**Отчет по лабораторной работе № 24 по курсу**

**«Практикум программирования»**

Студент группы М8О-109Б-22 Концебалов Олег Сергеевич

Контакты: telegram @baronpipistron

Работа выполнена: 20.05.2023

Преподаватель: каф.806 Сысоев Максим Алексеевич

Отчет сдан «25» июня 2023 г., итоговая оценка \_\_\_\_

Подпись преподавателя \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Тема:** Деревья выражений

**2. Цель работы:** Построить и обработать дерево выражений, а так же выполнить специальное действие

**3. Задание (вариант № 29):** Выполнить замену переменной на выражение

**4. Оборудование (студента):**

Процессор AMD Ryzen 5 5600H with Radeon Graphics 3.30 GHz, ОП 16,0 Гб, SSD 512 Гб. Монитор 1920x1080 144 Hz

**5. Программное обеспечение (студента):**

Операционная система семейства Linux, наименование Ubuntu, версия 18.10

Интерпретатор команд: bash, версия 4.4.19

Система программирования – версия --, редактор текстов Emacs, версия 25.2.2

Утилиты операционной системы –

Прикладные системы и программы –

Местонахождение и имена файлов программ и данных на домашнем компьютере –

**6. Идея, метод, алгоритм решения задачи** *(в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)*

Строю дерево выражений, затем строю поддерево для выражения, на которое необходимо заменить, ищу переменную в дереве и произвожу замену

**7. Сценарий выполнения работы** *(план работы, первоначальный текст программы в черновике [можно на отдельном листе] и тесты, либо соображения по тестированию)*

1. Читаю про деревья выражений
2. Пишу алгоритм Дейкстры для польской записи
3. Делаю дерево и пишу код функции специального действия

**8. Распечатка протокола** *(подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем)*

***Node.hpp***

#ifndef NODE\_HPP

#define NODE\_HPP

class ExpressionTree;

class Node{

    friend class ExpressionTree;

private:

    char data;

    Node\* left;

    Node\* right;

public:

    Node();

    Node(const char data);

    Node(const char data, Node\* left, Node\* right);

};

#endif

***Node.cpp***

#include "Node.hpp"

Node::Node(): data(0), left(nullptr), right(nullptr){}

Node::Node(const char data): data(data), left(nullptr), right(nullptr){}

Node::Node(const char data, Node\* left, Node\* right): data(data), left(left), right(right){}

***Stack.hpp***

#ifndef STACK\_HPP

#define STACK\_HPP

#include "MyVector.hpp"

#include "MyVector.cpp"

#include <iostream>

template <typename T>

class Stack{

private:

    myVector<T> stack;

public:

    Stack() = default;

    Stack(const std::initializer\_list<T>& list);

    ~Stack() = default;

    size\_t size() const;

    bool empty() const;

    void push(const T& value);

    void pop();

    T& top();

    const T& top() const;

};

#endif

***Stack.cpp***

#include "Stack.hpp"

#include <iostream>

template <typename T>

Stack<T>::Stack(const std::initializer\_list<T>& list){

    for (T elem: list){

        stack.push\_back(elem);

    }

}

template <typename T>

size\_t Stack<T>::size() const{

    return stack.vec\_size;

}

template <typename T>

bool Stack<T>::empty() const{

    return stack.vec\_size == 0;

}

template <typename T>

void Stack<T>::push(const T& value){

    stack.push\_back(value);

}

template <typename T>

void Stack<T>::pop(){

    stack.pop\_back();

}

template <typename T>

T& Stack<T>::top(){

    return stack.back();

}

template <typename T>

const T& Stack<T>::top() const{

    return stack.back();

}

***ExpressionTree.hpp***

#ifndef EXPRESSIONTREE\_HPP

#define EXPRESSIONTREE\_HPP

#include "Node.hpp"

#include <iostream>

#include <string>

class ExpressionTree{

private:

    Node\* root;

public:

    ExpressionTree();

    ExpressionTree(const std::string& expression);

    ~ExpressionTree();

    Node\* getRoot() const;

    void deleteTree(Node\* node);

    Node\* createTree(const std::string& postfix);

    std::string doPostfix(const std::string& expression);

    void replace(char var, const std::string& expression);

    Node\* replace(Node\* root, char var, Node\* expression\_tree);

    void printPostfix(Node\* root) const;

    void printInfix(Node\* root) const;

    void printTree(Node\* root, const size\_t height = 0) const;

};

#endif

***ExpressionTree.cpp***

#include "ExpressionTree.hpp"

#include "Stack.hpp"

#include "Stack.cpp"

#include "Node.hpp"

#include "Node.cpp"

ExpressionTree::ExpressionTree(): root(nullptr){}

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

    std::string postfix = doPostfix(expression);

    root = createTree(postfix);

}

ExpressionTree::~ExpressionTree(){

    deleteTree(root);

}

Node\* ExpressionTree::getRoot() const{

    return this->root;

}

size\_t getPriority(char c){

    if (c == '+' || c == '-'){

        return 1;

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

        return 2;

    } else if (c == '^'){

        return 3;

    } else if (c == '~'){

        return 4;

    }

    return 0;

}

bool isOperator(char c){

    return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

std::string ExpressionTree::doPostfix(const std::string& expression){

    std::string postfix = "";

    Stack<char> stack;

    for (size\_t i = 0; i != expression.size(); ++i){

        char c = expression[i];

        if (!isOperator(c) && c != '(' && c != ')'){

            postfix += c;

        } else if (c == '('){

            stack.push(c);

        } else if (c == ')'){

            while (stack.top() != '('){

                postfix += stack.top();

                stack.pop();

            }

            stack.pop();

        } else{

            if (c == '-' && expression[i - 1] == '(') c = '~';

            while (!stack.empty() && (getPriority(stack.top()) >= getPriority(c))){

                postfix += stack.top();

                stack.pop();

            }

            stack.push(c);

        }

    }

    while (!stack.empty()){

        postfix += stack.top();

        stack.pop();

    }

    return postfix;

}

void ExpressionTree::deleteTree(Node\* node){

    if (node == nullptr) return;

    deleteTree(node->left);

    deleteTree(node->right);

    delete node;

}

Node\* ExpressionTree::createTree(const std::string& postfix){

    if (postfix.length() == 0) return nullptr;

    Stack<Node\*> stack;

    for (char c: postfix){

        if (c == '~'){

            Node\* node\_x = stack.top();

            stack.pop();

            Node\* node = new Node('-', nullptr, node\_x);

            stack.push(node);

        } else if (isOperator(c)){

            Node\* node\_x = stack.top();

            stack.pop();

            Node\* node\_y = stack.top();

            stack.pop();

            Node\* node = new Node(c, node\_y, node\_x);

            stack.push(node);

        } else{

            stack.push(new Node(c));

        }

    }

    return stack.top();

}

void ExpressionTree::printPostfix(Node\* root) const{

    if (root == nullptr) return;

    printPostfix(root->left);

    printPostfix(root->right);

    std::cout << root->data;

}

void ExpressionTree::printInfix(Node\* root) const{

    if (root == nullptr) return;

    if (isOperator(root->data)){

        std::cout << '(';

    }

    printInfix(root->left);

    std::cout << root->data;

    printInfix(root->right);

    if (isOperator(root->data)){

        std::cout << ')';

    }

}

void ExpressionTree::printTree(Node\* root, const size\_t height) const{

    if (root != nullptr){

        printTree(root->right, height + 1);

        for (size\_t i = 0; i < height; ++i){

            std::cout << "\t";

        }

        std::cout << root->data << "\n";

        printTree(root->left, height + 1);

    }

}

void ExpressionTree::replace(char var, const std::string& expression){

    Node\* expression\_tree = createTree(doPostfix(expression));

    root = replace(root, var, expression\_tree);

}

Node\* ExpressionTree::replace(Node\* root, char var, Node\* expression\_tree){

    if (root == nullptr) return root;

    if (root->data == var){

        Node\* new\_node = new Node(\*expression\_tree);

        delete root;

        root = new\_node;

        return root;

    }

    root->left = replace(root->left, var, expression\_tree);

    root->right = replace(root->right, var, expression\_tree);

    return root;

}

***Run.cpp***

#include "ExpressionTree.hpp"

#include "ExpressionTree.cpp"

#include <iostream>

#include <string>

int main(){

    std::string expr = "(-2)\*a+4\*5/a+3-a^a^a-b";

    ExpressionTree tree(expr);

    tree.printInfix(tree.getRoot());

    std::cout << "\n";

    tree.printPostfix(tree.getRoot());

    std::cout << "\n";

    tree.printTree(tree.getRoot());

    std::cout << "\n\n\n";

    tree.replace('a', "i+4");

    tree.printTree(tree.getRoot());

}

**9. Дневник отладки** *(дата и время сеансов отладки и основные события [ошибки в сценарии и программе, нестандартные ситуации] и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы)*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *№* | *Лаб. или дом* | *Дата* | *Время* | *Событие* | *Действие по исправлению* | *Примечания* |

Проблем при выполнении лабы не возникло

**10. Замечания автора** *(по существу работы)*

Замечания отсутствуют

**11. Вывод**

При выполнении лабораторной познакомился с деревьями выражений. В целом довольно интересная вещь, особенно реализация, но нет идей, где получится применить их на практике. Получилось самому реализовать алгоритм Дейкстры для перевода в обратную польскую запись. Немного пришлось повозиться с унарным минусом, но в остальном все неплохо. Лаба понравилась

Работа на 9/10

Подпись студента \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_