**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 Т, 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 Т1, Т2, ..., Тm, каждое из которых, в свою очередь, является деревом. Деревья Т1, Т2, ..., Тm 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):

< БД > ::= < пусто > | < непустое БД >,

< непустое БД > ::= ( < корень > < БД > < БД > ).

## 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 в ширину, 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(n2) 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(n2).

**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):





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



## Выводы.

В ходе выполнения лабораторной работы была изучена такая структура данных как дерево, а также методы его обработки. Была реализована программа на С++, 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\_occurences; // 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\_occurences = true;

for (int i = 0; i < this->get\_length(); ++i) {

auto node = this->get(i);

if (no\_occurences && (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\_occurences = 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\_occurences);

(\*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);

}

}

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().toStdString();

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>

<item>

<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