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 № 5 з дисципліни

«Основи програмування - 2.

Модульне програмування.»

“Дерева”

Варіант: 30

Виконав студент ІП-11 Тихонов Федір Сергійович

(шифр, прізвище, ім'я, по батькові)

Перевірив

(прізвище, ім'я, по батькові)

**ЗМІСТ**

[**Мета**](#_ii9hyxrgysmu)

[**Завдання**](#_1giglz907kdo)

[**Постановка задачі**](#_1tgkxkj91i5a)

[**Код на С++**](#_cwttflz7qr96)

[**main.cpp**](#_isr5dl1uxn69)

[**lib.cpp**](#_3vyszs6w70a1)

[**lib.h**](#_9aex44eun6l1)

[**Результати на С++**](#_1e2xi43q5tq)

[**Висновок**](#_9vlvf03akx5j)

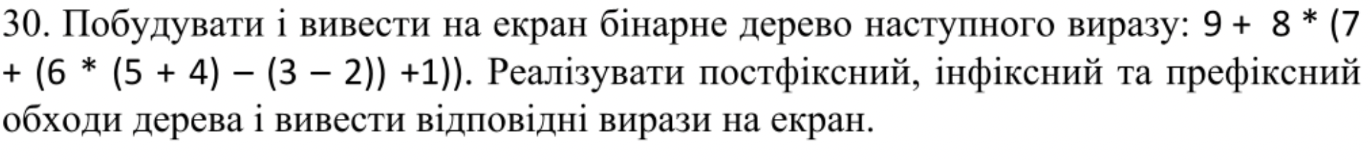
**Лабораторна робота №5**

**“Дерева”**

# **Мета**:

Вивчити механізм створення і використання об’єктів класів.

# **Завдання:**



# **Постановка задачі:**

Маємо деякий вираз, який потрібно перетворити на бінарне дерево операндів, після чого потрібно його обійти постфіксно, префіксно та інфіксно та вивести його наочними способами.

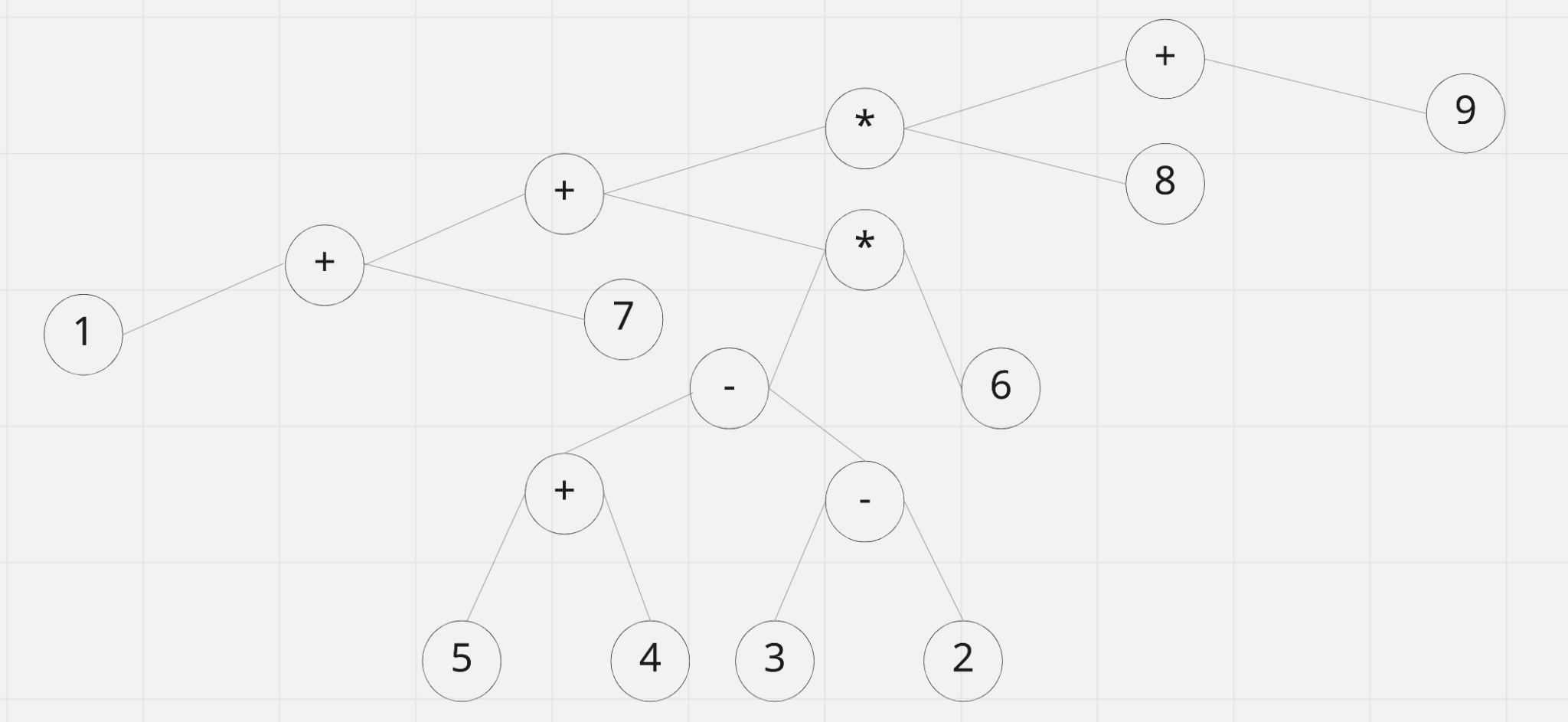


Рисунок 1 - Відображення дерева даного виразу

# **Код на С++:**

## **main.cpp**

#include "lib.h"

int main()

{

mainCode();

return 0;

}

## **lib.cpp**

#include "lib.h"

bool isOperator(char c)

{

if (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^') {

return true;

}

else {

return false;

}

}

int precedence(char c)

{

if (c == '^')

return 3;

else if (c == '\*' || c == '/')

return 2;

else if (c == '+' || c == '-')

return 1;

else

return -1;

}

std::string InfixToPrefix(std::stack<char> s, std::string infix)

{

std::string prefix;

reverse(infix.begin(), infix.end());

for (int i = 0; i < infix.length(); i++) {

if (infix[i] == '(') {

infix[i] = ')';

}

else if (infix[i] == ')') {

infix[i] = '(';

}

}

for (int i = 0; i < infix.length(); i++) {

if (infix[i] >= '1' && infix[i] <= '9'){

prefix += infix[i];

}

else if (infix[i] == '(') {

s.push(infix[i]);

}

else if (infix[i] == ')') {

while ((s.top() != '(') && (!s.empty())) {

prefix += s.top();

s.pop();

}

if (s.top() == '(') {

s.pop();

}

}

else if (isOperator(infix[i])) {

if (s.empty()) {

s.push(infix[i]);

}

else {

if (precedence(infix[i]) > precedence(s.top())) {

s.push(infix[i]);

}

else if ((precedence(infix[i]) == precedence(s.top()))

&& (infix[i] == '^')) {

while ((precedence(infix[i]) == precedence(s.top()))

&& (infix[i] == '^')) {

prefix += s.top();

s.pop();

}

s.push(infix[i]);

}

else if (precedence(infix[i]) == precedence(s.top())) {

s.push(infix[i]);

}

else {

while ((!s.empty()) && (precedence(infix[i]) < precedence(s.top()))) {

prefix += s.top();

s.pop();

}

s.push(infix[i]);

}

}

}

}

while (!s.empty()) {

prefix += s.top();

s.pop();

}

reverse(prefix.begin(), prefix.end());

return prefix;

}

bool isNum(char c){

std::string numbers = "1234567890";

for(const char& num : numbers){

if(c == num){

return true;

}

}

return false;

}

ExpTree::ExpTree(const std::string& expression) {

root = new TreeNode;

root->value = expression[0];

root->right = new TreeNode;

root->right->value = expression[1];

TreeNode\* tmpNode = root;

for(int i = 1; i < expression.length(); i++){

TreeNode\* newNode = new TreeNode;

newNode->value = expression[i];

if(isNum(expression[i])){

if(tmpNode->right == nullptr && tmpNode->left == nullptr){

tmpNode->right = newNode;

newNode->parent = tmpNode;

} else if (tmpNode->right != nullptr && tmpNode->left == nullptr){

tmpNode->left = newNode;

newNode->parent = tmpNode;

} else{

tmpNode = root;

int placed = 0;

while (placed == 0){

if(tmpNode->left != nullptr){

tmpNode = tmpNode->left;

}

else if(tmpNode->left == nullptr){

tmpNode->left = newNode;

newNode->parent = tmpNode;

placed = 1;

}

}

}

}

else if(isOperator(expression[i])){

if(tmpNode != root && tmpNode->parent->right != nullptr){

tmpNode->left = newNode;

newNode->parent = tmpNode;

tmpNode = newNode;

}

else if(tmpNode == root){

tmpNode->left = newNode;

newNode->parent = tmpNode;

tmpNode = newNode;

}

else{

tmpNode = tmpNode->parent;

tmpNode->right = newNode;

newNode->parent = tmpNode;

tmpNode = newNode;

}

}

}

}

void ExpTree::printSymmetrically(ExpTree::TreeNode \*node, int lvl) {

if(node != nullptr){

printSymmetrically(node->left, lvl + 1);

for (int i = 0; i < lvl; i++) std::cout << " ";

std::cout << node->value << std::endl;

printSymmetrically(node->right, lvl + 1);

}

}

void ExpTree::printPre(ExpTree::TreeNode \*node) {

if(node == nullptr){

return;

}

std::cout << node->value << " ";

printPre(node->left);

printPre(node->right);

}

void ExpTree::printPost(ExpTree::TreeNode \*node) {

if(node == nullptr){

return;

}

printPost(node->left);

printPost(node->right);

std::cout << node->value << " ";

}

void ExpTree::printIn(ExpTree::TreeNode \*node) {

if(node == nullptr){

return;

}

printIn(node->left);

std::cout << node->value << " ";

printIn(node->right);

}

void ExpTree::printDirectly(ExpTree::TreeNode \*node, int lvl) {

if (node) {

for (int i = 0; i < lvl-1; i++) std::cout << " ";

if (lvl != 0) std::cout << "----";

std::cout << node->value << std::endl;

printDirectly(node->right, lvl + 1);

printDirectly(node->left, lvl + 1);

}

}

void mainCode(){

setlocale(LC\_ALL, "");

std::string infix, prefix;

infix = "9+8\*(7+6\*((5+4)-(3-2))+1)";

std::stack<char> stack;

prefix = InfixToPrefix(stack, infix);

ExpTree k(prefix);

std::cout << "Пpeфіксний обхід:" << std::endl;

k.printPre(k.root);

std::cout << std::endl;

std::cout << "Пocтфіксний обхід:" << std::endl;

k.printPost(k.root);

std::cout << std::endl;

std::cout << "Iнфіксний обхід:" << std::endl;

k.printIn(k.root);

std::cout << std::endl;

std::cout << "Прямий обхід:" << std::endl;

k.printDirectly(k.root, 1);

std::cout << "Симетричний обхід:" << std::endl;

k.printSymmetrically(k.root, 1);

}

## **lib.h**

#pragma once

#include <iostream>

#include <stack>

#include <vector>

#include <algorithm>

class ExpTree{

public:

class TreeNode{

public:

TreeNode\* left;

TreeNode\* right;

TreeNode\* parent;

char value;

};

ExpTree() { root->right = nullptr; root->left = nullptr; root->parent = nullptr; };

explicit ExpTree(const std::string& expression);

TreeNode\* root;

void printSymmetrically(TreeNode\* node, int lvl);

void printDirectly(TreeNode\* node, int lvl);

void printPre(TreeNode\* node);

void printPost(TreeNode\* node);

void printIn(TreeNode\* node);

};

bool isOperator(char c);

bool isNum(char c);

int precedence(char c);

std::string InfixToPrefix(std::stack<char> s, std::string infix);

void mainCode();

# **Результати на С++:**



# **Висновок:**

Отже, ми навчилися використовувати на практиці основні принципи роботи з бінарними деревами операнд: навчилися їх обходити, наочно їх виводити та створювати дерева операнд зі звичайних виразів, що допоможе при вивченні компілювання у майбутньому.