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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студент гр. 8381 _____ Гречко В.Д.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

Цель работы.

Ознакомиться с основными характеристиками и особенностями такой структуры данных, как бинарное дерево, изучить особенности ее реализации на языке программирования C++. Разработать программу, использующую бинарное дерево для обработки формулы.

Задание.

Для заданного бинарного дерева b типа BT с произвольным типом элементов:

- определить максимальную глубину дерева b, т. е. число ветвей в самом длинном из путей от корня дерева до листьев;
- напечатать элементы из всех листьев дерева b;
- подсчитать число узлов на заданном уровне n дерева b (корень считать узлом 1-го уровня);

Основные теоретические положения.

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.

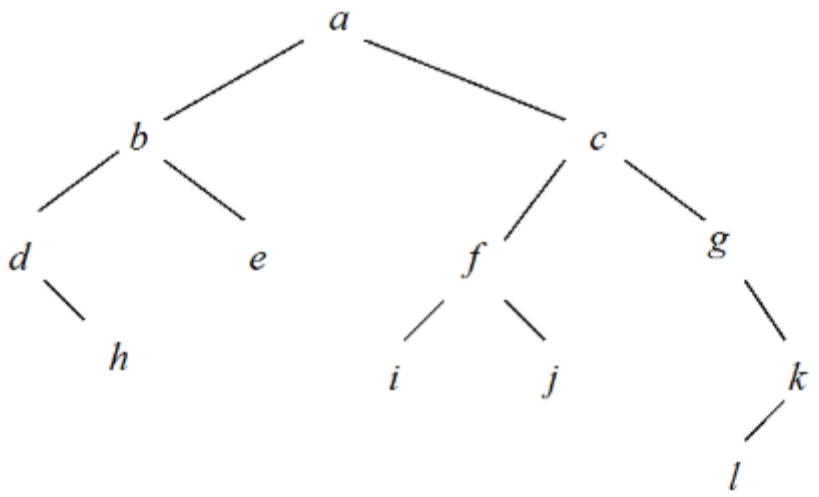


Рисунок 1 - Бинарное дерево

Бинарное дерево - конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом

Выполнение работы.

Написание работы производилось на базе операционной системы Сборка, отладка производились в QtCreator. Исходные коды файлов программы представлены в приложениях А-Ж.

Для реализации программы был разработан графический интерфейс с помощью встроенного в QtCreator UI-редактора. Он представляет из себя поле ввода, кнопку считывания, поле ввода для нахождения дополнительных значений и поле, для их вывода, а также поле вывода с возможностью графического отображения результата. Основные слоты для работы графического интерфейса приведены в табл. 1.

Таблица 1 – Слоты класса MainWindow и их назначение

Метод	Назначение

`on_onPrint_clicked()`	Слот, отвечающий за считывание данных и графического и текстового выводов
`on_onLeavesonLevel_clicked()`	Слот, отвечающий за дополнительную обработку данных

Для реализации бинарного дерева были созданы структуры узла Node и самого дерева BinTree, представленные на рис. 2.

Рисунок 2 – Структуры бинарного дерева и узла

Также были реализованы функции, создающие и изменяющие бинарное дерево, приведенные в табл. 2.

Таблица 2 – Основные функции работы с бинарным деревом

Функция	Назначение
`void BinTree()`	Создает пустое бинарное дерево
`Node* createTree(QStringList,int* count)`	Создает бинарное дерево из массива строк-элементов, полученного из входной строки
`int max_depth(Node *hd)`	Возвращает максимальную глубину дерева
`int count_node(Node *hd,int level, int curr_lev, int count)`	Возвращает количество узлов на заданном уровне

Программа имеет возможность графического отображения полученного бинарного дерева с помощью виджета QGraphicsView. Функции, необходимые для для графического представления дерева, представлены в табл. 3.

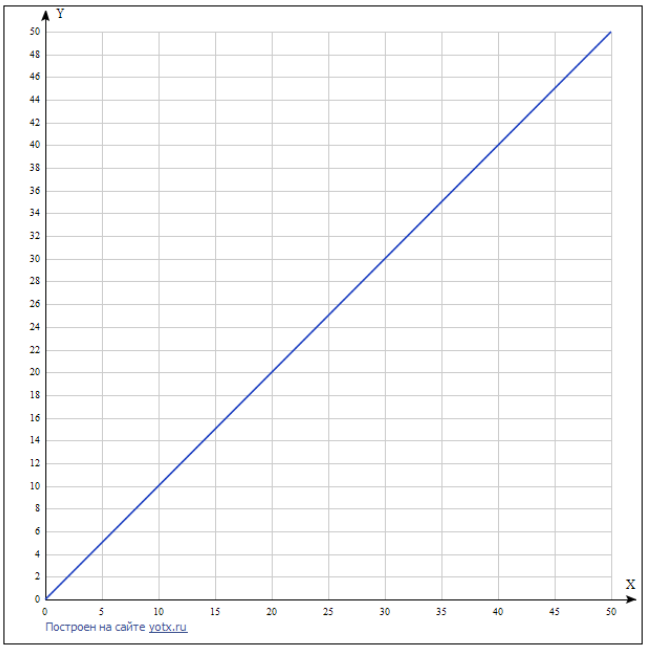
Таблица 3 – Функции, связующие графический интерфейс и алгоритмы

Функция	Назначение

`QGraphicsScene *graphic(BinTree *tree, QGraphicsScene *&scene, int depth)`	По заданному бинарному дереву выполняет рисование в объекте
`int treePainter(QGraphicsScene *&scene, Node *node, int w, int h, int wDelta, int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth)`	Рекурсивный алгоритм обхода дерева и рисования узлов в заданном объекте
`QStringList mySplit(QString rowInput)`	Обработка входной строки и преобразование её к массиву строк для дальнейшего создания бинарного дерева

Оценка сложности алгоритма.

Алгоритмы нахождения максимальной глубины и количества улов на заданном уровне являются рекурсивными, каждый узел дерева обрабатывается один раз, следовательно, сложность алгоритма $O(N)$



Тестирование программы.

Вид программы после выполнения представлен на рис. 3.

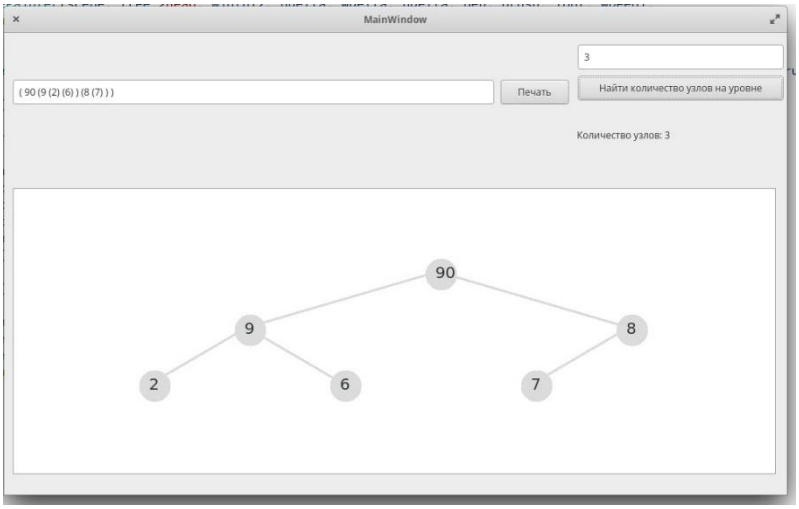


Рисунок 3 – Графический интерфейс программы

Также был рассмотрен случай некорректно введённых данных на рис. 4.

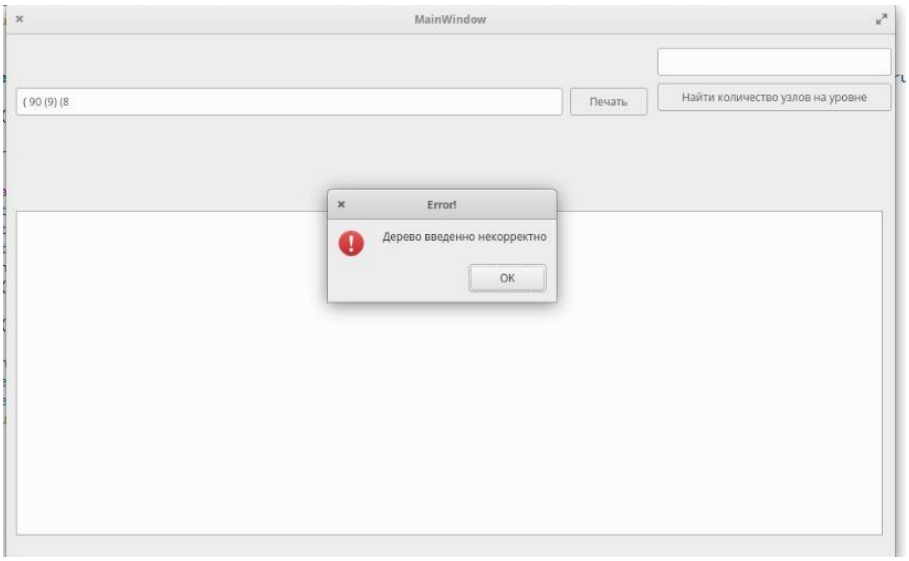


Рисунок 4 – Ошибка ввода

Был проведен ряд тестов, проверяющих корректность работы программы. Результаты тестирование приведены в табл. 4.

Таблица 4 – Тестирование программы

Входная строка	Вывод
(	Максимальная глубина: 2
(	Дерево введено некорректно
(	Количество узлов: 2
(	Количество узлов: 4

Выводы.

В ходе выполнения лабораторной работы была написана программа, создающая бинарное дерево, подсчитывающая его максимальную глубину, а также находящая количество узлов на заданном уровне. Печать бинарного дерева выполняется графически.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAIN.C

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAINWINDOW.H

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_onPrint_clicked();

    void on_onLeavesonLevel_clicked();

private:
    Ui::MainWindow *ui;
    QGraphicsScene *scene;
};

#endif // MAINWINDOW_H
```

ПРИЛОЖЕНИЕ В

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAINWINDOW.CPP

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <functionstree.h>
#include <bintree.h>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    scene = new QGraphicsScene;
    ui->graphicsView->setScene(scene);


}

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

void MainWindow::on_onPrint_clicked()
{
    QString rinput = ui->lineEdit->text();
    QString input = delSpace(rinput);
    if (input != ""){
        if (input[0] != '(' || input[input.size() - 1] != ')'){
            QMessageBox::critical(this, "Error!", "P”PµCЂPµPIPs PIPIPµPґPµPSPSPs
PSPµPєPsCЂCЂPµPєC,PSPs");
            return;
        }
    }
    else {
        QMessageBox::critical(this, "Error!", "P’PIPµPґPëC,Pµ PґPµCЂPµPIPs");
        return;
    }

    BinTree* BT = new  (BinTree);
    int* count = new int;
    *count = 0;
    BT->Head = BT->createTree(mySplit(input), count);
    QString out;
    out.append("P‟P»CґP±PëPSP° PґPµCЂPµPIP°: ");

    int depth = BT->max_depth(BT->Head);
    out.append(QString::number(depth - 1));
    out.append("    ");
    out.append(QString::number(*count));
    ui->label->setText(out);
    graphic(BT, scene,depth);


}

void MainWindow::on_onLeavesonLevel_clicked()
{
    QString rinput = ui->lineEdit->text();
```

```
    QString level_i = ui->lineEdit_2->text();
    int level = level_i.toInt();
    QString input = delSpace(rinput);
    if (input != ""){

    }
    else {
        QMessageBox::critical(this, "Error!", "P’PIPµPґPëC,Pµ PґPµCЂPµPIPs");
        return;
    }
    BinTree* BT = new  (BinTree);
   // BT->Head = BT->createTree(mySplit(input));
    QString out;
    out.append("PљPsP»PëC‡PµCЃC,PIPs CґP·P»PsPI: ");
    int count = BT->count_node(BT->Head,level, 1, 0);
    out.append(QString::number(count));
    ui->label->setText(out);

}
```

ПРИЛОЖЕНИЕ Г

ИСХОДНЫЙ КОД ПРОГРАММЫ. FUNCTIONSTREE.H

```
#ifndef FUNCTIONSTREE_H
#define FUNCTIONSTREE_H
#include <bintree.h>
#include <QMessageBox>
#include <QString>
#include <QStringList>

QString delSpace (QString rowInput);

QStringList mySplit(QString rowInput);

#endif // FUNCTIONSTREE_H
```

ПРИЛОЖЕНИЕ Д

ИСХОДНЫЙ КОД ПРОГРАММЫ. FUNCTIONSTREE.CPP

```
#include<functionstree.h>

QString delSpace (QString rowInput){
    QString out="";
    for (auto i = 0;i < rowInput.length();i++){
        if (rowInput[i]!=' ' && rowInput[i]!='\n' && rowInput[i]!='\t' )
            out.push_back(rowInput[i]);
        }
    return out;
}

QStringList mySplit(QString rowInput){
    QString st = delSpace(rowInput);
    auto i = 0;
    QStringList out={};
    QString tmp="";
    for(;i<st.length();i++){
        if (st[i] == ')' || st[i] == '(') out.push_back(QString(st[i]));
        else {
            for (; st[i]!=')' && st[i]!='(' ;i++){
                tmp.push_back(st[i]);
            }
            out.push_back(tmp);
            out.push_back(QString(st[i]));
            tmp.clear();
        }
    }
    return out;
}
```

ПРИЛОЖЕНИЕ Е

ИСХОДНЫЙ КОД ПРОГРАММЫ. BINTREE.H

```
#ifndef BINTREE_H
#define BINTREE_H
#include <QGraphicsItem>
#include <QGraphicsView>
#include <QGraphicsEffect>
#include <QString>

typedef QString type;

struct Node
{

    type data = "";
    Node* left = nullptr;
    Node* right = nullptr;

};

class BinTree
{
private:
    Node* Current = nullptr;
    QString data;
public:
    Node* Head = nullptr;
    BinTree();
    Node* createTree(QStringList,int* count);
    int max_depth(Node *hd);
    int count_node(Node *hd,int level, int curr_lev, int count);
};

QGraphicsScene *graphic(BinTree *tree, QGraphicsScene *&scene, int depth);
int treePainter(QGraphicsScene *&scene, Node *node, int w, int h, int wDelta,
int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth);



#endif // BINTREE_H
```

ПРИЛОЖЕНИЕ Ж

ИСХОДНЫЙ КОД ПРОГРАММЫ. BINTREE.CPP

```
#include<bintree.h>
#include<functionstree.h>
#include<cmath>
BinTree::BinTree(){
    Head = new Node;
    Head->data = "";
    Current = Head;
}

Node* BinTree::createTree(QStringList tokens, int* count){
    Node* finalNode = new Node;
    if(tokens.size()==2) return finalNode;
    int i =1;
    QString ltree = "";
    QString rtree = "";

    finalNode->data = tokens[i++];
    int index_i = i;    /* P□PSPrPµPєCЃ PsC,PєCЂC‹PIP°CЂC‰PµP№ CЃPєPsP±PєPё
P»PµPIPsPiPs PїPsPrPґPµCЂPµPIP° */
    *count++;
    if(tokens[i] == "("){
        auto openBrackets = 1;
        auto closeBrackets = 0;

        while (openBrackets != closeBrackets) {
            i++;
            if (tokens[i] == "("){
                openBrackets++;
            }
            else if (tokens[i] == ")"){
                closeBrackets++;
            }

        }

        for (;index_i<=i; index_i++){
            ltree.append(tokens[index_i]);
        }
        finalNode->left = createTree(mySplit(ltree), count);

        i++;

        if (tokens[i] == ")"){      /* P•CЃP»Pё PїCЂP°PIPsPiPs
PїPsPrPґPµCЂPµPIP° PSPµC, (PґPsCЃC,PёPiPSCґC, PєPsPSPµC† CЃC,CЂPsPєPё PїPsCЃP»Pµ
CЃC,CЂCґPєC,CґCЂC‹ P»PµPIPsPiPs PїPsPrPrPµCЂPµPIP°*/
            return finalNode;
        }

        int index_j = i;    /* P□PSPrPµPєCЃ PsC,PєCЂC‹PIP°CЂC‰PµP№ CЃPєPsP±PєPё
P»PµPIPsPiPs PїPsPrPrPµCЂPµPIP° */
        if(tokens[i] == "("){
            auto openBrackets = 1;
            auto closeBrackets = 0;

            while (openBrackets != closeBrackets) {
                i++;
                if (tokens[i] == "("){
```

```
                openBrackets++;
            }
            else if (tokens[i] == ")"){
                closeBrackets++;
            }

        }

        for (;index_j<=i; index_j++){
            rtree.append(tokens[index_j]);
        }
        finalNode->right = createTree(mySplit(rtree), count);
    }


    }
    return finalNode;
}
int BinTree::max_depth(Node *hd){
    if((hd == NULL) || (hd->data == '^')) return 0;
    else{
        int lDepth = max_depth(hd->left);
        int rDepth = max_depth(hd->right);

        if (lDepth > rDepth)  return(lDepth + 1);
        else return(rDepth + 1);
    }
}
int BinTree::count_node(Node *hd, int depth, int cur, int count){

    if(cur == depth){
         if(hd->data == '^') return 0;
         else return 1;
    }
    else {
       if(hd->left == nullptr && hd->right != nullptr )
            return count_node(hd->right, depth, cur + 1, count);
       else if (hd->right == nullptr && hd->left != nullptr )
            return count_node(hd->left, depth, cur + 1, count);
       else if(hd->right != nullptr && hd->left != nullptr)
        return count_node(hd->left, depth, cur + 1, count) + count_node(hd-
>right, depth, cur + 1, count);
    }

}
QGraphicsScene *graphic(BinTree *tree, QGraphicsScene *&scene, int depth)
{
    if (tree == nullptr)
        return scene;
    scene->clear();
    QPen pen;
    QColor color;
    color.setRgb(220, 220, 220);
    pen.setColor(color);
    QBrush brush (color);
    QFont font;
    font.setFamily("Tahoma");
    pen.setWidth(3);
    int wDeep = static_cast<int>(pow(2, depth + 2));
    int hDelta = 70;
    int wDelta = 15;
```

```
    font.setPointSize(wDelta);
    int width = (wDelta*wDeep)/2;
    treePainter(scene, tree->Head, width/2, hDelta, wDelta, hDelta, pen, brush,
font, wDeep);
    return scene;
}

int treePainter(QGraphicsScene *&scene, Node *node, int w, int h, int wDelta,
int hDelta, QPen &pen, QBrush &brush, QFont &font, int depth)
{
    if ((node == nullptr) || (node->data == '^'))
        return 0;
    QString out;
    out += node->data;
    QGraphicsTextItem *textItem = new QGraphicsTextItem;
    textItem->setPos(w, h);
    textItem->setPlainText(out);
    textItem->setFont(font);
    scene->addEllipse(w-wDelta/2, h, wDelta*5/2, wDelta*5/2, pen, brush);
    if ((node->left != nullptr) && (node->left->data != '^') )
        scene->addLine(w+wDelta/2, h+wDelta, w-(depth/2)*wDelta+wDelta/2,
h+hDelta+wDelta, pen);
    if (node->right != nullptr)
        scene->addLine(w+wDelta/2, h+wDelta, w+(depth/2)*wDelta+wDelta/2,
h+hDelta+wDelta, pen);
    scene->addItem(textItem);
    treePainter(scene, node->left, w-(depth/2)*wDelta, h+hDelta, wDelta, hDelta,
pen, brush, font, depth/2);
    treePainter(scene, node->right, w+(depth/2)*wDelta, h+hDelta, wDelta,
hDelta, pen, brush, font, depth/2);
    return 0;
}
```