# 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 на піддерева виду .

**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 , то всі такі піддерева треба перетворити на піддерева виду .

**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;  
}

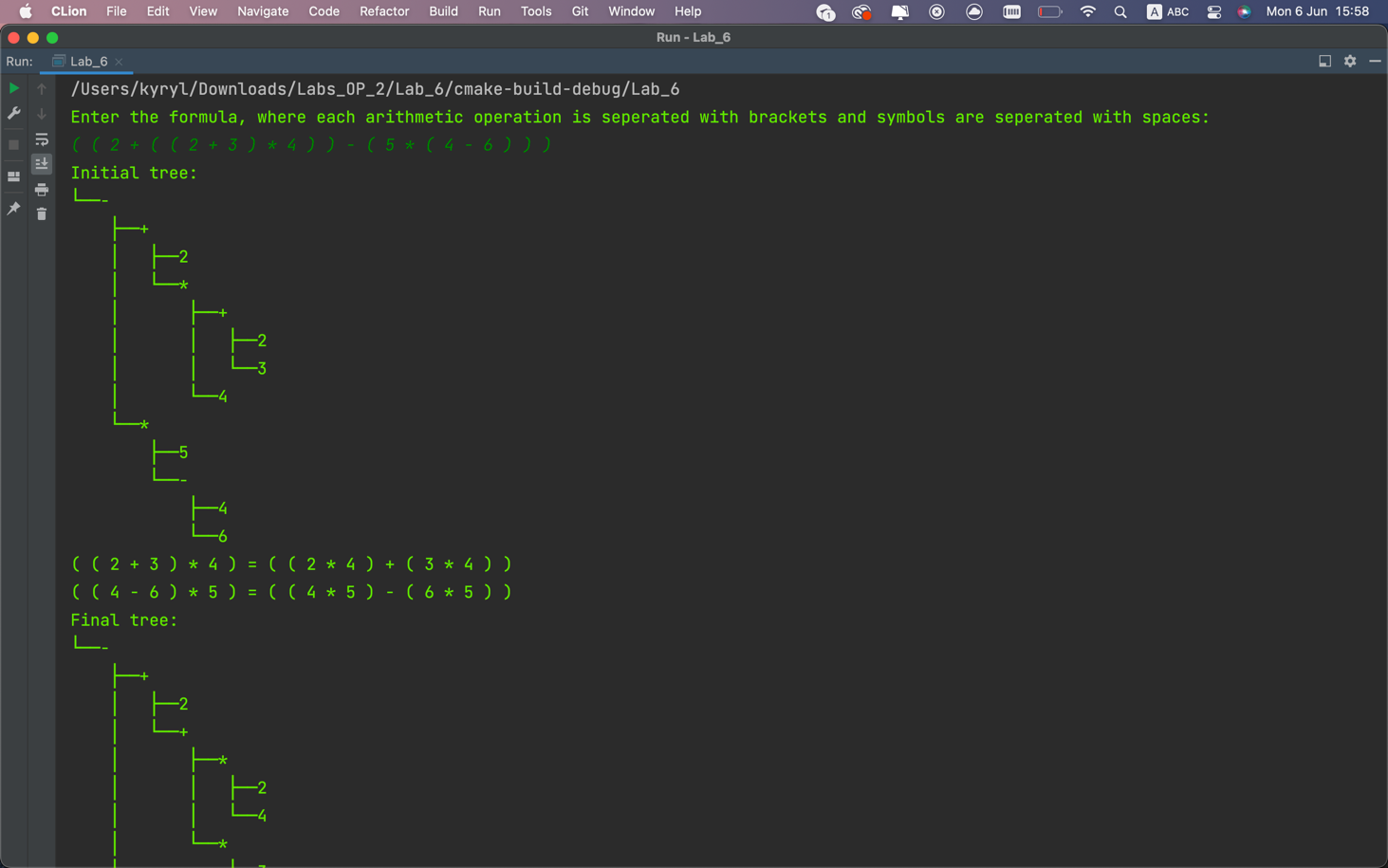
**tree\_operations.h**

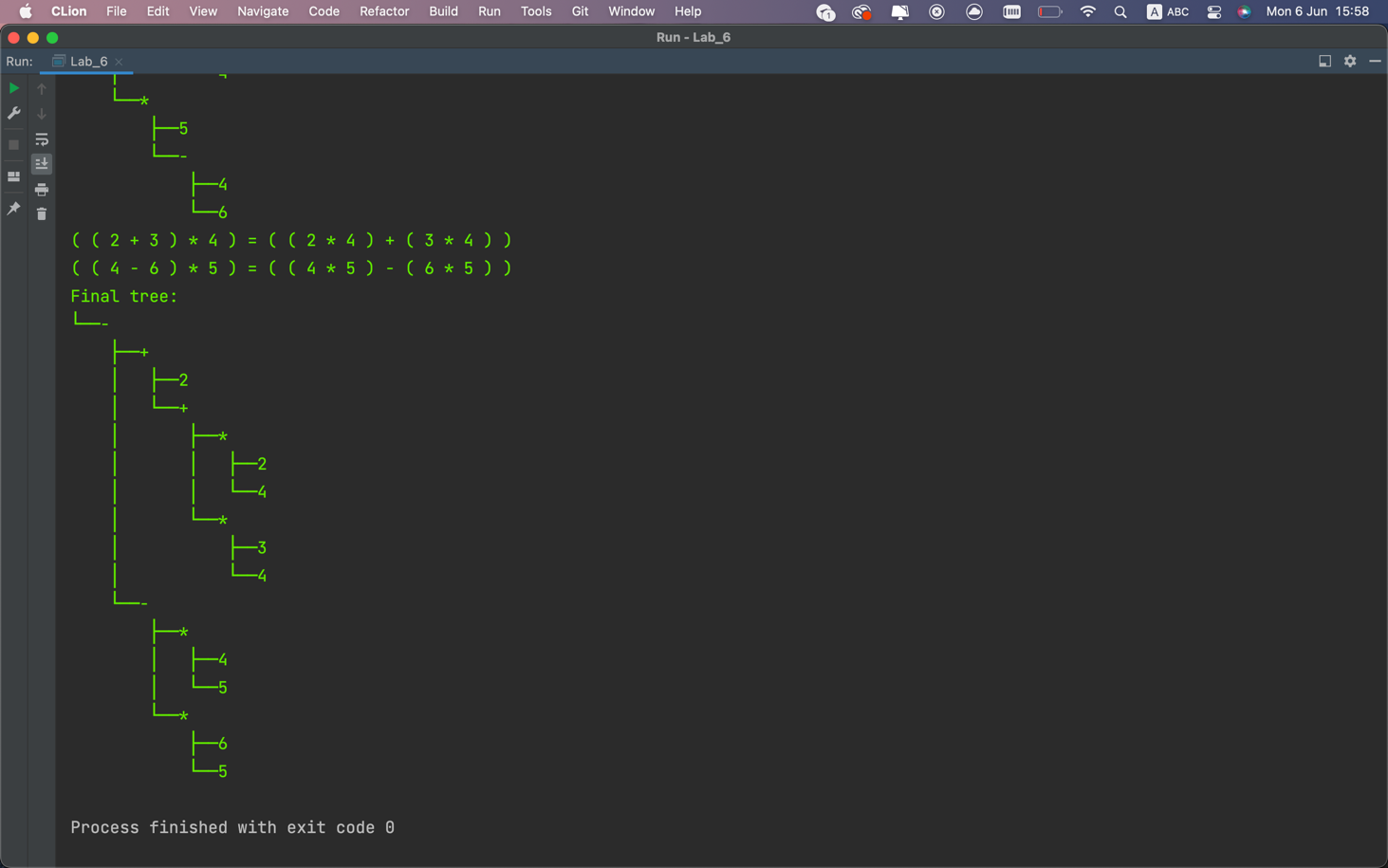
#ifndef LAB\_6\_TREE\_OPERATIONS\_H  
#define LAB\_6\_TREE\_OPERATIONS\_H  
#include <iostream>  
#include <vector>  
#include "Tree.h"  
using namespace std;  
vector<string> split(string);  
Tree\* create\_tree(const vector<string>&);  
void display\_tree(const string&, Tree\*, bool);  
void create\_final\_tree(Tree\*);  
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 на піддерева виду та отримав коректний результат.