МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ по

лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студентгр.8381 Перелыгин Д.С.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2019

Вариант 10-д

Цель работы.

Изучить основные принципы работы с деревьями и лесами, и принципы их обработки.

Основные теоретические сведения.

Дадим формальное определение дерева, следуя [10].

Дерево – конечное множество T, состоящее из одного или более узлов, таких, что

а) имеется один специально обозначенный узел, называемый корнем данного дерева;

б) остальные узлы (исключая корень) содержатся в m ³ 0 попарно не пересекающихся множествах T1, T2, ..., Tm, каждое из которых, в свою очередь, является деревом. Деревья T1, T2, ..., Tm называются поддеревьями данного дерева.

При программировании и разработке вычислительных алгоритмов удобно использовать именно такое рекурсивное определение, поскольку рекурсивность является естественной характеристикой этой структуры данных.

Каждый узел дерева является корнем некоторого поддерева. В том случае, когда множество поддеревьев такого корня пусто, этот узел называется концевым узлом, или листом. Уровень узла определяется рекурсивно следующим образом: 1) корень имеет уровень 1; 2) другие узлы имеют уровень, на единицу больший их уровня в содержащем их поддереве этого корня. Используя для уровня узла a дерева T обозначение уровень (a, T), можно записать это определение в виде

где Ti – поддерево корня дерева T, такое, что a Î Ti.

Говорят, что каждый корень является отцом корней своих поддеревьев и что последние являются сыновьями своего отца и братьями между собой. Говорят также, что узел n – предок узла m (а узел m – потомок узла n), если n – либо отец m, либо отец некоторого предка m.

Если в определении дерева существен порядок перечисления поддеревьев T1, T2, ..., Tm, то дерево называют упорядоченным и говорят о «первом» (T1), «втором» (T2) и т. д. поддеревьях данного корня. Далее будем считать, что все рассматриваемые деревья являются упорядоченными, если явно не оговорено противное. Отметим также, что в терминологии теории графов определенное ранее упорядоченное дерево более полно называлось бы «конечным ориентированным (корневым) упорядоченным деревом».

Постановка задачи.

Формулу вида

< формула > ::= < терминал > | (< формула > < знак > < формула >)

< знак > ::= + |–| *

< терминал > ::= 0 | 1 | ... | 9 | a | b | ... | z можно представить в виде бинарного дерева («дерева-формулы») с элементами типа Elem=char согласно следующим правилам:

- формула из одного терминала представляется деревом из одной вершины с этим терминалом;

с помощью построения дерева-формулы t преобразовать заданную формулу f из инфиксной формы в префиксную (перечисление узлов t в порядке КЛП) и в постфиксную (перечисление в порядке ЛПК); преобразовать дерево-формулу t, заменяя в нем все поддеревья, соответствующие формулам (f1 * (f2 + f3)) и ((f1 + f2) * f3), на поддеревья, соответствующие формулам ((f1 * f2) + (f1 * f3)) и ((f1 * f3) + (f2 * f3)).

Описание алгоритма.

Считывается выражение, после чего по его значениям рекурсивно заполняется дерево. После этого путём изменения указателей в дереве, формула меняется на указанную в задании.

Спецификация программы.

Программа предназначена для построения дерева по инфиксной форме выражения, конвертацию в префиксную и постфиксную форму, а так же сокращение по правилу.

Программа написана на языке C++. Входные данные подаются в виде строки текстового файла.

Описание функций.

1. boolis_brackets_correct(std::string&expression)

Определяет, правильно ли в строке expression расставлены скобки.

2. voidreplace_with_associative(std::strings)

Меняет дерево в соотетствии сзаданием.

3. voidaddRoots(std::shared_ptr<Branch>temp, std::strings)

Рекурсивно заполняет дерево.

Оценка сложности алгоритма.

Сложность алгоритма по времени является линейной. Во время обработки символы из строки заносятся в дерево при помощи рекурсивного алгоритма. Далее производится КЛП и ЛПК обходы дерева так же с помощью рекурсивных алгоритмов. Функция изменения дерева работает по аналогичному принципу. Она рекурсивно обходит дерево и в случае соответствия требованиям изменяет его. Так как дерево обходится 1 раз, эффективность по времени линейная.

Для каждого символа в дереве производится 4-5 затратных по времени операций (обращение к памяти и т. п.), однако в случае, если элементы дерева подвергаются изменению функцией по заданному правилу, то на каждое такое сокращение уходит 10 времязатратных операций.

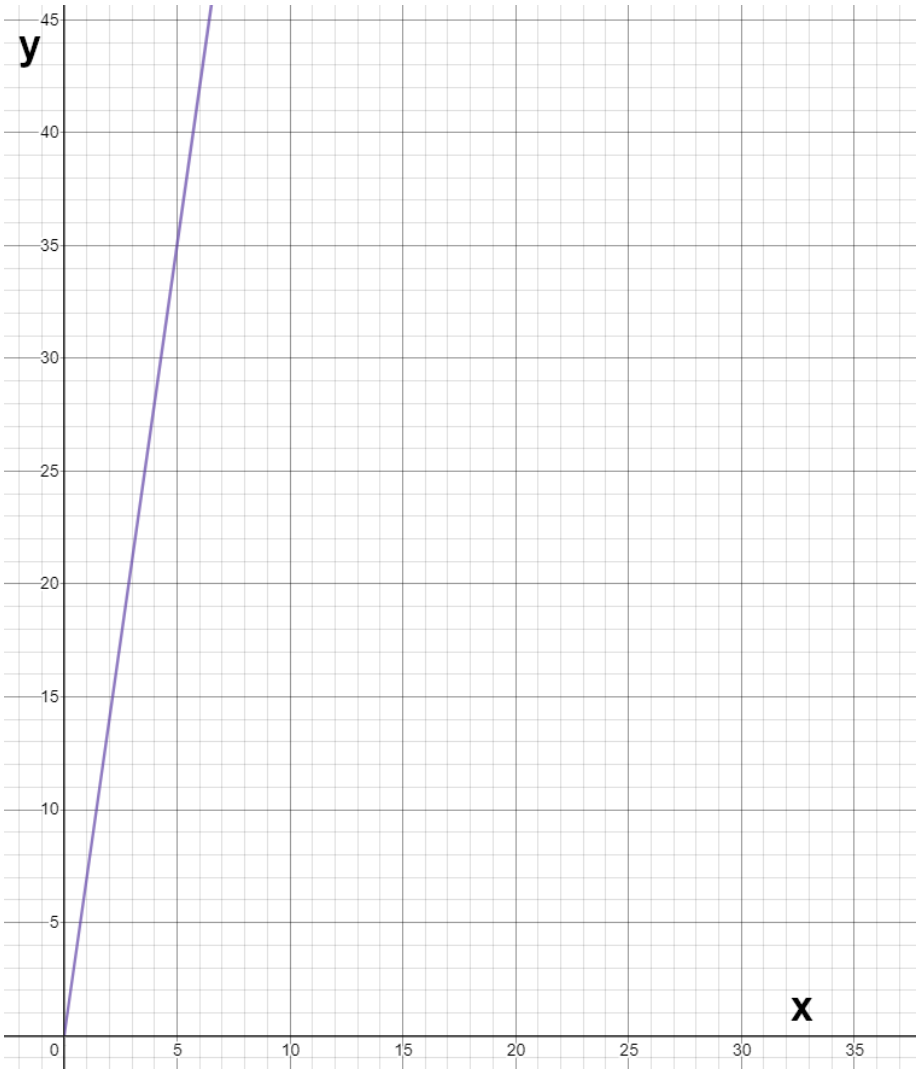

График 1 x – количество символов в входной строке, а y – количество операций.

Вывод.

Была реализована программа, позволяющая строить бинарные деревья по заданной форме, а так же упрощать их.

Приложение.

1)Тестирование.

Входные данные	Ожидаемый ответ	Ответ программы
((((A+B)*C)+ ((A+B)*C))-G)	prefix form: (- (+ (* (+ A B) C) (* (+ A B) C)) G) suffix form: ((((A B +) C *) ((A B +) C *) +) G -) expression after associative changes: ((((A * C) + (B * C)) + ((A * C) + (B * C))) - G)	prefix form: (- (+ (* (+ A B) C) (* (+ A B) C)) G) suffix form: ((((A B +) C *) ((A B +) C *) +) G -) expression after associative changes: ((((A * C) + (B * C)) + ((A * C) + (B * C))) - G)
((A+B)*C)	prefix form: (* (+ A B) C) suffix form: ((A B +) C *) expression after associative changes: ((A * C) + (B * C))	prefix form: (* (+ A B) C) suffix form: ((A B +) C *) expression after associative changes: ((A * C) + (B * C))

2)Исходныйкод.

Lab.h:

```
#pragma once
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <map>
#include <algorithm>
#include <memory>
#include <iostream>
#include <fstream>
#include <sstream>
#include <QMainWindow>
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>
#include <QFileDialog>
#include <QStandardPaths>
#include <QtGui>
#include <QLabel>
#include <QColorDialog>
#include <QInputDialog>
#include <QMainWindow>
#include <QPushButton>
#include <QMessageBox>
#include <QStringList>
#include <QTextBrowser>
#include <QTextEdit>
#include <stack>
#include "modify.h"
#include "graphic.h"

struct Branch {
    Branch() = default;

    std::string root = "0";
    std::shared_ptr<Branch> left = nullptr;
    std::shared_ptr<Branch> right = nullptr;
    int deep = 0;

};

class BinTree {
public:
    BinTree() {
        head = std::make_shared<Branch>();
    }

    bool iscorrect(std::string& s) {
        if ((s[0] != '(' || s[s.length() - 1] != ')') && s.length() != 1)
            return false;
        else return true;
    }

    bool addRoots(std::shared_ptr<Branch> temp, std::string s) {

        if (!iscorrect(s)) {
            return 0;
```

```
    }

    size_t bracket_cnter = 0;
    size_t i = 0;

    if (isalpha(s[0]) || isdigit(s[0])) {
        if (s.length() == 1) {
            temp->root = s[i];
            return 1;
        }
        else return 0;
    }

    for (size_t i = 0; i < s.length(); i++) {
        if (s[i] == '(') {
            bracket_cnter++;
        }
        else if (s[i] == ')') {
            bracket_cnter--;
        }

        if ((s[i] == '+' || s[i] == '-' || s[i] == '*') && bracket_cnter == 1) {

            temp->left = std::make_shared<Branch>();
            temp->right = std::make_shared<Branch>();

            temp->root = s[i];

            if (!addRoots(temp->left, s.substr(1, i - 1)))
                return 0;

            if (!addRoots(temp->right, s.substr(i + 1, s.length() - i - 2)))
                return 0;

            return 1;
        }
    }
    return 0;
}

int countDeep(std::shared_ptr<Branch> node)
{
    if (node == nullptr)
        return 0;
    int cl = countDeep(node->left);
    int cr = countDeep(node->right);
    return 1 + ((cl>cr)?cl:cr);
}

void pri( QGraphicsScene *&scene)
{
    head->deep = countDeep(head);
    graphic(head, scene);
}

void replace_with_associative(std::string s, QGraphicsScene *&scene) {
    if (!addRoots(head, s)) {
        std::cout << "incorrect expression form" << std::endl;
        return;
    }
```

```
        head->deep = countDeep(head);
        graphic(head, scene);
       // ui->textBrowser-
>insertPlainText(QString::fromStdString(make_prefix_expression(head)));
        std::ofstream out;          // поток для записи
        out.open("D:\\temp.txt");
        out << "prefix form:\n" << make_prefix_expression(head)  << std::endl;
        out << "suffix form:\n" << make_suffix_expression(head)  << std::endl;

        std::cout << "prefix form:\n" << make_prefix_expression(head) << std::endl;
        std::cout << "suffix form:\n" << make_suffix_expression(head) << std::endl;

        change_tree(head);
        head->deep = countDeep(head);
        std::cout << "expression after associative changes:\n" <<
make_infix_expression(head) << std::endl;

    }

    std::string make_prefix_expression(std::shared_ptr<Branch> temporary) {
        if (!temporary->left || !temporary->right) {
            return temporary->root;
        }
        return "(" + temporary->root + " " + make_prefix_expression(temporary->left) +
" " + make_prefix_expression(temporary->right) + ")";
    }

    std::string make_suffix_expression(std::shared_ptr<Branch> temporary) {
        if (!temporary->left || !temporary->right) {
            return temporary->root;
        }
        return "(" + make_suffix_expression(temporary->left) + " " +
make_suffix_expression(temporary->right) + " " + temporary->root + ")";
    }

    std::string make_infix_expression(std::shared_ptr<Branch> temporary) {
        if (!temporary->left || !temporary->right) {
            return temporary->root;
        }
        return "(" + make_infix_expression(temporary->left) + " " + temporary->root +
" " + make_infix_expression(temporary->right) + ")";
    }

    void change_tree(std::shared_ptr<Branch> temporary) {
        if (!temporary->left || !temporary->right) {
            return;
        }

        if (temporary->root == "*") {
            if (temporary->left->root == "+") {
                Branch buffer = *temporary->left->right;
                temporary->left->right = temporary->right;
                temporary->left->root = "*";
                temporary->root = "+";
                temporary->right = std::make_shared<Branch>();
                temporary->right->left = std::make_shared<Branch>();
                temporary->right->right = std::make_shared<Branch>();
                temporary->right->root = "*";
                *temporary->right->left = buffer;
                temporary->right->right = temporary->left->right;
            }
            else if (temporary->right->root == "+") {
```

```
                Branch buffer = *temporary->right->left;
                temporary->right->left = temporary->left;
                temporary->right->root = "*";
                temporary->root = "+";
                temporary->left = std::make_shared<Branch>();
                temporary->left->right = std::make_shared<Branch>();
                temporary->left->left = std::make_shared<Branch>();
                temporary->left->root = "*";
                *temporary->left->right = buffer;
                temporary->left->left = temporary->right->left;
            }
        }

        change_tree(temporary->left);
        change_tree(temporary->right);
    }

    QGraphicsScene *graphic(std::shared_ptr<Branch> node, QGraphicsScene *&scene)
    {
        if (head == nullptr)
            return scene;
        scene->clear();
        QPen pen;
        pen.setWidth(5);
        QColor color;
        color.setRgb(192, 192, 192);
        pen.setColor(color);
        QBrush brush (color);
        QFont font;
        font.setFamily("Helvetica");
        int w_deep = static_cast<int>(pow(2, head->deep)+2);
        int h = 60;
        int w = 12;
        font.setPointSize(w);
        int width = (w*w_deep)/2;
        paint(scene, head, width/2, h, w, h, pen, brush, font, w_deep);
        return scene;
    }

    int paint(QGraphicsScene *&scene, std::shared_ptr<Branch> node, int width, int
height, int w, int h, QPen &pen, QBrush &brush, QFont &font, int depth)
    {
        if (node == nullptr)
            return 0;
        QString out;
        out = node->root[0];
        QGraphicsTextItem *elem = new QGraphicsTextItem;
        elem->setPos(width, height);
        elem->setPlainText(out);
        elem->setFont(font);
        scene->addRect(width-w/2, height, w*5/2, w*5/2, pen, brush);
        if (node->left != nullptr)
            scene->addLine(width+w/2, height+w, width-(depth/2)*w+w/2, height+h+w,
pen);
        if (node->right != nullptr)
            scene->addLine(width+w/2, height+w, width+(depth/2)*w+w/2, height+h+w,
pen);
        scene->addItem(elem);
        paint(scene, node->left, width-(depth/2)*w, height+h, w, h, pen, brush, font,
depth/2);
        paint(scene, node->right, width+(depth/2)*w, height+h, w, h, pen, brush, font,
depth/2);
        return 0;
    }
```

```
private:
    std::shared_ptr<Branch> head;
    //std::map<size_t, std::string> depth_root_map;
    size_t depth = 0;
};
```

```
                getline(file, expression);

                std::cout<<expression<<std::endl;if(is
                _brackets_correct(expression)) {

                        delete_space_symbols(expression);tree.replace_with_associative(express

                        ion);

                        std::cout<<std::endl;
                }
                elsestd::cout<<"check if the brackets are correct"<<std::endl;
        }

}

intmain(intargc,char**argv)

        {if(argc== 1) {
                console_input();
        }
        elsefile_input(argv[1]);

}
```

Lab.h:

```
#pragma once
#include<iostream>#i
nclude<fstream>#incl
ude<string>#include<
cctype>#include<map>
#include<algorithm>

structBranch{
        Branch() =default;

        std::stringroot ="0";
        std::shared_ptr<Branch> left =nullptr;
        std::shared_ptr<Branch> right =nullptr;
};


classBinTree{pu
blic:
        BinTree() {
                head=std::make_shared<Branch>();
        }

        voidaddRoots(std::shared_ptr<Branch>temp, std::strings)

                {size_tbracket_cnter = 0;
                size_ti = 0;

                if(isalpha(s[0]) || isdigit(s[0]))
                        {temp->root=s[i];
                        return;
                }

                for(size_ti = 0; i <s.length(); i++)
                        {if(s[i]=='(') {
                                bracket_cnter++;
                        }
                        else if(s[i]==')'){
```

```
                        bracket_cnter--;
                }

                if((s[i]=='+'||s[i]=='-'||s[i]=='*') && bracket_cnter == 1) {

                        temp-
                        >left=std::make_shared<Branch>();temp-
                        >right=std::make_shared<Branch>();

                        temp->root=s[i];

                        addRoots(temp->left,s.substr(1, i - 1));

                        addRoots(temp->right,s.substr(i + 1,s.length() - i - 1));

                        return;
                }
        }
}

        voidfill_map(std::shared_ptr<Branch>temporary,size_tdepth, std::map<size_t,
std::string> &depth_root_map) {

                depth++;

                if(!temporary->left || !temporary->right) {
                        if(depth_root_map.find(depth)!
                                =depth_root_map.end())depth_root_map[depth]
                                +=temporary->root;
                        elsedepth_root_map.insert(make_pair(depth,temporary-
                        >root));return;
                }

                fill_map(temporary->left,depth,depth_root_map);

                fill_map(temporary->right,depth,depth_root_map);

                if(depth_root_map.find(depth)!
                        =depth_root_map.end())depth_root_map[depth]
                        +=temporary->root;
                elsedepth_root_map.insert(make_pair(depth,temporary->root));
        }

        voidprint_tree(std::map<size_t, std::string>&depth_root_map) {
                for(autoit =depth_root_map.begin(); it!=depth_root_map.end(); it++)
                        std::cout<<it->first<<" "<<it->second<<std::endl;
        }

        voidreplace_with_associative(std::strings)
                { addRoots(head,s);

                fill_map(head, depth, depth_root_map);
                print_tree(depth_root_map);

                change_tree(head);depth_root_map.clear

                ();

                fill_map(head, depth, depth_root_map);
                print_tree(depth_root_map);

        }

        voidchange_tree(std::shared_ptr<Branch>temporary)
                {if(!temporary->left || !temporary->right) {
```

```
        return;
    }
```

```
                if(temporary->root=="*") {
                        if(temporary->left->root=="+"){
                                Branchbuffer =*temporary->left-
                                >right;temporary->left->right=temporary-
                                >right;temporary->left->root="*";
                                temporary->root="+";
                                temporary-
                                >right=std::make_shared<Branch>();temporary->right-
                                >left=std::make_shared<Branch>();temporary->right-
                                >right=std::make_shared<Branch>();temporary->right-
                                >root="*";
                                *temporary->right->left=buffer;
                                temporary->right->right=temporary->left->right;
                        }
                        else if(temporary->right->root=="+")
                                {Branchbuffer =*temporary->right-
                                >left;temporary->right->left=temporary-
                                >left;temporary->right->root="*";
                                temporary->root="+";
                                temporary-
                                >left=std::make_shared<Branch>();temporary->left-
                                >right=std::make_shared<Branch>();temporary->left-
                                >left=std::make_shared<Branch>();temporary->left-
                                >root="*";
                                *temporary->left->right=buffer;
                                temporary->left->left=temporary->right->left;
                        }
                }

                change_tree(temporary-
                >left);change_tree(temporary-
                >right);
        }

  private:
        std::shared_ptr<Branch> head;
        std::map<size_t, std::string>
        depth_root_map;size_tdepth =0;
  };
```

mainwindow.cpp:

```
  #include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <map>
#include <algorithm>
#include <memory>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cstring>
#include <QMainWindow>
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>
#include <QFileDialog>
#include <QStandardPaths>
#include <QtGui>
#include <QLabel>
```

```cpp
#include <QColorDialog>
#include <QInputDialog>
#include <QMainWindow>
#include <QPushButton>
#include <QMessageBox>
#include <QStringList>
#include <QTextEdit>
#include <stack>
#include <QTextBrowser>
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "lab.h"
bool started = true;

std::string Name;

void delete_space_symbols(std::string& expression) {
    expression.erase(std::remove_if(expression.begin(), expression.end(), &isspace),
expression.end());
}

bool is_brackets_correct(std::string& expression) {

    int brackets_cnt = 0;

    for (size_t i = 0; i < expression.length(); i++) {
        if (brackets_cnt < 0)
            return false;
        else {
            if (expression[i] == '(')
                brackets_cnt++;
            else if (expression[i] == ')')
                brackets_cnt--;
            else continue;
        }
    }
    if (brackets_cnt == 0)
        return true;
    else return false;
}

void console_input(QGraphicsScene *&scene) {
    BinTree tree;

    std::cout << "Please, enter the expression" << std::endl;
    std::string expression;

    //getline(std::cin, expression);
    expression="((A+B)*C)";
    if (is_brackets_correct(expression)) {
        delete_space_symbols(expression);

        tree.replace_with_associative(expression,scene);
    }
    else std::cout << "check if the brackets are correct" << std::endl;
}

void file_input(QGraphicsScene *&scene) {

    std::ifstream file;
```

```
    file.open(Name);
    if (!file.is_open()) {
        std::cout << "Error! File isn't open" << std::endl;
        return;
    }

    size_t i = 1;

    std::string expression;

    while (!file.eof()) {

        BinTree tree;
        getline(file, expression);
        std::cout << i << ": " << expression << std::endl;
        if (is_brackets_correct(expression)) {

            delete_space_symbols(expression);

            tree.replace_with_associative(expression,scene);
            for( ; ; )
            {
            QApplication::processEvents();
            if(!started)break;

            }
            started = true;
             tree.pri(scene);
            std::cout << std::endl;
        }
        else std::cout << "check if the brackets are correct" << std::endl;
        i++;
    }

}

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),

    //graphicsView(new QGraphicsView,
    ui(new Ui::MainWindow)
{

    ui->setupUi(this);
    scene = new QGraphicsScene;
    ui->graphicsView->setScene(scene);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{
    ui->textBrowser->clear();
    std::string text;
  //console_input(scene);

  file_input(scene);
  std::ifstream file;
```

```
  file.open("D:\\temp.txt");
  for(int i=0;i<4;i++)
  {
  std::getline(file,text);
  if (i==2)
  ui->textBrowser->insertPlainText("\n\n\n");
  ui->textBrowser->insertPlainText(QString::fromStdString(text));
  }

}

void MainWindow::on_pushButton_2_clicked()
{
    QString FileName = QFileDialog::getOpenFileName(this, "OpenDialog",
    QDir::homePath(), "*.txt;; *.*");
    Name = FileName.toStdString();
}

void MainWindow::on_pushButton_3_clicked()
{
    started=false;
}
```