Complete Analysis of the task

The problem we are trying to solve in this task is to collect a fully parenthesized arithmetic expression from the user and convert it into a binary expression tree. At the end, we will display this tree for the user. The main challenge for this code was to take the arithmetic expression and somehow break it down into different sections so I could build a tree. So for example, any operator will always be a parent. The user must provide the arithmetic expression in parentheses otherwise the code will have no way of telling apart of what should be a child or a parent node.

What algorithm did you use? Why?

The algorithm of this code is primarily around using stacks. To be exact there were two stacks used. One to save the operators and the other to save the nodes. And using the brackets I could either know if a new subtree is starting or if it is ending. And for this exact reason stacks are very useful for this task since they follow the rule of first in last out (FILO). Meaning that I could identify a sub expression just by the brackets. For example if I find a left bracket '(' I know that it means that we have a new expression and when I find a right bracket ')' I can pop every character until I find a left bracket '(' which means that all these characters belong to a sub expression. When it also comes to efficiency using stacks is a very viable option for this task. As pushing and popping from the top of the stack will only give us a time complexity of O(1).

What other algorithms apply to this task?

Another algorithm that I can think of that could have been used here is a recursive approach instead of using a linear approach like stacks. Also, an algorithm that could handle un-parenthesized expression would also be an alternative but more complex to implement.

In what ways does the algorithm you used differ from these other algorithms?

The stack approached that I have used here require fully parenthesized arithmetic expressions while other algorithms may not. However, while the algorithm would be harder to implement it could be more suitable for larger scale problems and how it handles them.

How does the code work? A detailed explanation of the functions made.

The entire code includes comments that explain each functions made in details.

Time Complexity analysis of the algorithm

The worst-case scenario for this code is a time complexity of O(n). We can deduct this conclusion from the fact that when constructing the expression tree we will iterate over each character only once.

Code:

```
#include <iostream>

#include <string>

#include <stack>

#include <cctype>

using namespace std;


// Define our class

class Node {

public:

    // declaring a variable to store our values

    string value;

    // Defining a pointer to the left child of our tree

    Node* left;

    // Defining a pointer to the right child of our tree

    Node* right;


    // Constructor for our class that allows us to create a node

    Node(string val) : value(val), left(nullptr), right(nullptr) {}


    // Destructor to delete left and right children pointers for the tree

    ~Node() {
        delete left;

        delete right;

    }

};


// Define a class for our tree

class Tree {

public:

    // Declaring a variable to hold our root of the tree

    Node* root;


    // Initializing a constructor and setting the value of the root to null

    Tree() : root(nullptr) {}


    // And again we are calling a destructor to free up the memory

    ~Tree() {

        delete root;

    }


    // This function we will useful later as we will call it to check if a character is an operator. As
this is important when differentiate the input from the user

    bool isOperator(char c) {

        return c == '+' || c == '-' || c == '*' || c == '/';

    }


    // This function is to construct the tree based of the expression provided by the user

    Node* constructTree(const string& expression) {
        // Stack to hold the nodes
        stack<Node*> nodes;
        // Stack to hold the operator
        stack<char> operators;

        // This for loop iterates over each input in the expression while also skipping any spaces
        for (int i = 0; i < expression.length(); i++) {
            if (isspace(expression[i])) {
                continue;
            }

            // the reason we specified to the user that we need a paranthesized expression is due to the fact that we use the brackets to understand the expression that is provided to us
            // If we find a "(" we will push it to the operator stack we defined before
            if (expression[i] == '(') {
                operators.push(expression[i]);
            }

            // But if we find that the current character we are on is a digit we will parse the entire number
            else if (isdigit(expression[i])) {
                // Using a new variable we will save the numbers as a string since we could have more than one number
                string val;
                // This while loop will iterate over every number and append it to the string called val
                while (i < expression.length() && isdigit(expression[i])) {
                    val += expression[i++];
                }
                // Now we need to decrement i since the while loop increments i and therefore would otherwhise the next character
                i--;
                // Now all the numeric values we have collected will be saved as a node and pushed to the node stack
                nodes.push(new Node(val));
            }
            // If the current character we are on is an operator we will push it to the operator stack
            else if (isOperator(expression[i])) {
                operators.push(expression[i]);
            }


            // If the current character we are on is a ')' we know this indicates that a part of the expression is complete. This is our cue to build a subtree.
            else if (expression[i] == ')') {
                // Something important to note here is that the operator.top() != '(' is necessary here to keep track if we have processed all necessary characters in the subtree.
                // Since as soon as we encounter an '(' it means that we have reached the end of this subtree as a new one is starting
                while (!operators.empty() && operators.top() != '(') {
                    // We are now going to create a node for our operator symbol. Since our operator symbols are always going to be parent nodes. We will aslo pop the operator of the stack since we have now used it
                    Node* opNode = new Node(string(1, operators.top()));
                    operators.pop();


                    // Now we will pop the and store the value in a variable from each side. Which is the left and right child of the parent The first child will always be our right operand
                    Node* right = nodes.top(); nodes.pop();
                    Node* left = nodes.top(); nodes.pop();


                    // After popping the nodes we will now set our left and right child
                    opNode->left = left;
                    opNode->right = right;


                    // After we finish this we will push our complete subtree to the stack
                    nodes.push(opNode);
                }
                // Now we also cannot forget to pop the '(' from the stack to signal that we have
    finished the subtree
                operators.pop();
            }


        }
        // The top of the stack is going to be root of our tree
        return nodes.top();


    }




    // Function to print the tree in an in order traversal as well as adding indentation to make the
    presentation neat and easy to comprehend
    void printInOrder(Node* node, int depth = 0) {
        if (!node) return;
        // Print right subtree.
        printInOrder(node->right, depth + 1);
```

```
        // Print current node.
        cout << string(depth * 4, ' ') << node->value << endl;
        // Print left subtree.
        printInOrder(node->left, depth + 1);
    }
};

int main() {
    // First we will collect the arithmetic expression from the user
    string expression;
    cout << "Enter a fully parenthesized expression ex. ((4+1)*(7-3)): ";
    getline(cin, expression);

    // Now we will construct a tree based of the input collected previously
    Tree tree;
    Node* root = tree.constructTree(expression);

    cout << "Printed tree structure:\n";
    // Calling our function to print our tree
    tree.printInOrder(root);

    return 0;
}
```