

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

Студент гр. 8381

Сергеев А.Д.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Ознакомиться с основными характеристиками и особенностями типа данных дерево, изучить особенности его реализации на языке программирования C++. Разработать программу, использующую деревья, реализованные на базе массива, изменяющую вид выражения.

Задание.

- преобразовать дерево-формулу t , заменяя в нем все поддеревья, соответствующие формулам $((f1 * f2) + (f1 * f3))$ и $((f1 * f3) + (f2 * f3))$, на поддеревья, соответствующие формулам $(f1 * (f2 + f3))$ и $((f1 + f2) * f3)$;

- с помощью построения дерева-формулы t преобразовать заданную формулу f из постфиксной формы (перечисление узлов в порядке ЛПК) в инфиксную.

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 $m \geq 0$ попарно не пересекающихся множествах T_1, T_2, \dots, T_m , каждое из которых, в свою очередь, является деревом. Деревья T_1, T_2, \dots, T_m 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.

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 .

Наиболее важным типом деревьев являются бинарные деревья. Удобно дать следующее формальное определение. Бинарное дерево - конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом.

Определим скобочное представление бинарного дерева (БД):

$$\langle \text{БД} \rangle ::= \langle \text{пусто} \rangle \mid \langle \text{непустое БД} \rangle,$$
$$\langle \text{непустое БД} \rangle ::= (\langle \text{корень} \rangle \langle \text{БД} \rangle \langle \text{БД} \rangle).$$

Выполнение работы.

Написание работы производилось на базе операционной системы Ubuntu, в среде CLion, а также с использованием библиотек qt и среды QtCreator.

Для выполнения поставленной задачи был создан класс *measuring_array*, реализующий в себе функционал массива, стека и очереди. В качестве представления дерева используется класс *static_tree*, являющийся наследником *measuring_array*. Он содержит дерево таком виде, который может быть получен при его обходе горизонтально слева направо. Пустых элементов он не содержит, так что вычисление позиции ребёнка каждого из узлов производится динамически во время обхода массива.

Класс lab4 содержит в себе алгоритм построения и изменения дерева согласно заданию.

Оценка эффективности алгоритма.

Теоретически, все методы работы с деревом имеют сложность $O(n)$ за исключением метода поиска поддерева по маске, который имеет сложность $O(n^2)$ в худшем случае (т. е. в случае, когда каждый узел дерева соответствует маске и отдельно обрабатываются поддеревья каждого узла). Поэтому в целом программа имеет сложность $O(n^2)$.

Тестирование программы.

Ниже представлен снимок экрана работающей в режиме gui программы, а также результаты трёх различных тестов в консольном и gui режиме.

Ввод «((((3)(D)*)((3)(D)*+)(D)*((((3)(D)*)((3)(D)*+)(D)*+))» (корректный ввод):

$$\begin{array}{ccccccc} & & + & & \\ & * & & * & & & \\ 3 & A & 3 & D & A & 3 & A & 3 & D \end{array}$$

$$\begin{array}{ccccc} & * & & & \\ 3 & A & D & A & D \end{array}$$

Ввод «(((A))(B)*)((C)(D)*)+)» (некорректный ввод):

The screenshot shows a web-based interface for parsing mathematical expressions. At the top, there is a text input field containing the expression `(((A))(B)*)((C)(D)*)+)`. To the right of the input field is a button labeled "Build". Below the input field, the text "String wrongly formatted!" is displayed, indicating a syntax error. At the bottom of the interface, there is another text input field containing the expression `((3 * (A + D)) * (A + D))`. To the right of this field is a button labeled "Evaluate".

Выводы.

В ходе выполнения лабораторной работы была изучена такая структура данных как дерево, а также методы его обработки. Была реализована программа на C++, использующая дерево, которая изменяет математическое выражение по условиям задания.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

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

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

    return a.exec();
}
```

Файл lab4.h:

```
//
// Created by alex on 11/7/19.
//

#ifndef LAB4_LAB4_H
#define LAB4_LAB4_H

#include <bits/stdc++.h>
#include "static_tree.h"

using namespace std;

class lab4 {
private:
    string templ;

public:
    string base;
    string state;
    static_tree<char>* tree;

    void launch(string& str);
    void rush();
    void step(int i);
};

#endif //LAB4_LAB4_H
```

Файл lab4.cpp:

```

//
// Created by alex on 11/7/19.
//

#include "lab4.h"

bool is_operator(char ch) {
    return (ch == '+') || (ch == '-') || (ch == '*');
}

bool is_operand(char ch) {
    return isalnum(ch);
}

void lab4::launch(string &str) {
    this->base = str;
    this->tree = new static_tree<char>(str, is_operator, is_operand);
    templ = "((*)(*)+)";
}

void lab4::rush() {
    auto temp = new static_tree<char>(templ, is_operator, is_operand);

    auto locator = new measuring_array<int>();
    int new_pos = tree->get_first_tree_by_template(temp, &locator);
    while (new_pos != -1) {
        auto sub_tree = tree->get_subtree(new_pos);
        auto f1 = sub_tree->get_subtree(3);
        auto f2 = sub_tree->get_subtree(4);
        auto f3 = sub_tree->get_subtree(5);
        auto f4 = sub_tree->get_subtree(6);

        if (f1->compare_to(f3)) {
            sub_tree->get(0)->setTrunk('*');
            sub_tree->get(2)->setTrunk('+');

            sub_tree->insert_tree(5, f2);
            sub_tree->insert_tree(1, f1);
        } else if (f2->compare_to(f4)) {
            sub_tree->get(0)->setTrunk('*');
            sub_tree->get(1)->setTrunk('+');

            sub_tree->insert_tree(2, f3);
        }
        tree->insert_tree(new_pos, sub_tree);

        new_pos = tree->get_first_tree_by_template(temp, &locator);
    }
}

void lab4::step(int i) {

```

```
}
```

Файл measuring_array.h:

```
#ifndef UNTITLED1_MEASURING_ARRAY_H
#define UNTITLED1_MEASURING_ARRAY_H

#include "bits/stdc++.h"

using namespace std;

#define END_SYMBOL -1
#define MIN_OCCUPATION 10

template <typename T>
class measuring_array {
private:
    int occupation;

    void checkSizeStability();

protected:
    T* array;
    int length;

public:
    measuring_array();
    virtual ~measuring_array();

    void add(T element, int pos = END_SYMBOL);
    void add_all(measuring_array<T>* other);
    void remove(int pos = END_SYMBOL);
    void replace(T element, int pos = END_SYMBOL);
    void replace(T element, T* replacer);
    T get(int pos);
    int get_length();
    bool is_empty();
    bool contains(T element);
    int find_first(T element);
    void clear();

    string to_string();
};

template<typename T>
void measuring_array<T>::add(T element, int pos) {
```



```

    if ((pos == END_SYMBOL) || (pos == length)) {
        length++;
        checkSizeStability();
        array[length - 1] = element;

    } else if ((pos >= 0) && (pos < length)) {
        length++;
        checkSizeStability();
        for (int i = length - 2; i >= pos; --i) {
            array[i+1] = array[i];
        }
        array[pos] = element;

    } else {
        throw runtime_error("Position of element inserting in measuring array is not in its
bounds");
    }
}

template<typename T>
void measuring_array<T>::add_all(measuring_array<T>* other) {
    for (int i = 0; i < other->get_length(); ++i) {
        add(other->get(i));
    }
}

template<typename T>
void measuring_array<T>::remove(int pos) {
    if ((pos == END_SYMBOL) || (pos == length)) {
        length--;
        checkSizeStability();

    } else if ((pos >= 0) && (pos < length)) {
        for (int i = pos+1; i < length; ++i) {
            array[i-1] = array[i];
        }
        length--;
        checkSizeStability();

    } else {
        throw runtime_error("Position of element removing from measuring array is not in its
bounds");
    }
}

template<typename T>
void measuring_array<T>::replace(T element, int pos) {
    if ((pos >= 0) && (pos < length)) {
        array[pos] = element;

    } else {

```

```

        throw runtime_error("Position of element replacing in measuring array is not in its
bounds");
    }
}

template<typename T>
void measuring_array<T>::replace(T element, T* replacer) {
    for (int i = 0; i < length; ++i) {
        if (element == array[i]) {
            array[i] = *replacer;
            break;
        }
    }
}

template<typename T>
void measuring_array<T>::checkSizeStability() {
    if (length < 0) throw runtime_error("Measuring array minimum size reached and
crossed!");
    if (length > occupation) {
        occupation += occupation/2 > 0 ? occupation/2 : MIN_OCCUPATION;
        array = (T*) realloc(array, (size_t) occupation * sizeof(T*));
    } else if ((length < occupation/2) && (length > MIN_OCCUPATION)) {
        occupation -= occupation/3;
        array = (T*) realloc(array, (size_t) occupation * sizeof(T*));
    }
}

template<typename T>
T measuring_array<T>::get(int pos) {
    if ((pos >= 0) && (pos < length)) {
        return array[pos];
    } else {
        throw runtime_error("Position of element getting in measuring array is not in its
bounds");
    }
}

template<typename T>
int measuring_array<T>::get_length() {
    return length;
}

template<typename T>
bool measuring_array<T>::is_empty() {
    return get_length() == 0;
}

template<typename T>
bool measuring_array<T>::contains(T element) {

```

```

    for (int i = 0; i < length; ++i) {
        if (array[i] == element) return true;
    }
    return false;
}

template<typename T>
int measuring_array<T>::find_first(T element) {
    for (int i = 0; i < length; ++i) {
        if (array[i] == element) return i;
    }
    return -1;
}

template<typename T>
void measuring_array<T>::clear() {
    int prev_len = length;
    for (int i = 0; i < prev_len; ++i) {
        remove();
    }
}

template<typename T>
string measuring_array<T>::to_string() {
    string sig;
    for (int i = 0; i < length; ++i) {
        ostringstream ss;
        ss << array[i] << " ";
        sig += ss.str();
    }
    sig += "-> " + to_string(length) + "/" + to_string(occupation);
    return sig;
}

template<typename T>
measuring_array<T>::measuring_array() {
    this->array = (T*) calloc(0, sizeof(T*));
    this->length = 0;
    this->occupation = 0;
}

template<typename T>
measuring_array<T>::~~measuring_array() {
    free(this->array);
    this->length = 0;
    this->occupation = 0;
}

```

```
#endif //UNTITLED1_MEASURING_ARRAY_H
```

Файл static_tree.h:

```
#ifndef UNTITLED1_STATIC_TREE_H  
#define UNTITLED1_STATIC_TREE_H
```

```
#include "measuring_array.h"  
#include "tree_node.h"
```

```
template<typename T>  
class static_tree : public measuring_array<tree_node<T>*> {  
private:
```

```
    const static char LOSS = '#';  
    const static char REPL = '%';
```

```
    static_tree();
```

```
    bool is_operator(char ch); // defines if the given is a supported math operator
```

```
    int getBracketPos(string& str); // returns a position between two bracket blocks in a  
string, e.g. getBracketPos("(abc)(abc)") = 4
```

```
public:
```

```
    static_tree(string& str, bool (*is_operator)(char ch), bool (*is_operand)(char ch));  
    virtual ~static_tree();
```

```
    int get_first_tree_by_template(static_tree<T>* templ, measuring_array<int>** location =  
nullptr); // get first tree matching tree template
```

```
    static_tree<T>* get_subtree(measuring_array<int>* location); // returns a tree build from  
this tree with a locations map
```

```
    static_tree<T>* get_subtree(unsigned int pos); // returns a tree build from this tree starting  
from a specific node
```

```
    bool compare_to(static_tree<T>* another); // compares tree with a tree
```

```
    void delete_subtree(unsigned int pos); // deletes tree from given element
```

```
    void insert_tree(unsigned int pos, static_tree<T>* sub_tree); // inserts an element into the  
tree
```

```
    string to_tree_string(); // prints tree graphic
```

```
    string to_infix_string(); // prints infix form of the tree
```

```
};
```

```
template<typename T>
```

```
    static_tree<T>::static_tree(string& str, bool (*is_operator)(char), bool (*is_operand)(char)) :  
measuring_array<tree_node<T>*>() {
```

```
        auto passer = new measuring_array<string*>(); // strings representing the current tree  
level
```

```
        passer->add(&str);
```

```

while (!passer->is_empty()) { // while there are strings to add to this tree level
    string curr = *(passer->get(0)); // getting new element to add to the tree
    curr = curr.substr(1, curr.size() - 2); // removing brackets

    if ((curr.size() > 1) && (is_operator(curr[curr.size() - 1]))) { // if it is not a leaf
        auto node = new tree_node<T>(curr[curr.size() - 1], false); // creating tree node
        this->add(node);

        int br_pos = getBracketPos(curr); // place where the string representations of two
branches meet
        auto child1 = new string(curr.substr(0, br_pos + 1)); // adding first branch
representation to queue
        passer->add(child1);
        auto child2 = new string(curr.substr(br_pos + 1, curr.size() - br_pos - 2)); // adding
second branch representation to queue
        passer->add(child2);

        } else if ((curr.size() == 1) && ((is_operand(curr[curr.size() - 1])) ||
(is_operator(curr[curr.size() - 1])))) { // if it is a leaf
            auto node = new tree_node<T>(curr[0], true); // creating tree node
            this->add(node);

        } else { //TODO: add other error conditions;
            throw runtime_error("String wrongly formatted!");
        }

        passer->remove(0); // removing the first, already added element from the queue
    }
}

```

```

template<typename T>
static_tree<T>::static_tree() : measuring_array<tree_node<T>*>() {}

```

```

template<typename T>
int static_tree<T>::getBracketPos(string &str) {
    int br_counter = 0; // quantity of opened brackets passed
    for (unsigned long i = 0; i < str.size(); ++i) {
        switch (str[i]) {
            case '(':
                br_counter++;
                break;
            case ')':
                br_counter--;
                if (br_counter == 0) {
                    return (int) i;
                }
                break;
        }
    }
}

```

```

    }
    throw runtime_error("String wrongly formatted!"); // if the place was not found
}

template<typename T>
bool static_tree<T>::is_operator(char ch) {
    return (ch == '+') || (ch == '-') || (ch == '*');
}

template<typename T>
static_tree<T>::~static_tree() {}

template<typename T>
int static_tree<T>::get_first_tree_by_template(static_tree<T>* templ,
measuring_array<int>** location) {
    if (location == nullptr) {
        auto lock = new measuring_array<int>; // an array containing positions of found
elements of subtree
        location = &lock;
    }
    auto inspected = new measuring_array<int>; // an array containing positions of elements
that match template root but were proofed wrong roots
    bool no_occurrences; // no roots were met

    do {
        (*location)->clear();
        int expected_child_pos = 1; // position of the first child of this node
        int templ_iterator = 0; // iterator of the template tree
        bool matches = false; // fits with template
        no_occurrences = true;

        for (int i = 0; i < this->get_length(); ++i) {
            auto node = this->get(i);

            if (no_occurrences && (node->get_trunk() == templ->get(templ_iterator)-
>get_trunk()) && (!inspected->contains(i))) { // new possible root found
                inspected->add(i); // it is added to inspected
                no_occurrences = false;
                matches = true;
                (*location)->add(i); // it is added to location
            }

            if ((*location)->contains(i)) { // new child of possible root found
                if (node->get_trunk() == templ->get(templ_iterator)->get_trunk()) { // child meets
all the requirements
                    if ((!node->is_leaf()) && (!templ->get(templ_iterator)->is_leaf())) { // both this
node and template nodes are leaves
                        int children_pos = i + expected_child_pos;
                        (*location)->add(children_pos); // expected children added to location map

```

```

        (*location)->add(children_pos + 1);
    }
    templ_iterator++;
} else {
    matches = false;
    break;
}
}

```

if (!node->is_leaf()) expected_child_pos += 2; // adding two lacunas to the map as children of a leaf

```

    expected_child_pos--;
}

if (matches) return (*location)->get(0);

} while (!no_occurrences);

(*location)->clear();
return -1;
}

```

```

template<typename T>
static_tree<T>* static_tree<T>::get_subtree(measuring_array<int>* location) {
    auto sub_tree = new static_tree<T>();
    for (int i = 0; i < location->get_length(); ++i) {
        sub_tree->add(this->get(location->get(i)));
    }
    return sub_tree;
}

```

```

template<typename T>
static_tree<T>* static_tree<T>::get_subtree(unsigned int pos) {
    auto sub_tree = new static_tree<T>();
    auto children = new measuring_array<int>();
    children->add(pos);
    int expected_child_pos = 1;

    if (pos >= this->get_length()) throw runtime_error("Subtree index not in tree!");

    for (int i = 0; i < this->get_length(); ++i) {
        tree_node<T> node = *(this->get(i));

        if (children->contains(i)) {
            sub_tree->add(this->get(i));

            if (!node.is_leaf()) {
                int children_pos = i + expected_child_pos;
                children->add(children_pos);
                children->add(children_pos + 1);
            }
        }
    }
    return sub_tree;
}

```

```

    }

    }

    if (!node.is_leaf()) expected_child_pos += 2; // adding two lacunas to the map as
children of a leaf

    expected_child_pos--;
}

return sub_tree;
}

template<typename T>
bool static_tree<T>::compare_to(static_tree<T>* another) {
    if (this->get_length() != another->get_length()) return false;
    bool same = true;

    for (int i = 0; i < this->get_length(); ++i) {
        same &= (this->get(i)->get_trunk() == another->get(i)->get_trunk());
    }

    return same;
}

template<typename T>
void static_tree<T>::delete_subtree(unsigned int pos) {
    auto marked = new measuring_array<int>(); // indexes marked for deletion
    marked->add(pos);
    int expected_child_pos = 1;

    if (pos >= this->get_length()) throw runtime_error("Deleting index not in tree!");

    for (int i = 0; i < this->get_length(); ++i) {
        tree_node<T> node = *(this->get(i));

        if (marked->contains(i)) {
            marked->remove(0);

            if (!node.is_leaf()) {
                int children_pos = i + expected_child_pos;
                marked->add(children_pos);
                marked->add(children_pos + 1);
            }

            if (i == pos) {
                auto new_node = new tree_node<T>(LOSS, true);
                this->replace(new_node, i); // first entry node is replaced with a special sign node
not to disbalance tree
            } else {
                this->remove(i);
            }
        }
    }
}

```



```

        for (int j = 0; j < marked->get_length(); ++j) marked->replace(marked->get(j) - 1,
j); // after the element was deleted all elements moved left so their deletion indexes should be
moved too

```

```

        i--; // to check this index again
    }
}

```

```

        if (!node.is_leaf()) expected_child_pos += 2; // adding two lacunas to the map as
children of a leaf

```

```

        expected_child_pos--;
    }
}

```

```

template<typename T>

```

```

void static_tree<T>::insert_tree(unsigned int pos, static_tree<T>* sub_tree) {
    auto insertion_places = new measuring_array<int>();
    insertion_places->add(pos);
    int expected_child_pos = 1;
    int sub_tree_iterator = 0;

```

```

    if (pos >= this->get_length()) throw runtime_error("Inserting index not in tree!");
    if (!this->get(pos)->is_leaf()) delete_subtree(pos); // if selected node is not a leaf all its
children will be deleted

```

```

    for (int i = 0; i < this->get_length(); ++i) {
        if (insertion_places->contains(i)) {
            insertion_places->remove(0);
            tree_node<T> node = *(sub_tree->get(sub_tree_iterator));

```

```

            if (!node.is_leaf()) {
                int children_pos = i + expected_child_pos;
                insertion_places->add(children_pos);
                insertion_places->add(children_pos + 1);
            }

```

```

            if (i == pos) {
                this->replace(sub_tree->get(sub_tree_iterator), i);
            } else {
                this->add(sub_tree->get(sub_tree_iterator), i);
            }
            sub_tree_iterator++;
        }
    }

```

```

        if (!this->get(i)->is_leaf()) expected_child_pos += 2; // adding two lacunas to the map
as children of a leaf

```

```

        expected_child_pos--;
    }
}

```

```

        for (; sub_tree_iterator < sub_tree->get_length(); ++sub_tree_iterator) { // if there are
some nodes left in the sub_tree, we can add them as-is, it is safe
            this->add(sub_tree->get(sub_tree_iterator));
        }
    }
}

```

```

template<typename T>
string static_tree<T>::to_tree_string() {
    auto arr = new measuring_array<string*>(); // array of strings representing tree levels
    auto expected_nodes = new measuring_array<bool>(); // string representing map of nodes
(with 1) and lacunas (with 0) in current and next tree level
    auto next_expected = new measuring_array<bool>();
    expected_nodes->add(true);

    arr->add(new string()); // adding new string level representation
    for (int i = 0; i < this->get_length(); ++i) {
        if (expected_nodes->is_empty()) { // the current level is empty
            expected_nodes->add_all(next_expected); // switching level maps
            next_expected->clear();
            arr->add(new string()); // adding new string level representation
        }

        if (expected_nodes->get(0)) { // if there is a real node
            tree_node<T> node = *(this->get(i)); // getting node
            ostringstream ss;
            ss << node.get_trunk(); // writing its contents to stream
            *(arr->get(arr->get_length() - 1)) += ss.str(); // adding the node value to string
representation

            if (node.is_leaf()) { // adding two lacunas to the map as children of a leaf
                next_expected->add(false);
                next_expected->add(false);
            } else { // adding two children to the map as children of a non-leaf node
                next_expected->add(true);
                next_expected->add(true);
            }
        } else {
            next_expected->add(false);
            next_expected->add(false);
            *(arr->get(arr->get_length() - 1)) += " "; // adding a long enough lacuna
representation to representation string
            i--; // "unreading" a tree node as lacuna is not a tree node
        }
    }

    expected_nodes->remove(0); // erasing first (represented) symbol from the map
    *(arr->get(arr->get_length() - 1)) += REPL; // adding a space between nodes
}

```

```

string res;
string offset, gap = " ";
for (int j = arr->get_length() - 1; j >= 0; --j) { // binding if string representations
    arr->get(j)->pop_back();

    size_t index = arr->get(j)->find(REPL);
    while (index != string::npos) {
        arr->get(j)->replace(index, 1, gap);
        index += gap.length();
        index = arr->get(j)->find(REPL, index);
    }

    if (j < arr->get_length() - 1) offset += gap.substr(0, gap.length() / 4 + 1);
    *(arr->get(j)) = offset + *(arr->get(j));

    gap += gap + " ";

    res.insert(0, *(arr->get(j)) + "\n");
}

return res;
}

template<typename T>
string static_tree<T>::to_infix_string() {
    measuring_array<string*> repr;
    string* part;
    for (int i = 0; i < this->get_length(); ++i) {
        tree_node<T> node = *(this->get(i));

        ostringstream ss;
        ss << node.get_trunk();

        if (this->get(i)->is_leaf()) {
            part = new string(ss.str());
        } else {
            part = new string("(" + string(1, REPL) + " " + ss.str() + " " + string(1, REPL) +
                ")");
        }

        repr.add(part);
    }

    for (int j = repr.get_length() - 1; j >= 0; --j) {
        for (int i = j - 1; i >= 0; --i) {
            int index = repr.get(i)->find_last_of(REPL);
            if (index != string::npos) {
                repr.get(i)->replace(index, 1, *(repr.get(j)));
                break;
            }
        }
    }
}

```

```

    }

    return *(repr.get(0));
}

#endif //UNTITLED1_STATIC_TREE_H

```

Файл **tree_node.h**:

```

#ifndef UNTITLED1_TREE_ITEM_H
#define UNTITLED1_TREE_ITEM_H

#include <ostream>

template<typename T>
class tree_node {
private:
    bool isLeaf;
    T data;

public:
    explicit tree_node(T data, bool isLeaf);
    virtual ~tree_node();

    bool is_leaf();
    T get_trunk();
    void setTrunk(T item);

    template<typename V>
    friend ostream &operator<<(ostream &os, const tree_node<V> &node);
};

template<typename T>
tree_node<T>::tree_node(T data, bool isLeaf) {
    this->isLeaf = isLeaf;
    this->data = data;
}

template<typename T>
tree_node<T>::~~tree_node() {}

template<typename T>
bool tree_node<T>::is_leaf() {
    return isLeaf;
}

```

```

}

template<typename T>
T tree_node<T>::get_trunk() {
    return data;
}

template<typename T>
void tree_node<T>::setTrunk(T item) {
    data = item;
}

template<typename T>
ostream &operator<<(ostream &os, const tree_node<T> &node) {
    return os << node.data;
}

#endif //UNTITLED1_TREE_ITEM_H

```

Файл mainwindow.h:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "lab4.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

private slots:
    void build();
    void run();

private:
    lab4* lr;
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H

```

Файл `mainwindow.cpp`:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->buildButton, SIGNAL (clicked()), this, SLOT (build()));
    connect(ui->rush_button, SIGNAL (clicked()), this, SLOT (run()));
}

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

void MainWindow::build() {
    string input = ui->input->text().toString();

    try {
        lr = new lab4();
        lr->launch(input);

        ui->tree_view->setText(QString::fromStdString(lr->tree->to_tree_string()));
        ui->answer_label->setText(QString::fromStdString(lr->tree->to_infix_string()));

        ui->input->setEnabled(false);
        ui->buildButton->setEnabled(false);
        ui->rush_button->setEnabled(true);
    } catch (runtime_error re) {
        ui->tree_view->setText(QString::fromStdString(re.what()));
    }
}

void MainWindow::run() {
    ui->input->setEnabled(true);
    ui->buildButton->setEnabled(true);
    ui->rush_button->setEnabled(false);

    try {
        lr->rush();

        ui->tree_view->setText(QString::fromStdString(lr->tree->to_tree_string()));
        ui->answer_label->setText(QString::fromStdString(lr->tree->to_infix_string()));
    } catch (runtime_error re) {
```

```

        ui->tree_view->setText(QString::fromStdString(re.what()));
    }
}

```

Файл mainwindow.ui:

```

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
    <class>MainWindow</class>
    <widget class="QMainWindow" name="MainWindow">
        <property name="geometry">
            <rect>
                <x>0</x>
                <y>0</y>
                <width>800</width>
                <height>600</height>
            </rect>
        </property>
        <property name="windowTitle">
            <string>MainWindow</string>
        </property>
        <widget class="QWidget" name="centralWidget">
            <property name="sizePolicy">
                <sizepolicy hsizeType="Expanding" vsizeType="Expanding">
                    <horstretch>0</horstretch>
                    <verstretch>0</verstretch>
                </sizepolicy>
            </property>
            <widget class="QWidget" name="verticalLayoutWidget">
                <property name="geometry">
                    <rect>
                        <x>0</x>
                        <y>0</y>
                        <width>801</width>
                        <height>601</height>
                    </rect>
                </property>
                <layout class="QVBoxLayout" name="verticalLayout">
                    <property name="leftMargin">
                        <number>5</number>
                    </property>
                    <property name="topMargin">
                        <number>5</number>
                    </property>
                    <property name="rightMargin">
                        <number>5</number>
                    </property>
                    <property name="bottomMargin">
                        <number>5</number>
                    </property>
                </layout>
            </widget>
        </widget>
    </widget>
</ui>

```

```

<layout class="QHBoxLayout" name="horizontalLayout">
  <item>
    <widget class="QLineEdit" name="input">
      <property name="font">
        <font>
          <family>Ubuntu Mono</family>
        </font>
      </property>
      <property name="placeholderText">
        <string>(((A)(B)*((A)(3)*)+)</string>
      </property>
    </widget>
  </item>
  <item>
    <widget class="QPushButton" name="buildButton">
      <property name="text">
        <string>Build</string>
      </property>
    </widget>
  </item>
</layout>
</item>
<item>
  <widget class="QLabel" name="tree_view">
    <property name="sizePolicy">
      <sizepolicy hsizeType="Expanding" vsizeType="Expanding">
        <horstretch>0</horstretch>
        <verstretch>0</verstretch>
      </sizepolicy>
    </property>
    <property name="font">
      <font>
        <family>Ubuntu Mono</family>
      </font>
    </property>
    <property name="alignment">
      <set>Qt::AlignLeading|Qt::AlignLeft|Qt::AlignVCenter</set>
    </property>
  </widget>
</item>
<item>
  <layout class="QHBoxLayout" name="horizontalLayout_2">
    <item>
      <widget class="QLabel" name="answer_label">
        <property name="sizePolicy">
          <sizepolicy hsizeType="Expanding" vsizeType="Preferred">
            <horstretch>0</horstretch>
            <verstretch>0</verstretch>
          </sizepolicy>
        </property>
        <property name="font">

```



```

    <font>
      <family>Ubuntu Mono</family>
    </font>
  </property>
</widget>
</item>
<item>
  <widget class="QPushButton" name="rush_button">
    <property name="enabled">
      <bool>>false</bool>
    </property>
    <property name="text">
      <string>Evaluate</string>
    </property>
  </widget>
</item>
</layout>
</item>
</layout>
</widget>
</widget>
</widget>

```

Файл lab3.pro:

```

#-----
#
# Project created by QtCreator 2019-11-14T01:42:01
#
#-----

```

```
QT      += core gui
```

```
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

```
TARGET = lab4
```

```
TEMPLATE = app
```

```

# The following define makes your compiler emit warnings if you use
# any feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

```

```
# You can also make your code fail to compile if you use deprecated APIs.
```

```
# In order to do so, uncomment the following line.
```

```
# You can also select to disable deprecated APIs only up to a certain version of Qt.
```

```
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the
APIs deprecated before Qt 6.0.0
```

```
SOURCES += \  
    main.cpp \  
    mainwindow.cpp \  
    lab4.cpp
```

```
HEADERS += \  
    mainwindow.h \  
    measuring_array.h \  
    static_tree.h \  
    tree_node.h \  
    lab4.h
```

```
FORMS += \  
    mainwindow.ui
```