initialize the Binary Search Tree as explained in the algorithm for question 1.

Step 2: To find the minimum node, recursively traverse to the left subtree. When a null pointer is found, its parent will be the minimum element

Step 3: To find the maximum node, recursively traverse to the right subtree. When a null pointer is found, its parent will be the maximum element.

**Program:** [In next page]

****

****

****

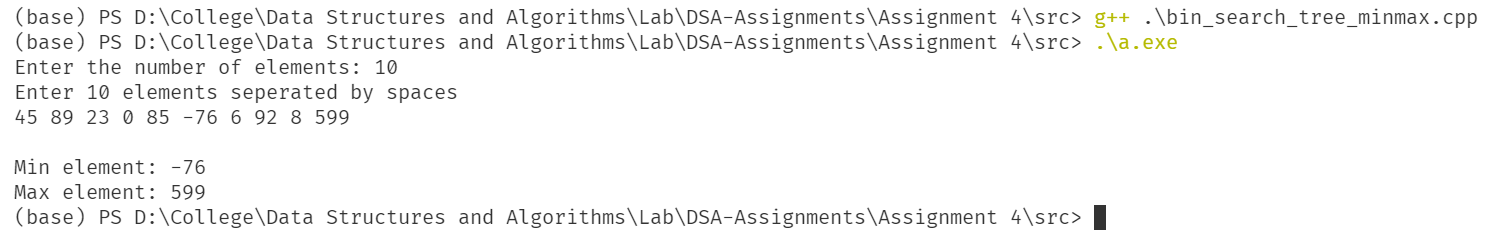
****

****

[Continued in next page]

****

**Output:**



**Results:**

Thus, the program to find minimum and maximum elements in a binary tree is implemented.

**[Note: Since questions C and D were very similar, I have merged them into a single program]**

**4C | D. Expression Tree:** Implement a program to create an expression tree given an infix expression and get the prefix **and** postfix form of the expression using **pre-order** and **post-order** traversal respectively.

**Algorithm:**

Step 1: Input the infix expression.

Step 2: Initialize two stacks – one for nodes and one for characters.

Step 3: Initialize three temp nodes t, t1 and t2.

Step 4: Traverse through the infix expression.

Step 5: If a left parenthesis is found, push it to the character stack.

Step 6: Else if an operand has been found, create a new node with the operand and push it to the nodes stack.

Step 7: Else if the standard operators are found, while the stack is not empty **and** the stack top is not “(“, create a new node t from the top of the characters stack. Then pop two nodes from the nodes stack, t1 and t2. Set the left child of t to be t2, and the right child to be t1. Push t to the nodes stack.

Step 8: Else if a “)” is found, while the stack is not empty and the stack top is not “(“, pop two nodes from the nodes stack, t1 and t2. Set the left child of t to be t2, and the right child to be t1. Push t to the nodes stack.

Step 9: Go to step 4.

Step 10: Pop the only remaining node from the top of the nodes stack. This will be the root of the expression tree.

**Pre-order traversal:**

Step 1: If the current node is not NULL, print the current nodes data.

Step 2: Go to step 1 for the left child of the current node, then come back and then for the right child of the current node.

**Post-order traversal:**

Step 1: If the current node is not NULL, print the current left child’s data recursively then the right child’s data.

Step 2: Then print the current nodes data.

**In-order traversal:**

Step 1: If the current node is not NULL, print the left child’s data recursively.

Step 2: Print the current nodes data, then recursively print the right child’s data.

**Program:** [In next page]



[Continued in next page]

****

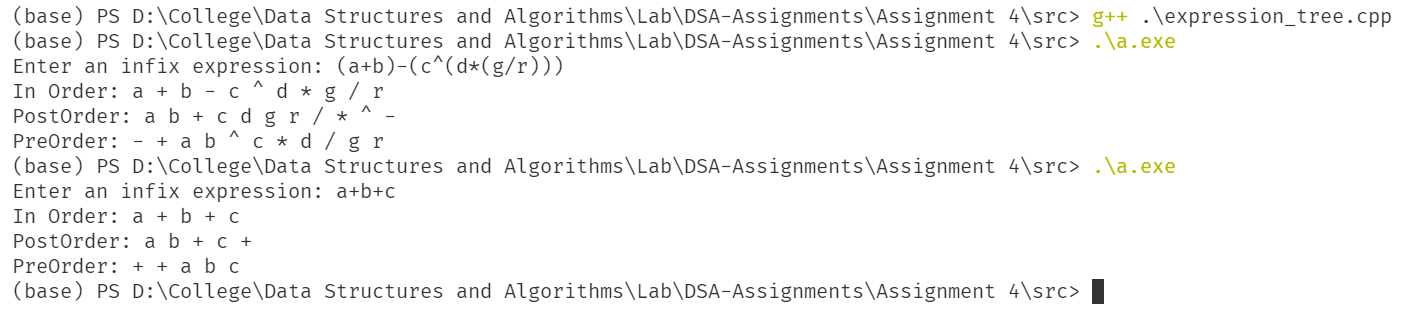
****

****

****

****

**Output:**

****

**Results:**

Thus, the program to build and traverse an expression tree from a given infix expression is implemented.