МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студент гр. 9304 Ламбин А.В.

Преподаватель Филатов А.Ю.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с понятиями дерева, леса и бинарного дерева, реализовать бинарное дерево для решения поставленной задачи с помощью языка программирования C++.

Задание.

Вариант 15.

Необходимо:

- для заданной формулы f построить дерево-формулу t;
- для заданного дерева-формулы t напечатать соответствующую формулу f;
- преобразовать дерево-формулу t, заменяя в нем все поддеревья, соответствующие формулам $((f1 * f2) + (f1 * f3))$ и $((f1 * f3) + (f2 * f3))$, на поддеревья, соответствующие формулам $(f1 * (f2 + f3))$ и $((f1 + f2) * f3)$.

Выполнение работы.

При запуске программы из файла или стандартного потока ввода считывается строка, являющаяся формулой, из которой удаляются все пробелы. Если строка некорректна, выводится информация об ошибке, и программа завершается. Корректность строки определяют статические методы класса *Tree*: *checkString()* и *recursionCheck()*. В остальных случаях создаётся бинарное дерево – экземпляр класса *Tree*. У дерева вызывается метод *transform()*, преобразующий это дерево. Итоговое дерево выводится в стандартный поток вывода или записывается в файл.

Класс *Node*:

Для реализации узлов дерева создан класс *Node*, приватными полями которого являются указатели на левое и правое поддеревья, а также символ *data*, хранящий данные узла.

Конструктору класса *Node* передаются указатели на левое и правое поддеревья, а также данные, хранящиеся в узле.

Помимо этого в данном классе перегружен оператор << для вывода экземпляров класса.

Класс *Tree*:

В классе *Tree* определено приватное поле *root* (указатель на корень всего дерева).

В конструкторе этого класса удаляются внешние скобки из строки и вызывается рекурсивная функция *createNode()*. В функции *createNode()* входная строка разделяется на две части относительно знака, после чего возвращается новый экземпляр класса *Node*.

В деструкторе вызывается рекурсивная функция *deleteNode()*. В случае, если указатели на левое и правое поддеревья не *nullptr*, то вызывается эта же функция для них, после чего удаляется сам узел.

Перегруженный оператора << вызывает перегруженный оператор << для корня дерева.

Метод *transform()* вызывает рекурсивную функцию *transformNode()*, которая при определённых условиях (данный узел содержит знак *'+'*, оба сына данного узла хранят *'*'*, а два левых или два правых сына этих узлов равны) видоизменяет дерево.

Для сравнения двух поддеревьев используется функция *cmp()*, рекурсивно проверяющая, являются ли поддеревья идентичными.

Разработанный программный код см. в приложении А.

Тестирование.

Запуск программы начинается с ввода команды "make", что приведёт к компиляции и линковки программы, после чего будет создан исполняемый файл lab3. Запуск программы возможен тремя способами:

– командой “./lab3”, после запуска которой необходимо ввести выражение; результат будет напечатан на экране;
– “./lab3 <input.txt>”, где input.txt – входной файл; результат будет напечатан на экране;
– “./lab3 <input.txt> <output.txt>”, где input.txt – входной файл, а output.txt – выходной, в который будет помещён результат.

Тестирование программы производится с помощью скрипта script.py, написанного на языке программирования Python3. Запуск скрипта осуществляется командой “python3 script.py”.

Ниже представлена визуализация дерева, при входной строке *“((5*(q+7))*(3-t))”* (рисунок 1).

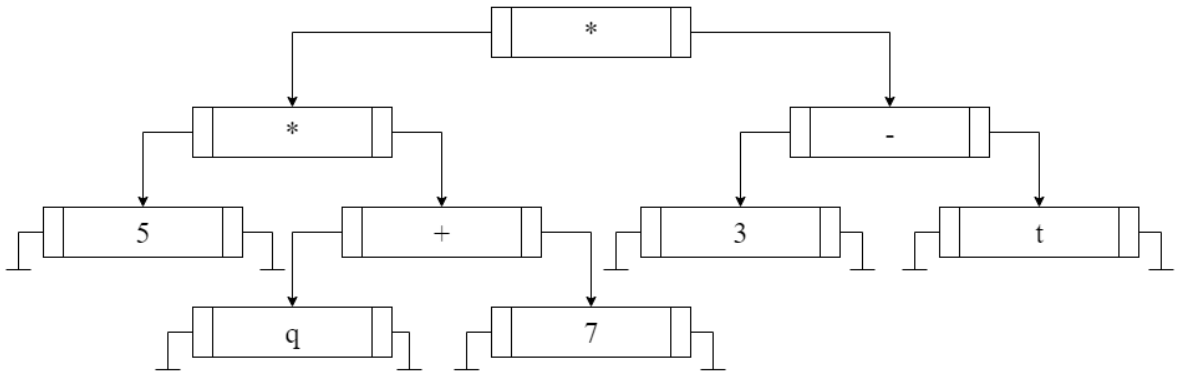

Рисунок 1 – Визуализация дерева

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	((q * t) + (q * 7))	(q*(t+7))	
2.	((q * t) + (7 * t))	((q+7)*t)	
3.	((((q * t) + (q * 7)) * p) + (((q * t) + (q * 7)) * 0))	((q*(t+7))*(p+0))	
4.	((6 * ((q * t) + (7 * t))) + (p * ((q * t) + (7 * t))))	((6+p)*((q+7)*t))	
5.	((2 * 7) + (2 * 7))	(2*(7+7))	
6.	(3 * (t + 1)))	Error: expression is incorrect	

Выводы.

Было проведено ознакомление с понятиями дерева, леса и бинарного дерева, был реализован класс бинарного дерева с помощью языка программирования C++.

Была разработана программа, создающая и обрабатывающая бинарное дерево. Использование дерева в поставленной задачи оправдано, ввиду простой реализации.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "tree.h"

int main(int argc, char *argv[]) {
    std::string formula;
    if (argc < 2) {
        std::getline(std::cin, formula);
    } else {
        std::string inFile = argv[1];
        std::ifstream in(inFile);
        if (!in.is_open()) {
            std::cerr << "Error: input file can't be open\n";
            return 0;
        }
        std::getline(in, formula);
        in.close();
    }

    for (int i = 0; i < formula.size(); i++)
        if (formula[i] == ' ') {
            formula.erase(i, 1);
            i--;
        }
    if (!Tree::checkString(formula)) {
        std::cerr << "Error: expression is incorrect\n";
        return 0;
    }

    Tree tree(formula);
    tree.transform();
    if (argc < 3) {
        std::cout << tree << '\n';
    } else {
        std::string outFile = argv[2];
        std::ofstream out(outFile);
        if (!out.is_open()) {
            std::cerr << "Error: input file can't be open\n";
            return 0;
        }
        out << tree << '\n';
        out.close();
    }

    return 0;
}
```

Название файла: node.h

```
#ifndef NODE_H
```

```
#define NODE_H

#include <iostream>

class Node {
public:
    Node (Node *, Node *, char);
    friend std::ostream &operator<< (std::ostream &, const Node *);

private:
    Node *left, *right;
    char data;
    friend class Tree;

};

#endif //NODE_H
```

Название файла: node.cpp

```
#include "node.h"

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

std::ostream &operator<< (std::ostream &out, const Node *cur) {
    if (cur->left)
        out << "(" << cur->left << cur->data << cur->right << ")";
    else
        out << cur->data;
    return out;
}
```

Название файла: tree.h

```
#ifndef TREE_H
#define TREE_H

#include <iostream>
#include <string>
#include "node.h"

class Tree {
public:
    Tree (std::string &);
    ~Tree ();
    friend std::ostream &operator<< (std::ostream &, const Tree &);
    void transform ();
    static bool checkString (std::string);
    static bool recursionCheck (std::string);

private:
    Node *createNode (std::string);
    void deleteNode (Node *);
    void transformNode (Node *);
    bool cmp (Node *, Node *);
```

```
    Node *root;

};

#endif //TREE_H
```

Название файла: tree.cpp

```
#include "tree.h"

Tree::Tree (std::string &formula) {
    formula.erase(formula.size() - 1, 1);
    formula.erase(0, 1);
    this->root = createNode(formula);
}

Tree::~Tree () {
    deleteNode(this->root);
}

std::ostream &operator<< (std::ostream &out, const Tree &cur) {
    out << cur.root;
    return out;
}

void Tree::transform () {
    transformNode(this->root);
}

bool Tree::checkString (std::string str) {
    if (str[0] != '(' || str[str.size() - 1] != ')')
        return false;
    return Tree::recursionCheck(str);
}

bool Tree::recursionCheck (std::string str) {
    if (str.size() < 5)
        return false;

    int pos1, pos2;
    if (str[1] == '(') {
        pos1 = 1, pos2;
        int counter = 0;
        for (pos2 = pos1; pos2 < str.size(); pos2++) {
            if (str[pos2] == '(')
                counter++;
            if (str[pos2] == ')')
                counter--;
            if (counter == 0)
                break;
        }
        if (recursionCheck(str.substr(pos1, pos2)))
            str.erase(pos1, pos2);
        else
            return false;
    } else {
```

```
            if ((str[1] <= '9' && str[1] >= '0') || (str[1] <= 'Z' &&
str[1] >= 'A') || (str[1] <= 'z' && str[1] >= 'a'))
                str.erase(1, 1);
            else
                return false;
        }

        if (str[1] == '+' || str[1] == '-' || str[1] == '*')
            str.erase(1, 1);
        else
            return false;

        if (str[1] == '(') {
            pos1 = 1, pos2;
            int counter = 0;
            for (pos2 = pos1; pos2 < str.size(); pos2++) {
                if (str[pos2] == '(')
                    counter++;
                if (str[pos2] == ')')
                    counter--;
                if (counter == 0)
                    break;
            }
            if (recursionCheck(str.substr(pos1, pos2)))
                str.erase(pos1, pos2);
            else
                return false;
        } else {
            if ((str[1] <= '9' && str[1] >= '0') || (str[1] <= 'Z' &&
str[1] >= 'A') || (str[1] <= 'z' && str[1] >= 'a'))
                str.erase(1, 1);
            else
                return false;
        }

        if (str == "()")
            return true;
        else
            return false;

    }

    Node *Tree::createNode (std::string expr) {
        if (expr.size() == 1)
            return new Node(nullptr, nullptr, expr[0]);

        std::string left, right;
        char sign;
        int pos = 0;
        if (expr[0] == '(') {
            int count = 0;
            for (pos; pos < expr.size(); pos++) {
                if (expr[pos] == '(')
                    count++;
                if (expr[pos] == ')')
                    count--;
                if (count == 0)
                    break;
```

```
            }
            left = expr.substr(1, pos);
        } else {
            left = expr.substr(0, 1);
        }
        sign = expr[++pos];
        if (expr[pos + 1] == '(') {
            right = expr.substr(pos + 2, expr.size() - pos - 3);
        } else {
            right = expr.substr(pos + 1, 1);
        }

        return new Node(createNode(left), createNode(right), sign);
    }

    void Tree::deleteNode (Node *cur) {
        if (cur->left)
            deleteNode(cur->left);
        if (cur->right)
            deleteNode(cur->right);
        delete cur;
    }

    void Tree::transformNode (Node *cur) {
        if (cur->left)
            transformNode(cur->left);
        if (cur->right)
            transformNode(cur->right);

        if (cur->data == '+' && cur->left->data == '*' &&
cur->right->data == '*') {
            if (cmp(cur->left->left, cur->right->left)) {
                deleteNode(cur->right->left);
                cur->right->left = cur->left->right;
                Node *temp = cur->left->left;
                cur->left->left = nullptr;
                cur->left->right = nullptr;
                deleteNode(cur->left);
                cur->left = temp;
                cur->data = '*';
                cur->right->data = '+';
            } else if (cmp(cur->right->right, cur->left->right)) {
                deleteNode(cur->left->right);
                cur->left->right = cur->right->left;
                Node *temp = cur->right->right;
                cur->right->left = nullptr;
                cur->right->right = nullptr;
                deleteNode(cur->right);
                cur->right = temp;
                cur->data = '*';
                cur->left->data = '+';
            }
        }
    }

    bool Tree::cmp (Node *node1, Node *node2) {
        if (node1->data != node2->data)
            return false;
```

```
        bool flag = true;
        if (node1->left && node2->left)
            flag = flag && cmp(node1->left, node2->left);
        if    ((!node1->left    &&    node2->left)    ||    (node1->left
&& !node2->left))
            return false;
        if (node1->right && node2->right)
            flag = flag && cmp(node1->right, node2->right);
        if    ((!node1->right    &&    node2->right)    ||    (node1->right
&& !node2->right))
            return false;

        return flag;
    }
```