

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. 9382

Павлов Р.В.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель

Изучение и реализация структуры бинарного дерева, а также алгоритмов его обработки.

Основные теоретические сведения

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

$$\begin{aligned} \langle \text{БД} \rangle &::= \langle \text{пусто} \rangle \mid \langle \text{непустое БД} \rangle, \\ \langle \text{пусто} \rangle &::= \Lambda, \\ \langle \text{непустое БД} \rangle &::= (\langle \text{корень} \rangle \langle \text{БД} \rangle \langle \text{БД} \rangle). \end{aligned}$$

Например, скобочному представлению

$$(a (b (d \wedge (h \wedge \Lambda)) (e \wedge \Lambda)) (c (f(i \wedge \Lambda) (j \wedge \Lambda)) (g \wedge (k (l \wedge \Lambda) \Lambda))))$$

соответствует рис.1.

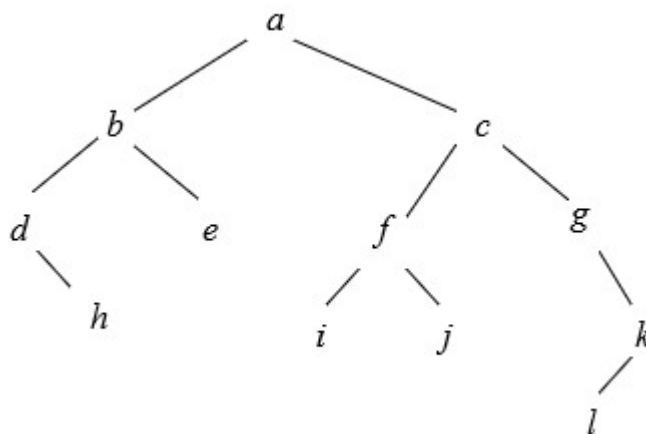


рис. 1

Задание

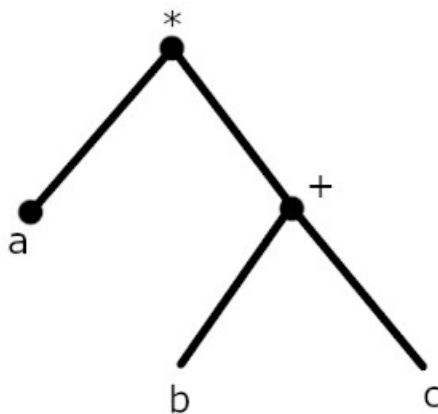
Вариант 14 (д): преобразовать дерево-формулу t , заменяя в нем все поддеревья, соответствующие формулам $(f_1 * (f_2 + f_3))$ и $((f_1 + f_2) * f_3)$, на поддеревья, соответствующие формулам $((f_1 * f_2) + (f_1 * f_3))$ и $((f_1 * f_3) + (f_2 * f_3))$.

Алгоритм

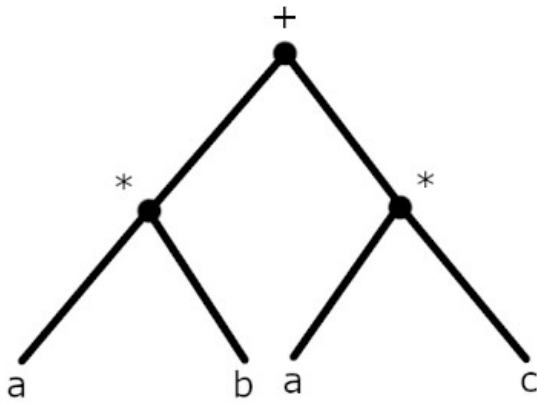
1. Дерево заполняется при ЛКП-обходе, т. е. сначала при возможности обрабатываются левые поддеревья, в содержание заносится значение символа, а затем происходит обработка правых поддеревьев. Выражения, для которых выполняется бинарная операция, и являются этими поддеревьями, а знак операции — это содержание узла. Если встречен терминальный символ, содержанием становится он, а его поддеревья пусты, и происходит возврат из рекурсии.

2. Разложение выражений на множители осуществляется посредством ЛПК-обхода, что позволяет обрабатывать дерево с листьев. В зависимости от того, где была раньше обнаружена операция сложения, один из операндов копируется и переносится в одну из частей выражения (со знаком «+»), из которой, в свою очередь, извлекается элемент, который при добавлении нового узла (и снижении одного из операндов на уровень) в ту часть, где сложение не обнаружено, крепится к этому новому узлу. Выглядит это следующим образом:

1)



2)



При переносе узлов переносятся также и их поддеревья, а при копировании создаётся новое дерево, чтобы избежать одновременной работы с несколькими поддеревьями (поскольку дерево построено на динамической памяти).

Функции и СД

- **class bTree** — класс, описывающий структуру бинарного дерева
 - **private**
 - **bTree* createTree(const string& tree, int& i, int& depth)** – создание дерева; tree – строка-выражение, i – счётчик символов, depth – глубина рекурсии; возвращаемое значение — указатель на корень дерева
 - **bTree* createNode(const char c)** – создание узла; c – содержание узла; возвращаемое значение — указатель на данный узел
 - **bTree* copyTree(bTree const* const tree)** – копирование дерева; tree – указатель на корень копируемого дерева; возвращаемое значение — указатель на корень скопированного дерева
 - **public**
 - **bTree(const string& tree, int& i, int& depth)** – конструктор дерева (вызывается только для начального узла, который не является фактическим корнем дерева); tree – строка-выражение, i – счётчик символов, depth – глубина рекурсии
 - **~bTree()** - деструктор, рекурсивно удаляющий поддеревья

- **void distribute(int& depth)** – распределение множителей; depth – глубина рекурсии
- **void printResult()** - вывод результирующего выражения на экран
- **void printResultToFile(const string filename)** – печать результирующего выражения в файл
- **void printTree(int& depth)** – вывод в виде дерева (повёрнутого на 90 градусов влево) на экран
- **inline bool isTerminal(const char c)** — проверка на терминальный символ; c – проверяемый символ; возвращаемое значение — соответствие терминальному символу
- **inline bool isSign(const char c)** - проверка на знак; c – проверяемый символ; возвращаемое значение — соответствие знаку
- **inline void avoid(const string& s, int& i)** – пропуск пробелов при анализе выражения; s – строка-выражение, i – счётчик символов
- **inline void indent(int n)** – отступ, соответствующий глубине рекурсии; n – глубина рекурсии
- **void writeToFile(const string filename, const string arg)** - запись строки-выражения в файл с указанным именем; filename – имя файла назначения, arg – записываемое выражение
- **int main():**
 - пользовательский интерфейс (выбор способа ввода выражений)
 - вывод промежуточных и итоговых результатов на экран и в файл

Тестирование

Входные данные	Выходные данные
q	q -> q
3	3 -> 3
q+3	INVALID ENTRY !
(q+3)	(q+3) -> (q+3)
(2*(q+3))	(2*(q+3)) -> ((2*q)+(2*3))
((q+3)*2)	((q+3)*2) -> ((q*2)+(3*2))
((e+5)*(9+m))	((e+5)*(9+m)) -> (((e*9)+(e*m))+((5*9)+(5*m)))
pp	INVALID ENTRY !
66	INVALID ENTRY !
((5 -g)*(t+6))	((5-g)*(t+6)) -> (((5-g)*t)+((5-g)*6))

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

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

using namespace std;

inline bool isTerminal(const char c) {
    if (c > 47 && c < 58 || c > 96 && c < 123) {
        return true;
    }
    return false;
}

inline bool isSign(const char c) {
    if (c == '+' || c == '-' || c == '*') {
        return true;
    }
    return false;
}

inline void avoid(const string& s, int& i) {
    while (s[i] == ' ' || s[i] == '\t') {
        ++i;
    }
}

inline void indent(int n) {
    for (int i = 0; i < n; i++) {
        cout << "\t";
    }
}

void writeToFile(const string filename, const string arg) {
    ofstream output;
    output.open(filename, ios::app);
    output << arg;
    output.close();
}

class bTree {
private:
    char content;
    bTree* t;
    bTree* l;
    bTree* r;

    bTree(const char c) {
        this->t = nullptr;
        this->l = nullptr;
        this->r = nullptr;
        this->content = c;
    }

    bTree* createTree(const string& tree, int& i, int& depth) {
        bTree* t = nullptr;
        if (tree.length() > 1) {
            avoid(tree, i);
            if (tree[i] == '(') {
                indent(depth);
            }
        }
    }
}
```

```

        cout << "Examining expression of level " << depth << " :
" << tree[i] << "\n";
        t = createNode(' ');
        t->l = createTree(tree, ++i, ++depth);
        avoid(tree, i);
        indent(depth);
        cout << "Examining sign " << tree[i] << "\n";
        t->content = tree[i++];
        avoid(tree, i);
        t->r = createTree(tree, i, ++depth);
        indent(depth);
        cout << "Finished examining expression of level " <<
depth << " : " << tree[i] << "\n";
        ++i;
    }
    else if (isTerminal(tree[i])) {
        indent(depth);
        cout << "Examining terminal symbol " << tree[i] << "\n";
        t = createNode(tree[i++]);
    }
}
else if (tree.length() == 1) {
    indent(depth);
    cout << "Created a tree with one node : " << tree[0] << "\n\
n";

    t = createNode(tree[0]);
}
--depth;
return t;
}
bTree* createNode(const char c) {
    bTree* node = new bTree(c);
    node->l = nullptr;
    node->r = nullptr;
    return node;
}
bTree* copyTree(bTree const* const tree) {
    bTree* t = nullptr;
    if (tree != nullptr) {
        t = createNode(tree->content);
        t->l = copyTree(tree->l);
        t->r = copyTree(tree->r);
    }
    return t;
}
public:
    bTree(const string& tree, int& i, int& depth) {
        this->t = createTree(tree, i, depth);
        this->l = nullptr;
        this->r = nullptr;
        this->content = '\0';
    }
    ~bTree() {
        if (this->t != nullptr) {
            delete this->t;
        }
        else {
            if (this->l != nullptr) {
                delete this->l;
            }
            if (this->r != nullptr) {
                delete this->r;
            }
        }
    }
}

```



```

    }
}

void distribute(int& depth) {
    if (this->t != nullptr) {
        this->t->distribute(depth);
    }
    else {
        bTree* l = this->l, * r = this->r;
        indent(depth);
        cout << "Examining symbol " << this->content << "\n";

        if (l != nullptr) {
            depth++;
            l->distribute(depth);
            depth--;
        }
        if (r != nullptr) {
            depth++;
            r->distribute(depth);
            depth--;
        }

        if (this->content == '*') {
            if (l->content == '+') {
                this->content = '+';
                l->content = '*';
                this->r = createNode('*');
                this->r->r = r;
                this->r->l = l->r;
                l->r = copyTree(r);
                indent(depth);
                cout << "Repeat distribution\n";
                this->distribute(depth);
            }
            else if (r->content == '+') {
                this->content = '+';
                r->content = '*';
                this->l = createNode('*');
                this->l->l = l;
                this->l->r = r->l;
                r->l = copyTree(l);
                indent(depth);
                cout << "Repeat distribution\n";
                this->distribute(depth);
            }
        }
    }
}

void printResult() {
    if (this->t != nullptr) {
        this->t->printResult();
    }
    else{
        if (isSign(this->content)) {
            cout << "(";
        }
        if (this->l != nullptr) {
            this->l->printResult();
        }
        cout << this->content;
        if (this->r != nullptr) {
            this->r->printResult();
        }
    }
}

```

```

        }
        if (isSign(this->content)) {
            cout << ")";
        }
    }
}

void printResultToFile(const string filename) {
    if (this->t != nullptr) {
        this->t->printResultToFile(filename);
    }
    else {
        string arg = "";
        arg += this->content;
        if (isSign(this->content)) {
            writeToFile(filename, "(");
        }
        if (this->l != nullptr) {
            this->l->printResultToFile(filename);
        }
        writeToFile(filename, arg);
        if (this->r != nullptr) {
            this->r->printResultToFile(filename);
        }
        if (isSign(this->content)) {
            writeToFile(filename, ")");
        }
    }
}

void printTree(int& depth) {
    if (this->t != nullptr) {
        this->t->printTree(depth);
    }
    else{
        if (this->r != nullptr) {
            depth++;
            this->r->printTree(depth);
            depth--;
        }
        indent(depth);
        cout << this->content << "\n";
        if (this->l != nullptr) {
            depth++;
            this->l->printTree(depth);
            depth--;
        }
    }
}

};

bool isEntryValid(const string& tree, int& i) {
    if (tree.length() == 1 && isTerminal(tree[0])) {
        return true;
    }
    avoid(tree, i);
    if (tree[i] == '(') {
        if (isEntryValid(tree, ++i)) {
            avoid(tree, i);
            if (isSign(tree[i])) {
                if (isEntryValid(tree, ++i)) {
                    avoid(tree, i);
                    if (tree[i] == ')') {
                        ++i;
                        return true;
                    }
                }
            }
        }
    }
}

```

```

    }
}

}

}
else if (i != 0 && isTerminal(tree[i++])) {
    return true;
}
return false;
}

int main() {
    int command = 0;
    cout << "Enter : 1 - Console input , 2 - File Input:\n\n";
    cin >> command;
    if (command == 2) {
        string input_filename;
        const string output_filename = "output.txt";
        ifstream in;
        ofstream out;

        out.open(output_filename);
        out << "";
        out.close();

        cout << "Enter the input file name: \n\n";
        cin >> input_filename;
        in.open(input_filename);

        if (in.is_open()) {
            string s = "";

            getline(in, s);
            do {
                cout << "\n\n" << s << "\n";
                int i = 0;
                int depth = 0;
                if (isEntryValid(s, i)) {
                    i = 0;
                    bTree* tree = new bTree(s, i, depth);
                    depth = 0;
                    tree->printResultToFile(output_filename);
                    tree->printTree(depth);
                    cout << "\n --distributing-- \n\n";
                    tree->distribute(depth);
                    writeToFile(output_filename, " -> ");
                    tree->printResultToFile(output_filename);
                    writeToFile(output_filename, "\n");
                    cout << "\nNew tree: \n\n";
                    tree->printTree(depth);
                    cout << "\n\nResult expression: \n\n";
                    tree->printResult();
                    cout << "\n\n";
                    delete tree;
                }
            } else {
                cout << "Invalid entry, the tree wasn't created\n\n";
                writeToFile(output_filename, "INVALID ENTRY !\n");
            }
            cout << " _____ \n";
            s = "";
        }
    }
}

```

```

        getline(in, s);
    } while (!in.eof());
    cout << "\nCheck out the results in \"output.txt\"\n";
}
else {
    cout << input_filename << " doesn't exist!\n";
}
}
else if (command == 1) {
    string s = "";

    cout << "\nEnter the expression (or \"!\n\" to quit): ";
    getline(cin, s);
    while (s != "!") {
        int i = 0;
        int depth = 0;
        if (isEntryValid(s, i)) {
            i = 0;
            bTree* tree = new bTree(s, i, depth);
            depth = 0;
            cout << "\n";
            tree->printTree(depth);
            cout << "\n  --distributing--  \n\n";
            tree->distribute(depth);
            cout << "\nNew tree: \n\n";
            tree->printTree(depth);
            cout << "\n\nResult expression: \n\n";
            tree->printResult();
            cout << "\n\n";
            cout << "_____ \n\n";
            delete tree;
        }
        else if (s != "") {
            cout << "Invalid entry, the tree wasn't created\n";
        }
        s = "";
        getline(cin, s);
    }
}
else {
    cout << "\nUnknown command, program finished\n";
}
return 0;
}

```