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 рекурсивных функций.

Задание.

Вариант №12 лабораторной работы.

Построить дерево-формулу *t* из строки, задающей формулу в префиксной форме (перечисление узлов *t* в порядке КЛП). 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 *Tree* представляет собой элемент иерархического списка. Структура имеет 4 поля:

a) поле *char data* представляет собой данные узла дерева;

b, c) поля *Tree* left* и *Tree* right* представляют собой указатели на узлы слева и справа от данного узла дерева.

В программе реализованы следующие функции:

1. Функция *void lineToTree(string& line, vector<Elem*>& pointerCollector)* принимает на вход переменные *line* типа *string&* – ссылку на строку, содержащую дерево, и *pointerCollector* типа *vector<Elem*>* – ссылка на вектор элементов типа *Elem**. Функция обрабатывает исходную строку и создаёт дерево элементов при помощи функции *getTree*, проверяет их на корректность, находит равные ветви деревьев с помощью функции

equalSubTrees и сокращает их при помощи функции *reduceTree*, и печатает его при помощи функции *PrintTree*. В вектор *pointerCollector* записываются указатели на все созданные элементы узлов дерева для последующей очистки перед завершением работы программы. Функция не имеет возвращаемого значения. Функция была реализована для сокращения исходного кода программы.

2. Функция *Tree* getTree(string line, int& i, Tree* root, vector<Elem*> pointerCollector, int depth = 0)* принимает на вход переменную типа *string*, содержащую дерево, ссылку на переменную *i* типа *int&* – индекс символа строки *line*, указатель на кореневой узел дерева *root*, вектор указателей *pointerCollector* и переменную *depth* типа *int* – текущая глубина рекурсии, параметр умолчанию равен 0.

Рекурсивная функция обрабатывает поддерево, которое содержится во вхождении строки *line*, начиная с символа с индексом *i.* Следует отметить, что всё дерево является поддеревом, поэтому функция является точкой входа рекурсивного алгоритма. На каждом шаге обработки функция создаёт новые объекты дерева (структура *Tree*), указатели на которые помещаются в вектор *pointerCollector*.

Функция проверяет встречающиеся символы подстроки *string*. Если текущим обрабатываемым символом является символ оператора, проверяемый функцией *isOperator*, то функция рекурсивно вызывается для левой и правой ветвей дерева (соответствует полям структуры *left* и *right*). Если текущим обрабатываемым символом является символ переменной, проверяемый функцией *isData*, то функция создаёт элемент дерева. Проверка производится для левой и правой ветвей поддерева. В процессе обработки функция печатает символы и уровень вложенности рекурсии с помощью функции *recursionDepth*.

В случае, если отсутствуют необходимые символы в строке или если

строка содержит лишние или некорректные символы, функция возвращает *NULL* посредством функции *error*. В случае успеха, функция возвращает указатель на узел дерева, который был передан функции.

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

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

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

6. Функция *void recursionDepth(char s, int n)* принимает на вход обрабатываемый символ *char s,* текущий уровень вложенности рекурсии *int n*. Функция печатает символ со значением уровня вложенности рекурсии. Функция не имеет возвращаемого значения.

7. Функция *bool equalSubTrees(Tree* tree1, Tree* tree2)* принимает на вход указатели на поддеревья *tree1* и *tree2*. Рекурсивная функция 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 функция рекурсивно проверяет соответствующие поддеревья из *left* и *right*, в противном случае возвращается *false.* Если рекурсивный алгоритм не вернул *false* на каком-то шаге работы функции, то функция вернёт *true*.

8. Функция *void reduceTree (Tree* tree)* принимает на вход указатель на корень дерева *tree.* Рекурсивная функция выполняет второе задание работы. Если левое и правое поддеревья равны и в значение поля *data* соответствует

символу «+», то функция 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 «*». Функция вызывает себя для левого и правого поддерева. Функция не имеет возвращаемого значения.

9. Функция *void printTree(Tree* tree)* принимает на вход указатель на корень дерева *tree.* Функция печатает значения из дерева в нотации ЛКП (обычная запись алгебраических выражений). Функция не имеет возвращаемого значения.

10. Функция *Tree* error(int n)* принимает на вход переменную *n* типа *int* – номер ошибки, из-за которой не выполнилось одно из условий при проверке в функциях, описанных выше. Функция печатает на экран сообщение об ошибке. Функция всегда возвращает *NULL.* Функция реализована для сокращения исходного кода программы.

11. Функция *int main(int argc, char *argv[])* вызывает функцию *lineToTree* и обрабатывает входные данные. После запуска программы пользователю предлагается ввести выражение, и программа напечатает на экран шаги алгоритма 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 для создания и анализа структуры типа

деревьев. Программа проверяет узлы дерева в порядке КЛП на наличие выражения типа (*f*+*f*). Программа 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>
#include <vector>

using namespace std;

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

void lineToTree(string& line, vector<Tree*>& pointerCollector);
Tree* getTree(string& line, int& i, Tree* root, vector<Tree*>& poi
nterCollector, int depth = 0);
bool isOperator(char c);
bool isData(char c);
void setData(Tree* node, char c);
void recursionDepth(char s, int n);
Tree* error(int n);
void printTree(Tree* tree);
void reduceTree(Tree* tree);
bool equalSubTrees(Tree* tree1, Tree* tree2);

int main(int argc, char *argv[]){
    string line;
    vector<Tree*> pointerCollector;
    if(argc==1){
        cout << "Write input tree:\t";
        getline(cin, line);
        lineToTree(line, pointerCollector);
    }else if(argc==2){
        line = argv[1];
        ifstream in(line);
        if(!in){
            error(2);
            return 0;
        }
        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;
            lineToTree(line, pointerCollector);
        }
        cout.rdbuf(coutbuf);
        if(lineNum == 1) error(0); else cout << "Results are saved
 in Trees.txt file." << endl;
    } else error(1);
    for(int i=0; i<pointerCollector.size(); i++) delete pointerCol
lector[i];
    return 0;
```

```
}

void lineToTree(string& line, vector<Tree*>& pointerCollector){
    int i = 0;
    Tree* tree = new Tree;
    pointerCollector.push_back(tree);
    if(getTree(line, i, tree, pointerCollector) == NULL){
        error(5);
        return;
    }
    cout << "Your input:\t";
    printTree(tree);
    cout << endl;
    reduceTree(tree);
    cout << "Reduced tree:\t";
    printTree(tree);
    cout << endl;
    return;
}

Tree* getTree(string& line, int& i, Tree* root, vector<Tree*>& poi
nterCollector, int depth){
    if(i >= line.length()) return error(6);
    recursionDepth(line[i], depth);
    if(isOperator(line[i])){
        setData(root, line[i]);
        Tree* nodeLeft = new Tree; pointerCollector.push_back(node
Left);
        if(getTree(line, ++i, nodeLeft, pointerCollector, depth+1)
 == NULL) return NULL; else root->left = nodeLeft;
        Tree* nodeRight = new Tree; pointerCollector.push_back(nod
eRight);
        if(getTree(line, ++i, nodeRight, pointerCollector, depth+1)
 == NULL) return NULL; else root->right = nodeRight;
    } else if(isData(line[i])) setData(root, line[i]);
    else return error(3);
    if((depth == 0)&&(i+1 != line.length())) return error(4); else
 return root;
}

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

bool isData(char c){
    return ((c>=48)&&(c<=57))||((c>=65)&&(c<=90))||((c>=97)&&(c<=1
22));
}

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

void recursionDepth(char s, int n){
```

```
    for(int i=0; i<n; i++) cout << "\t";
    cout << "Symbol: " << s << "\tdepth: " << n << endl;
    return;
}

void printTree(Tree* tree){
    if(tree->left != NULL){
        cout << "(";
        printTree(tree->left);
        cout << tree->data;
        printTree(tree->right);
        cout << ")";
    } else cout << tree->data;
    return;
}

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

void reduceTree(Tree* tree){
    if((tree->data == '+')&&(equalSubTrees(tree->left, tree-
>right))){
        tree->left->data = '2';
        tree->left->left = NULL;
        tree->left->right = NULL;
        tree->data = '*';
    }
    if(tree->left != NULL) reduceTree(tree->left);
    if(tree->right != NULL) reduceTree(tree->right);
    return;
}

Tree* 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;
        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 << "Tree string has invalid syntax." << endl;
 break;
        case 6: cout << "Expected more symbols in tree formula." <
```

```
< endl; break;
    };
    return NULL;
}
```

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)))	