Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 6 з дисципліни
«Основи програмування-2.
Методології програмування.»

«Дерева»

Варіант 28

Виконав студент ІП-11 Сідак Кирил Ігорович
(шифр, прізвище, ім'я, по батькові)

Перевірив _____
(прізвище, ім'я, по батькові)

Київ 2022

Лабораторна робота №6

Мета: вивчити особливості організації і обробки дерев.

Варіант 28

Написати програму, що будує дерево-формулу та перетворює в ньому всі піддерева, що відповідають формулам $((f_1 \pm f_2) * f_3)$ на піддерева виду $((f_1 * f_3) \pm (f_2 * f_3))$.

Постановка задачі: за умовою задачі треба на основі введеної формули, де кожна арифметична операція відокремлена дужками, а кожний символ записаний через пробіл, створити бінарне дерево-формулу, тобто дерево, листками якого є числа. Якщо воно містить хоча б одне піддерево, що відповідає формулі $((f_1 \pm f_2) * f_3)$, то всі такі піддерева треба перетворити на піддерева виду $((f_1 * f_3) \pm (f_2 * f_3))$.

Програма на C++:

main.cpp

```
#include "tree_operations.h"
using namespace std;

int main() {
    string formula;
    cout << "Enter the formula, where each arithmetic
operation is seperated with brackets and symbols are
seperated with spaces:" << endl;
    getline(cin, formula);
    vector<string> formula_vector = split(formula);
    Tree* tree = create_tree(formula_vector);
    cout << "Initial tree:" << endl;
    display_tree("", tree, false);
    create_final_tree(tree);
    cout << "Final tree:" << endl;
    display_tree("", tree, false);
    delete tree;
    return 0;
}
```

Tree.h

```
#ifndef LAB_6_TREE_H
#define LAB_6_TREE_H
#include <iostream>
using namespace std;
class Tree
{
    string key;
    Tree* left, * right;
public:
    explicit Tree(string key) { this -> key = key; left =
nullptr; right = nullptr; }
    void insert_left_child(const string&);
    void insert_right_child(const string&);
```

```
    Tree* get_left_child() { return left; }
    Tree* get_right_child() { return right; }
    void set_right_child(Tree* node) { right = node; }
    void set_left_child(Tree* node) { left = node; }
    string get_root() { return key; }
    void set_root(string key) { this -> key = key; }
    bool is_leaf();
};
#endif
```

Tree.cpp

```
#include "Tree.h"
void Tree::insert_left_child(const string& new_key) {
    if (left == nullptr)
        left = new Tree(new_key);
    else {
        Tree* p = new Tree(new_key);
        p -> left = this -> left;
        this-> left = p;
    }
}

void Tree::insert_right_child(const string& new_key) {
    if (right == nullptr)
        right = new Tree(new_key);
    else {
        Tree* p = new Tree(new_key);
        p -> right = this -> right;
        this-> right = p;
    }
}

bool Tree::is_leaf() {
    if (left == nullptr && right == nullptr) {
        return true;
    }
    return false;
}
```

tree_operations.h

```
#ifndef LAB_6_TREE_OPERATIONS_H
#define LAB_6_TREE_OPERATIONS_H
#include <iostream>
#include <vector>
#include "Tree.h"
using namespace std;
vector<string> split(string);
Tree* create_tree(const vector<string>&);
void display_tree(const string&, Tree*, bool);
void create_final_tree(Tree*);
```

```
void modify_tree(Tree*);
#endif
```

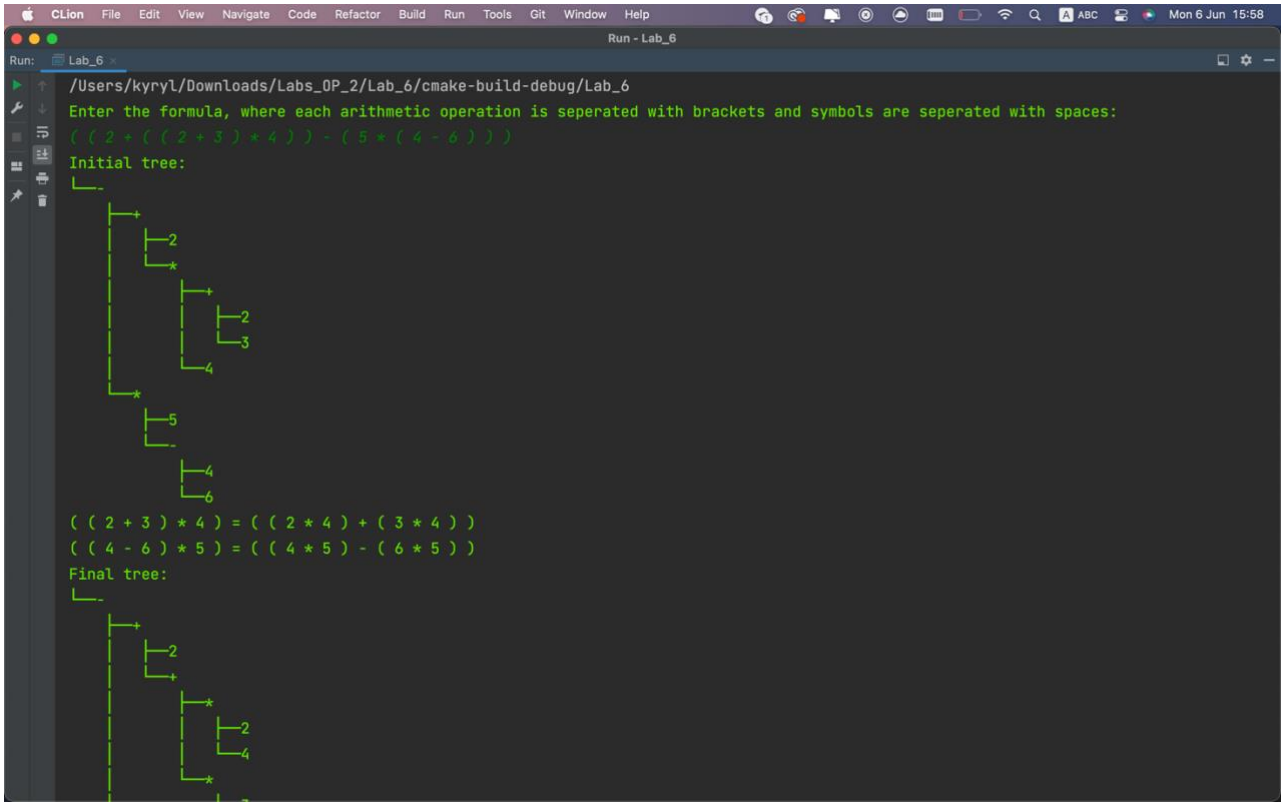
tree_operations.cpp

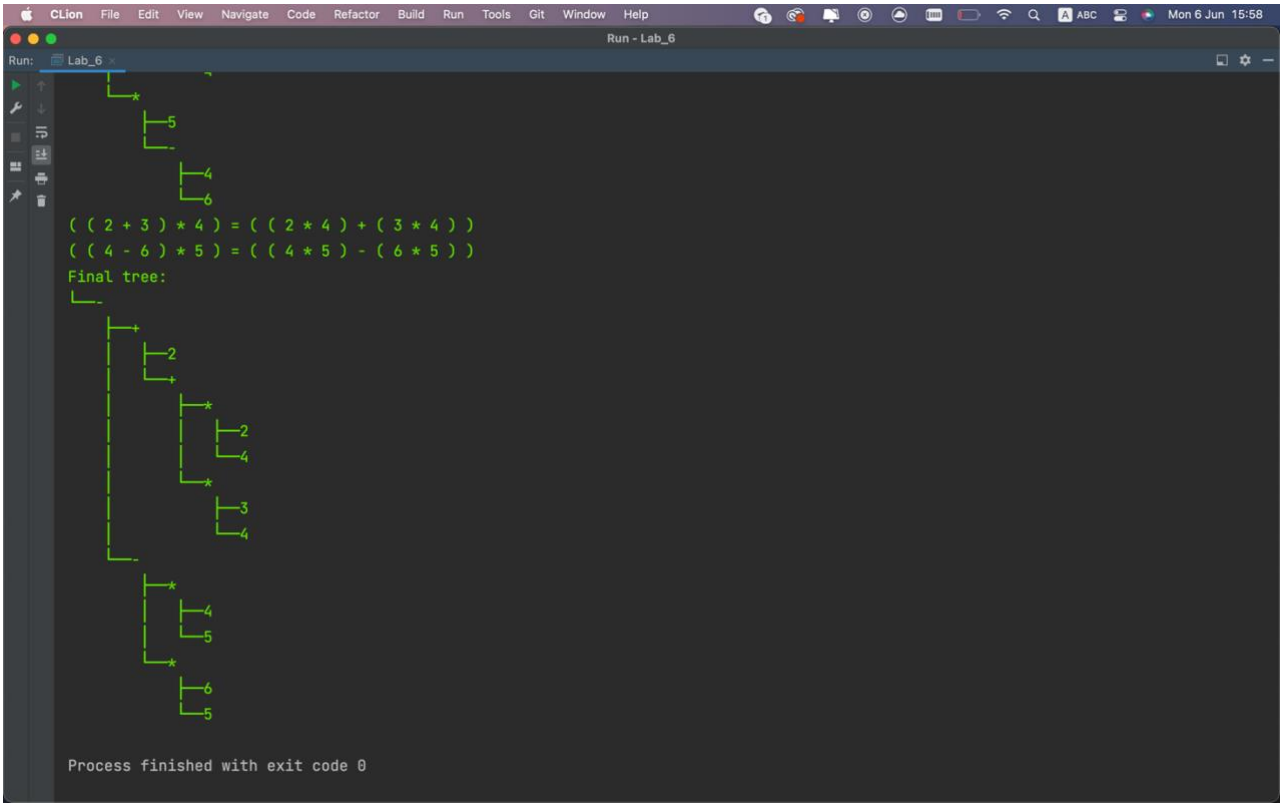
```
#include "tree_operations.h"
#include <stack>

vector<string> split(string formula) {
    vector<string> formula_vector;
    int pos;
    while (formula.find(' ') != string::npos) {
        pos = formula.find(' ');
        formula_vector.push_back(formula.substr(0, pos));
        formula = formula.substr(pos + 1);
    }
    formula_vector.push_back(formula);
    return formula_vector;
}

Tree* create_tree(const vector<string>& formula) {
    stack<Tree*> tree_stack;
    Tree *tree = new Tree("");
    tree_stack.push(tree);
    Tree* current_tree = tree;
    string signs = "+-*/";
    for (string i : formula) {
        if (i == "(") {
            current_tree -> insert_left_child("");
            tree_stack.push(current_tree);
            current_tree = current_tree -> get_left_child();
        }
        else if (i == ")")
        {
            current_tree = tree_stack.top();
            tree_stack.pop();
        }
        else if (signs.find(i) != string::npos) {
            current_tree -> set_root(i);
            current_tree -> insert_right_child("");
            tree_stack.push(current_tree);
            current_tree = current_tree -> get_right_child();
        }
        else if (signs.find(i) == string::npos) {
            current_tree -> set_root(i);
            current_tree = tree_stack.top();
            tree_stack.pop();
        }
    }
    return tree;
}
```

```
void display_tree(const string& prefix, Tree* tree, bool
is_left) {
    if (tree != nullptr) {
        cout << prefix;
        cout << (is_left ? "├──" : "└──" );
        cout << tree -> get_root() << endl;
        display_tree( prefix + (is_left ? "│   " : "    "),
tree -> get_left_child(), true);
        display_tree( prefix + (is_left ? "│   " : "    "),
tree->get_right_child(), false);
    }
}

void create_final_tree(Tree* tree) {
    if (tree) {
        modify_tree(tree);
        create_final_tree(tree -> get_left_child());
        create_final_tree(tree -> get_right_child());
    }
}

void modify_tree(Tree* tree) {
    Tree* left = tree -> get_left_child();
    Tree* right = tree -> get_right_child();
    string plus1, plus2, mult, symb;
    if (tree -> get_root() == "*") {
        if (left -> is_leaf())
            mult = left-> get_root();
        if (right->is_leaf())
            mult = right-> get_root();
    }
    if (!mult.empty()) {
        if (left -> get_root() == "+" || left -> get_root()
== "-") {
            symb = left -> get_root();
            if(left -> get_left_child() -> is_leaf())
                plus1 = left -> get_left_child() ->
get_root();
            if (left -> get_right_child() -> is_leaf())
                plus2 = left -> get_right_child() ->
get_root();
        }
        if (right -> get_root() == "+" || right -> get_root()
== "-") {
            symb = right -> get_root();
            if (right -> get_left_child() -> is_leaf())
                plus1 = right -> get_left_child() ->
get_root();
            if (right -> get_right_child() -> is_leaf())
                plus2 = right -> get_right_child() ->
get_root();
```

```
        }
    }
    if (!mult.empty() && !plus1.empty() && !plus2.empty()) {
        string formula = "( ( " + plus1 + " * " + mult + " )
" + symb + " ( " + plus2 + " * " + mult + " ) )";
        string old_formula = "( ( " + plus1 + " " + symb + "
" + plus2 + " ) * " + mult + " )";
        cout << old_formula << " = " << formula << endl;
        vector<string> formula_vector = split(formula);
        Tree* new_tree;
        new_tree = create_tree(formula_vector);
        tree -> set_root(symb);
        tree -> set_right_child(new_tree ->
get_right_child());
        tree -> set_left_child(new_tree -> get_left_child());
    }
}
```

Результат на C++:





Висновок

Отже, я вивчив особливості організації і обробки дерев та застосував ці знання на практиці, створивши програму, яка перетворює введену формулу, де кожна арифметична операція відокремлена дужками, на бінарне дерево-формулу, тобто дерево, в якому числа є листками, реалізував перетворення всіх піддерев цього дерева, що відповідають формулам $((f_1 \pm f_2) * f_3)$ на піддерева виду $((f_1 * f_3) \pm (f_2 * f_2 * f_3))$ та отримав коректний результат.