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]

```cpp
// Exp4A: Insert, delete and search in binary trees
// Author: Pranjal Timsina
#include<iostream>
using namespace std;

// Create class BST for storing data
class BST {
public:
    // node to store individial elements
    struct node {
        int data;
        node* left;
        node* right;
    };
    // initialize the root of the BST
    node* root;

    // function to insert into the BST recursively
    node* insert(int x, node* t) {

        if(t == NULL) {

            // check if root
            t = new node;
            t->data = x;
            t->left = t->right = NULL;

        } else if(x < t->data)

            // insert in the left subtree if smaller
            t->left = insert(x, t->left);

        else if(x > t->data)

            // insert in the right subtree if greater
            t->right = insert(x, t->right);

        return t;
    }
```

```
node* findMin(node* t) {
    // find the minimum element recursively
    if(t == NULL)
        return NULL;
    else if(t->left == NULL)
        // if leftmost leaf then return the node
        return t;
    else

        // continue recursion
        return findMin(t->left);
}

node* findMax(node* t) {
    // find the maximum element recursively
    if(t == NULL)
        return NULL;
    else if(t->right == NULL)
        // if right mostleaf then return the node
        return t;
    else
        // continue recursion
        return findMax(t->right);
}

node* remove(int x, node* t) {
    // search and remove a node with data x

    node* temp;
    if(t == NULL)

        // if empty tree, return null
        return NULL;
    else if(x < t->data)

        // if x is less than data in current node
        // find and remove from left sub tree
        t->left = remove(x, t->left);
```

```
    else if(x > t->data)
            // if x is more than data in current node
            // find and remove from right sub tree
            t->right = remove(x, t->right);
        else if(t->left && t->right) {
            // rearrage the tree
            temp = findMin(t->right);
            t->data = temp->data;
            t->right = remove(t->data, t->right);
        } else {
            // rearrage the tree
            temp = t;
            if(t->left == NULL)
                t = t->right;
            else if(t->right == NULL)
                t = t->left;
            delete temp;
        }
        return t;
    }
    void inorder(node* t) { // inorder traversal of the tree
        if(t == NULL)
            return;
        // recurse for left node
        inorder(t->left);
        cout << t->data << " ";
        // recurse for right node
        inorder(t->right);
    }
    node* find(node* t, int x) { // recursively find node
        if(t == NULL)
            return NULL;
        else if(x < t->data)
            return find(t->left, x);
        else if(x > t->data)
            return find(t->right, x);
        else return t;
    }
```

```
    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;
}
```

```
int main() {
    BST t;
    int choice;
    while (true) {
        // main driver code for getting user input
        // and handling different choices
        menu();
        cin >> choice;

        if (!choice) {
            return 0; // exit
        } else if (choice == 4) {
            // display tree (inorder traversal)
            t.display();
            continue;
        }
        else {
            int query;
            cout << "Enter data: ";
            cin >> query;
            switch (choice) {
                case 1:
                    t.insert(query);
                    break;
                case 2:
                    t.remove(query);
                    break;
                case 3:
                    t.search(query);
                    break;
            }
        }
    }
    return 0;
}
```

Output:

```
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> g++ .\bin_search_tree.cpp
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> .\a.exe
1. Insert 2. Remove 3. Find 4. Display 0. Exit
1
Enter data: 55
1. Insert 2. Remove 3. Find 4. Display 0. Exit
1
Enter data: 89
1. Insert 2. Remove 3. Find 4. Display 0. Exit
1
Enter data: 754
1. Insert 2. Remove 3. Find 4. Display 0. Exit
2
Enter data: 55
1. Insert 2. Remove 3. Find 4. Display 0. Exit
4
89 754
1. Insert 2. Remove 3. Find 4. Display 0. Exit
1
Enter data: 90
1. Insert 2. Remove 3. Find 4. Display 0. Exit
3
Enter data: 754
Found
1. Insert 2. Remove 3. Find 4. Display 0. Exit
3
Enter data: 1322141
Not found!
1. Insert 2. Remove 3. Find 4. Display 0. Exit
4
89 90 754
1. Insert 2. Remove 3. Find 4. Display 0. Exit
0
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src>
```

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]

```cpp
// Exp4A: Insert, delete and search in binary trees
// Author: Pranjal Timsina
#include<iostream>

using namespace std;

// Create class BST for storing data
class BST {
public:
    // node to store individial elements
    struct node {
        int data;
        node* left;
        node* right;
    };

    // initialize the root of the BST
    node* root;

    // recursively remove a node
    node* makeEmpty(node* t) {
        if(t == NULL)
            return NULL;
        {
            makeEmpty(t->left);
            makeEmpty(t->right);
            delete t;
        }
        return NULL;
    }

    // function to insert into the BST recursively
    node* insert(int x, node* t) {

        if(t == NULL) {

            // check if root
            t = new node;
            t->data = x;
```

```
            t->left = t->right = NULL;

        } else if(x < t->data)

            // insert in the left subtree if smaller
            t->left = insert(x, t->left);

        else if(x > t->data)

            // insert in the right subtree if greater
            t->right = insert(x, t->right);

        return t;
    }

    node* findMin(node* t) {
        // find the minimum element recursively
        if(t == NULL)
            return NULL;
        else if(t->left == NULL)
            // if leftmost leaf then return the node
            return t;
        else

            // continue recursion
            return findMin(t->left);
    }

    node* findMax(node* t) {
        // find the maximum element recursively
        if(t == NULL)
            return NULL;
        else if(t->right == NULL)
            // if right mostleaf then return the node
            return t;
        else
            // continue recursion
            return findMax(t->right);
    }
```

```
node* remove(int x, node* t) {
    // search and remove a node with data x

    node* temp;
    if(t == NULL)

        // if empty tree, return null
        return NULL;
    else if(x < t->data)

        // if x is less than data in current node
        // find and remove from left sub tree
        t->left = remove(x, t->left);

    else if(x > t->data)

        // if x is more than data in current node
        // find and remove from right sub tree
        t->right = remove(x, t->right);
    else if(t->left && t->right) {
        // rearrage the tree
        temp = findMin(t->right);
        t->data = temp->data;
        t->right = remove(t->data, t->right);
    } else {

        // rearrage the tree
        temp = t;
        if(t->left == NULL)
            t = t->right;
        else if(t->right == NULL)
            t = t->left;
        delete temp;
    }

    return t;
}
```

```
void inorder(node* t) {
        // inorder traversal of the tree
        if(t == NULL)
            return;
        // recurse for left node
        inorder(t->left);
        cout << t->data << " ";
        // recurse for right node
        inorder(t->right);
    }

    node* find(node* t, int x) {
        // recursively find node with int x
        if(t == NULL)
            return NULL;
        else if(x < t->data)
            return find(t->left, x);
        else if(x > t->data)
            return find(t->right, x);
        else
            return t;
    }

    BST() {
        // default constructor
        root = NULL;
    }

    ~BST() {
        // destructor
        root = makeEmpty(root);
    }

    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;
}
```

[Continued in next page]

```
int main() {
    BST t;

    int elements;
    cout << "Enter the number of elements: ";
    cin >> elements;

    cout << "Enter " << elements << " elements seperated by spa
ces\n";

    int a;

    for (int i = 0; i < elements; i++) {
        cin >> a;
        t.insert(a);
    }


    cout << "\nMin element: " << t.findMin(t.root)->data;
    cout << "\nMax element: " << t.findMax(t.root)->data;
    return 0;
}
```

Output:

```
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> g++ .\bin_search_tree_minmax.cpp
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> .\a.exe
Enter the number of elements: 10
Enter 10 elements seperated by spaces
45 89 23 0 85 -76 6 92 8 599

Min element: -76
Max element: 599
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src>
```

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]

```cpp
// Exp 4CD: Expression trees traversal
// Author: Pranjal Timsina; 20BDS0392

#include <iostream>
#include <string>
#include <vector>
#include <stack>

#define MAX 500

// Initializing a stack
std::string stack[MAX];
int top = -1;

// functions to check if stack is empty or full
bool is_empty() { return (top == -1); }
bool is_full() { return (top == MAX-1); }

// returns the top element of stack without removing
std::string peek() {
    if (is_empty()) { throw "Stack underflow!";}
    return stack[top];
}

// pops the top most element of the stack and decrements top
std::string pop() {
    if (is_empty()) { throw "Stack underflow!";}
    return stack[top--];
}

// pushes data to the stack and increments top
void push(std::string data) {
    if (is_full()) {throw "Stack Overflow!";}
    stack[++top] = data;
}
```

[Continued in next page]

```
// A class to store data
class node {
    public:
    std::string data;
    node* left;
    node* right;
};

// a factory: to create nodes with a data
node* node_factory(std::string data) {
    node* new_node = new node;
    new_node->left = NULL;
    new_node->right = NULL;
    new_node->data = data;
    return new_node;
}

// left parenthesis returns -1
// right parenthesis returns 1
// else returns 0
int parentheses(std::string a) {
    std::string left[3] = {"[", "{", "("};
    std::string right[3] = {"]", "}", ")"};

    for (std::string ob: left) {
        if (a == ob) {
            return -1;
        }
    }

    for (std::string ob: right) {
        if (a == ob) {
            return 1;
        }
    }
    return 0;
}
```

```
// returns true operator is passed
bool isoperator(std::string character) {
    // just a simple function to check whether character is
    // an operator or not
    std::string operators[6] = {"+", "-
", "*", "\\", "/", "^" };

    for (auto op: operators) {
        if (character == op)
            return true;
    }

    return false;
}

// returns the precedence of an operator
int operator_precedence(std::string op) {
    if (!isoperator(op)) {
        return 0;
    }
    if (op == "+" || op == "-") {
        return 1;
    } else if (op == "*" || op ==" /" || op == "%") {
        return 2;
    } else
        return 3;
}

// takes a single string and returns a vector of
// tokens ie. a+b -> [a, +, b]
std::vector<std::string> tokenize(std::string infix) {
    // to tokenized string, which is initially empty
    std::vector<std::string> tokenized;
    tokenized.push_back("(");
    std::string previous = "";
```

```
for (char &c: infix) {

        // hacky way to convert char to string
        std::string temp;
        temp = c;
        if (temp == " ") continue;
        // conditions for handling numbers with more than one d
igit
        if (isoperator(temp) || previous == "" || isoperator(pr
evious) || parentheses(previous) || parentheses(temp))
            tokenized.push_back(temp);
        else {
            // if the previous and current are a digit
            // make them a single entry in the vector
            temp = previous + temp;
            tokenized.pop_back();
            tokenized.push_back(temp);
        }
        previous = temp;
    }

    tokenized.push_back(")");
    return tokenized;
}

node* tree_factory(std::vector<std::string> infix) {
    std::stack<node*> staccNodes;
    std::stack<std::string> staccTokens;

    node *t, *t1, *t2;

    for (std::string c: infix) {
        if (parentheses(c) == -1) {
            staccTokens.push(c);
        } else if (!isoperator(c)) {
            t = node_factory(c);
            staccNodes.push(t);
        } else if (operator_precedence(c) > 0) {
```

```
            while (
                !staccTokens.empty()&& staccTokens.top() != "("
                && ((c != "^" && operator_precedence(staccToken
s.top()) >= operator_precedence(c))
                || (c == "^" &&
                        operator_precedence(staccTokens.top())
> operator_precedence(c)
                ))
            ) {
                t = node_factory(staccTokens.top());
                staccTokens.pop();
                t1 = staccNodes.top();
                staccNodes.pop();
                t2 = staccNodes.top();
                staccNodes.pop();

                t->left = t2;
                t->right = t1;
                staccNodes.push(t);
            }
        } else if (c == ")") {
            while (!staccTokens.empty() && staccTokens.top() !=
 "(") {
                t = node_factory(staccTokens.top());
                staccTokens.pop();
                t1 = staccNodes.top();
                staccNodes.pop();
                t2 = staccNodes.top();
                staccNodes.pop();
                t->left = t2;
                t->right = t1;
                staccNodes.push(t);
            }
            staccTokens.pop();
        }
    }
    t = staccNodes.top();
    return t;
}
```

```
void inorder_traverse(node* root) {
    if (root != NULL) {
        inorder_traverse(root->left);
        std::cout << root->data << " ";
        inorder_traverse(root->right);
    }
}

void preorder_traverse(node* root) {
    if (root != NULL) {
        std::cout << root->data << " ";
        preorder_traverse(root->left);
        preorder_traverse(root->right);
    }
}

void postorder_traverse(node* root) {
    if (root != NULL) {
        postorder_traverse(root->left);
        postorder_traverse(root->right);
        std::cout << root->data << " ";
    }
}

int main() {
    std::string infix;
    std::cout << "Enter an infix expression: ";
    std::cin >> infix;

    std::vector<std::string> tokenized = tokenize(infix);
    node* tree = tree_factory(tokenized);
    std::cout << "In Order: ";
    inorder_traverse(tree);
    std::cout << "\nPostOrder: ";
    postorder_traverse(tree);
    std::cout << "\nPreOrder: ";
    preorder_traverse(tree);
    return 0;
}
```

Output:

```
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> g++ .\expression_tree.cpp
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> .\a.exe
Enter an infix expression: (a+b)-(c^(d*(g/r)))
In Order: a + b - c ^ d * g / r
PostOrder: a b + c d g r / * ^ -
PreOrder: - + a b ^ c * d / g r
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src> .\a.exe
Enter an infix expression: a+b+c
In Order: a + b + c
PostOrder: a b + c +
PreOrder: + + a b c
(base) PS D:\College\Data Structures and Algorithms\Lab\DSA-Assignments\Assignment 4\src>
```

Results:

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