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. 9303 _____ Максимов Е.А.

Преподаватель _____ Филатов Ар. Ю.

Санкт-Петербург
2020

Цель работы.

Познакомиться со структурами типа деревьев и способами их создания и рекурсивной обработки. Решить поставленную задачу для обработки деревьев с использованием классов, рекурсивных функций и методов.

Задание.

Вариант №12 (д) лабораторной работы.

Построить дерево-формулу *t* из строки, задающей формулу в префиксной форме (перечисление узлов *t* в порядке КЛП) с реализацией через динамическую память. Бинарные деревья должны быть реализованы как классы. Преобразовать дерево-формулу *t*, заменяя в нем все поддеревья, соответствующие формуле (*f* + *f*), на поддеревья, соответствующие формуле (*2* * *f*).

Основные теоретические положения.

Дерево – конечное множество *T*, состоящее из одного или более узлов, соответствующих следующим условиям:

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

2) остальные узлы (исключая корень) содержатся в $m \geq 0$ 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 такое рекурсивное определение, поскольку рекурсивность является естественной характеристикой этой структуры

данных.

Формулы 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:

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

2) формула вида (*f1 s f2*) представляется деревом, в котором корень – знак s, а левое и правое поддеревья – соответствующие представления формул *f1* и *f2*. Например, формула *(5*(a+3))* представляется деревом-формулой.

Выполнение работы.

В программе реализованы следующие структуры:

1. Структура *Node* представляет собой узел дерева. Структура имеет 3 поля:

a) поле *char data* представляет собой данные узла дерева, значение по умолчанию – 0;

b) поля *Tree* left* и *Tree* right* представляют собой указатели на узлы слева и справа от данного узла дерева, значения по умолчанию – *NULL.*

В программе реализован класс *Tree* – класс, содержащий структуру данных типа дерево и основные методы для его реализации.

1.1. Класс содержит приватные свойства, перечисленные ниже.

a) Свойство *Node* tree* представляет собой указатель на корень структуры. Является основным элементом реализации класса.

b) Свойство *bool errorFlag* представляет собой индикатор корректной обработки исходных данных. Определяет поведение некоторых методов класса.

1.2. Класс содержит приватные методы, перечисленные ниже.

a) Метод *bool isData(char c)* принимает на вход символ *c*. Метод возвращает *true*, если символ является символом английского алфавита или цифрой, и *false* в противном случае.

b) Метод *bool isOperator(char c)* принимает на вход символ *c*. Метод возвращает *true*, если символ является одним из символов операторов («+», «-», «*»), и *false* в противном случае.

c) Метод *void setData(Node* node, char data)* принимает на вход указатель на узел дерева *node* и данные *data* типа *char.* Метод записывает в поле *data* структуры *Node* значение из переменной *data.* Метод не имеет возвращаемого значения.

d) Метод *bool equal(Node* tree1, Node* tree2)* принимает на вход принимает на вход указатели на узлы *tree1* и *tree2*, содержащие поддеревья. Рекурсивный метод обходит поддеревья. На каждом шаге сравниваются значения узлов поддеревьев, и возвращает *false,* если значения не равны. Если поле *left* у обоих указателей не равно *NULL*, то метод рекурсивно проверяет соответствующие поддеревья из *left* и *right*, в противном случае возвращается *false.* Если рекурсивный алгоритм не вернул *false* на каком-то шаге работы метода, то метод вернёт *true*.

e) Метод *void recursionPrintDepth(char s, int n)* принимает на вход обрабатываемый символ *char s,* текущий уровень вложенности рекурсии *int n*.

Метод печатает символ со значением уровня вложенности рекурсии. Метод не имеет возвращаемого значения.

f) Метод *void recursionPrint(Node* tree, int depth = 0)* принимает на вход указатель на корень дерева *tree* и текущий уровень вложенности алгоритма, *depth,* значение по умолчанию – *0.* Метод печатает значения из дерева в нотации ЛКП (обычная запись алгебраических выражений). Метод не имеет возвращаемого значения.

g) Метод *void recursionReduce (Node* tree)* принимает на вход указатель на корень дерева *tree.* Рекурсивный метод выполняет второе задание работы. Если левое и правое поддеревья равны и в значение поля *data* соответствует символу «+», то метод записывает в узел из *left* значение «*2*», заменяет указатели на левую и правую ветвь узла из *left* на *NULL* и меняет значение поля *data* на знак «*». Метод вызывает себя для левого и правого поддерева. Метод не имеет возвращаемого значения.

h) Метод *void recursionError(int n)* принимает на вход переменную *n* типа *int* – номер ошибки, из-за которой не выполнилось одно из условий при обработке входных данных. Метод печатает на экран сообщение об ошибке и устанавливает свойство *errorFlag* на *true*. Метод не имеет возвращаемого значения.

i) Метод *void recursionConstruct(const string& line, int& I, Node* root, int depth = 0)* принимает на вход ссылку на переменную типа *string*, содержащую конфигурацию дерева, ссылку на переменную *i* типа *int* – индекс символа строки *line*, указатель на кореневой узел дерева *root* и переменную *depth* типа *int* – текущая глубина рекурсии, значение по умолчанию – 0. Рекурсивный метод выполняет первое задание работы.

Рекурсивный метод обрабатывает поддерево, которое содержится во вхождении строки *line*, начиная с символа с индексом *i.* На каждом шаге обработки метод создаёт новые объекты дерева (структура *Node*).

Метод проверяет встречающиеся символы подстроки *string*. Если текущим обрабатываемым символом является символ оператора, проверяемый методом *isOperator*, то метод рекурсивно вызывается для левой и правой ветвей дерева (соответствует полям структуры *left* и *right*), а затем устанавливает указатели в текущем узле дерева на левое и правое поддерево. Если текущим обрабатываемым символом является символ переменной, проверяемый методом *isData*, то метод создаёт элемент дерева. Проверка производится для левой и правой ветвей поддерева. В процессе обработки метод печатает символы и уровень вложенности рекурсии с помощью метода *recursionPrintDepth*.

В случае, если отсутствуют необходимые символы в строке или если строка содержит лишние или некорректные символы, метод устанавливает свойство *errorFlag* на *true* посредством метода *error*. Метод не имеет возвращаемого значения.

j) Метод *void recursionDestruct(Node* tree)* принимает на вход указатель на корень структуры дерева *tree.* Метод рекурсивно обходит узлы дерева и удаляет выделенную под них память, начиная с нижних узлов деревьев. Метод не имеет возвращаемого значения.

2.1. Класс содержит публичные методы, перечисленные ниже.

a) Конструктор класса принимает на вход ссылку на переменную *line* типа *string.* Конструктор вызывает метод *recursionConstruct*. Конструктор является входной точкой рекурсивного алгоритма. Конструктор был реализован таким образом для корректной работы с пользователем.

b) Деструктор класса не имеет входных аргументов. Деструктор вызывает метод *recursionDestruct*. Деструктор является входной точкой рекурсивного алгоритма.

c) Метод *void reduce* не имеет входных аргументов. Метод вызывает метод *recursionReduce.* Метод не имеет возвращаемого значения. Метод был

реализован таким образом для корректной работы с пользователем.

d) Метод *void print* не имеет входных аргументов. Метод вызывает метод *recursionPrint.* Метод не имеет возвращаемого значения. Метод был реализован таким образом для корректной работы с пользователем.

e) Метод *bool isValid* не имеет входных аргументов. Метод возвращает инвертированное значение приватного свойства *errorFlag*.

В программе реализованы следующие функции:

1. Функция *int error(int n)* принимает на вход переменную *n* типа *int* – номер ошибки, которая произошла при обработке аргументов из консоли. Функция печатает на экран сообщение об ошибке. Функция всегда возвращает 0. Функция была реализована для сокращения исходного кода.

2. Функция *int main(int argc, char *argv[])* создаёт объект класса *Tree*, и в случае корректной обработки исходных данных, функция печатает данные дерева с помощью методов класса. Программа напечатает на экран шаги работы программы и результат работы.

Пользователь может при запуске исполняемого файла ввести в качестве аргумента файл, в котором построчно содержатся пары иерархических списков, разделённые символом переноса строки. Если файл не пуст, то программа обработает каждую строку и запишет все шаги программы в файл «*Trees.txt*».

Исходный код программы представлен в приложении А.

Результаты тестирования программы представлены в приложении Б.

Выводы.

Была реализована программа и класс для создания и анализа структуры

типа деревьев. Класс проверяет узлы дерева в порядке КЛП на наличие выражения типа (*f*+*f*). Класс включает в себя рекурсивные функции, благодаря которым осуществляется создание и анализ.

Были выполнены следующие требования: возможность ввода данных из файла или с консоли, вывод сообщений о глубине рекурсии, вывод информации о работе программы в отдельный файл или консоль.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp.

```
#include <iostream>
#include <fstream>

using namespace std;

typedef struct Node{
    char data = 0;
    Node* left = NULL;
    Node* right = NULL;
} Node;

class Tree{
private:
    Node* tree = new Node;
    bool errorFlag = false;

    bool isData(char c){
        return ((c>=48)&&(c<=57))||((c>=65)&&(c<=90))||((c>=97)&&(
c<=122));
    }

    bool isOperator(char c){
        return ((c == '+')||(c == '-')||(c == '*'));
    }

    void setData(Node* node, char data){
        node->data = data;
        return;
    }

    bool equal(Node* tree1, Node* tree2){
        if(tree1->data != tree2->data) return false;
        if((tree1->left != NULL)&&(tree2->left != NULL)){
            if(equal(tree1->left, tree2-
>left) == false) return false;
            if(equal(tree1->right, tree2-
>right) == false) return false;
        }
        return true;
    }

    void recursionPrintDepth(char s, int n){
        for(int i=0; i<n; i++) cout << "\t";
        cout << "Symbol: " << s << "\tdepth: " << n << endl;
        return;
    }

    void recursionPrint(Node* tree, int depth = 0){
        if(errorFlag) return;
```

```
        if(tree->left != NULL){
            cout << "(";
            recursionPrint(tree->left, depth+1);
            cout << tree->data;
            recursionPrint(tree->right, depth+1);
            cout << ")";
        } else cout << tree->data;
        if(depth == 0) cout << endl;
        return;
    }

    void recursionReduce(Node* tree){
        if(errorFlag) return;
        if((tree->data == '+')&&(equal(tree->left, tree->right))){
            tree->left->data = '2';
            tree->left->left = NULL;
            tree->left->right = NULL;
            tree->data = '*';
        }
        if(tree->left != NULL) recursionReduce(tree->left);
        if(tree->right != NULL) recursionReduce(tree->right);
        return;
    }

    void recursionError(int n){
        cout << "Error!\t";
        switch(n){
            case 3: cout << "Non-
permitted symbol is string. Permitted symbols for tree are: [A-Za-
z0-9+-*]." << endl; break;
            case 4: cout << "Unexpected sequence after tree string.
" << endl; break;
            case 5: cout << "Expected more symbols in tree formula.
" << endl; break;
        };
        errorFlag = true;
        return;
    }

    void recursionConstruct(const string& line, int& i, Node* root,
 int depth = 0){
        if(errorFlag) return;
        if(i >= line.length()) return recursionError(5);
        recursionPrintDepth(line[i], depth);
        if(isOperator(line[i])){
            setData(root, line[i]);
            Node* nodeLeft = new Node;
            Node* nodeRight = new Node;
            recursionConstruct(line, ++i, nodeLeft, depth+1);
            recursionConstruct(line, ++i, nodeRight, depth+1);
            root->left = nodeLeft;
            root->right = nodeRight;
        }
        else if(isData(line[i])) setData(root, line[i]);
        else return recursionError(3);
```

```
        if((depth == 0)&&(i+1 != line.length())) return recursionE
rror(4);
        return;
    }

    void recursionDestruct(Node* tree){
        if(tree->left != NULL){
            recursionDestruct(tree->left);
            recursionDestruct(tree->right);
        }
        delete tree;
    }

public:

    Tree(const string& line){
        int i = 0;
        recursionConstruct(line, i, tree);
    }

    ~Tree(){
        recursionDestruct(tree);
    }

    void reduce(){
        recursionReduce(tree);
    }

    void print(){
        recursionPrint(tree);
    }

    bool isValid(){
        return !errorFlag;
    }

};

int error(int n){
    cout << "Error!\t";
    switch(n){
        case 0: cout << "Empty file." << endl; break;
        case 1: cout << "Incorrect input arguments. Write file pat
h or tree string." << endl; break;
        case 2: cout << "File not found." << endl; break;
    };
    return 0;
}

int main(int argc, char *argv[]){
    string line;
    switch(argc){
        case 1:
        {
            cout << "Write input tree:\t";
```

```
            getline(cin, line);
            Tree tree(line);
            if(tree.isValid()){
                cout << "Your input:\t";
                tree.print();
                tree.reduce();
                cout << "Reduced tree:\t";
                tree.print();
            }
        } break;
        case 2:
        {
            line = argv[1];
            ifstream in(line);
            if(!in) return error(2);
            ofstream out("Trees.txt");
            streambuf *coutbuf = cout.rdbuf();
            cout.rdbuf(out.rdbuf());
            int lineNum;
            for(lineNum = 1; getline(in, line); lineNum++){
                cout << "\n\tString #" << lineNum << endl;
                Tree tree(line);
                if(tree.isValid()){
                    cout << "Your input:\t";
                    tree.print();
                    tree.reduce();
                    cout << "Reduced tree:\t";
                    tree.print();
                }
            }
            cout.rdbuf(coutbuf);
            if(lineNum == 1) return error(0); else cout << "Result
s are saved in Trees.txt file." << endl;
        } break;
        default: return error(1);
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица 1 - Тестирование программы.

№	Входные данные	Выходные данные	Комментарий
1.		Error! Expected more symbols in tree formula. Error! Tree string has invalid syntax.	Тест обработки пустой строки.
2.		Symbol: depth: 0 Error! Non-permitted symbol is string. Permitted symbols for tree are: [A-Za-z0-9+-*]. Error! Tree string has invalid syntax.	Тест обработки строки с пробелом.
3.	Абвгд	Symbol: P depth: 0 Error! Non-permitted symbol is string. Permitted symbols for tree are: [A-Za-z0-9+-*]. Error! Tree string has invalid syntax.	Тест обработки некорректной строки с недопустимыми символами.
4.	333	Symbol: 3 depth: 0 Error! Unexpected sequence after tree string. Error! Tree string has invalid	Тест обработки некорректной строки.

		syntax.	
5.	+++-*	Symbol: + depth: 0 Symbol: + depth: 1 Symbol: + depth: 2 Symbol: - depth: 3 Symbol: * depth: 4 Error! Expected more symbols in tree formula. Error! Tree string has invalid syntax.	Тест обработки некорректной строки.
6.	++3	Symbol: + depth: 0 Symbol: + depth: 1 Symbol: 3 depth: 2 Error! Expected more symbols in tree formula. Error! Tree string has invalid syntax.	Тест обработки некорректной строки.
7.	+abc	Symbol: + depth: 0 Symbol: a depth: 1 Symbol: b depth: 1 Error! Unexpected sequence after tree string. Error! Tree string has invalid	Тест обработки некорректной строки.

		syntax.	
8.	3	Symbol: 3 depth: 0 Your input: 3 Reduced tree: 3	Тест обработки корректной простой строки.
9.	+de	Symbol: + depth: 0 Symbol: d depth: 1 Symbol: e depth: 1 Your input: (d+e) Reduced tree: (d+e)	Тест обработки корректной строки.
10.	+ff	Symbol: + depth: 0 Symbol: f depth: 1 Symbol: f depth: 1 Your input: (f+f) Reduced tree: (2*f)	Тест обработки корректной строки с заменой переменных в узлах дерева.
11.	+++++++12345g7h	Symbol: + depth: 0 Symbol: + depth: 1 Symbol: + depth: 2 Symbol: + depth: 3 Symbol: + depth: 4 Symbol: + depth: 5 Symbol: + depth: 6 Symbol: 1 depth: 7	Тест обработки корректной строки.

		Symbol: 2 depth: 7 Symbol: 3 depth: 6 Symbol: 4 depth: 5 Symbol: 5 depth: 4 Symbol: g depth: 3 Symbol: 7 depth: 2 Symbol: h depth: 1 Your input: (((((((1+2)+3)+4)+5)+g)+7)+h) Reduced tree: (((((((1+2)+3)+4)+5)+g)+7)+h)	
12.	+++33+33++33+33	Symbol: + depth: 0 Symbol: + depth: 1 Symbol: + depth: 2 Symbol: 3 depth: 3 Symbol: 3 depth: 3 Symbol: + depth: 2 Symbol: 3 depth: 3 Symbol: 3 depth: 3 Symbol: + depth: 1 Symbol: + depth: 2	Тест обработки корректной строки с заменой переменных в узлах дерева.

		Symbol: 3 depth: 3 Symbol: 3 depth: 3 Symbol: + depth: 2 Symbol: 3 depth: 3 Symbol: 3 depth: 3 Your input: (((3+3)+(3+3))+((3+3)+(3+3))) Reduced tree: (2*(2*(2*3)))	