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: Бинарные деревья

Студентка гр. 7382 _____ Павлов А.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

Задание.

Вариант 3-д. Заданы два бинарных дерева b1 и b2 типа BT с произвольным типом элементов. Проверить:

а) подобны ли они (два бинарных дерева подобны, если они оба пусты либо они оба непусты и их левые поддеревья подобны и правые поддеревья подобны);

б) равны ли они (два бинарных дерева равны, если они подобны и их соответствующие элементы равны);

в) зеркально подобны ли они (два бинарных дерева зеркально подобны, если они оба пусты либо они оба непусты и для каждого из них левое поддерево одного подобно правому поддереву другого);

г) симметричны ли они (два бинарных дерева симметричны, если они зеркально подобны и их соответствующие элементы равны).

Пояснение к заданию.

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.

Например, бинарное дерево, изображенное на рис. 1, имеет скобочное представление:

(*a* (*b* (*d* Λ (*h* Λ Λ)) (*e* Λ Λ)) (*c* (*f* (*i* Λ Λ) (*j* Λ Λ)) (*g* Λ (*k* (*l* Λ Λ) Λ)))).

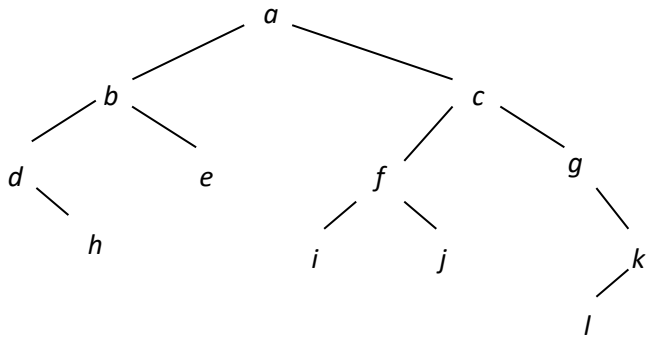

Рис. 1 – Бинарное дерево

Определим скобочное представление бинарного дерева (БД):

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

< пусто > ::= ʌ,

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

Задача каждого из пунктов сводится к рекурсивному обходу дерева, заданного скобочным выражением, и выполнение определенных действий.

Нужно написать программу, принимающую на вход два скобочных представления бинарных деревьев (в полной форме), сравнить их и сделать вывод об их подобности, равенстве, зеркальности и симметричности.

Описание алгоритма.

На подобность: сравнение начинается с корней БД. Если корни равны (они оба равны `nullptr`), то деревья подобны. Если один из корней равен `nullptr`, то деревья не подобны. В ином случае происходит рекурсивный вызов сначала для левых детей первого и второго дерева, затем для правых детей.

На зеркальность: сравнение начинается с корней БД. Если корни равны (они оба равны `nullptr`), то деревья зеркальны. Если один из корней равен `nullptr`, то деревья не зеркальны. В ином случае рекурсивно вызываем алгоритм для левого (правого) ребенка первого и правого (левого) ребенка второго.

Описание функций и структур данных.

1. Функция `void print_tabs(size_t tabs)` – функция, печатающая необходимое количество символов табуляции.

`size_t tabs` – количество выводимых символов табуляции.

Возвращаемое значение: функция ничего не возвращает.

2. `enum side_t {LEFT, RIGHT}` – перечисление, используемое для выбора стороны поддерева при создании бинарного дерева.

3. Функции `bool treatment(std::string& inputStr, int& index)` и `bool analyzer(std::string& inputStr, int& index)`

Функции принимают на вход строку `inputStr`, содержащую скобочное представление БД, переменную `index`, которая хранит текущий индекс символа входной строки. Функции проверяют его на корректность: наличие правильно расположенных скобочек, соблюдение скобочной формы, наличие элементов (или «#»).

Описание класса `binTree`.

поля:

`char value` – значение корня.

`binTree *left` – указатель на левое поддерево.

`binTree *root` – указатель на родителя.

`binTree *right` – указатель на правое поддерево.

`binTree()` – стандартный конструктор. Инициализирует указатели `left`, `right`, `root` нулевыми указателями. Используется при создании поддеревьев.

`binTree(const std::string &string, int& index)` – конструктор, передающий выходную строку в метод чтения и создания дерева.

`~binTree()` – деструктор, удаляющий узлы дерева с конца.

Методы класса.

1. `binTree * read_non_empty_binary_tree(const std::string& inputStr, int& index)`

Метод принимает на вход скобочную форму БД и индекс текущего символа. Записывает символы скобочной формы в узлы. Метод возвращает указатель на корень дерева.

2. `void binTree::read_binary_tree(const std::string& inputStr, int& index, side_t side)`

Метод, создающий поддеревья в дереве. В нем выделяется память под дерево и вызывается `read_non_empty_binary_tree` для дальнейшего создания дерева.

5. `int checkRoot(binTree * tree1, binTree * tree2)`

Метод принимает на вход 2 дерева и сравнивает их корни. Описание работы метода приведено в разделе «Описание алгоритма».

6. `bool binTree::similarityNode(binTree * tree1, binTree * tree2, size_t tabs_count)`

Метод проверяет узлы на равенство по алгоритму , описанному в разделе «Описание алгоритма».

7. `bool binTree::isMirrowNode(binTree * tree1, binTree * tree2, size_t tabs_count)`

Метод проверяет узлы левого (правого) поддерева первого дерева и узлы правого (левого) поддерева второго дерева по алгоритму, описанному в разделе описание «Описание алгоритма».

Тестирование.

Для более наглядной демонстрации работы программы был создан ряд тестов. В табл. 1 представлены входные и выходные данные всех тестов

Таблица 1 — входные и выходные данные

Входные данные	Выходные данные
(a(k##)(i##)) (a(k##)(i##))	are similar are equal are mirrored are not symmetrical
(o(u##)(v##)) (m(c##)(x##))	are similar are not equal are mirrored are not symmetrical
(j(y(w##)#)(z##)) (k(d##)(a#(b##)))	are not similar are not equal are mirrored are not symmetrical
(j(y(w##)#)(z##))	are not similar

(j(z##)(y#(w##)))	are not equal are mirrored are symmetrical
(a(b(c##)(d(e##)#))(f##)) (a(f##)(b(d#(e##))(c##)))	are not similar are not equal are mirrored are symmetrical
(k(b##)(c(d##)#)) (w(p##)(o#(e##)))	are not similar are not equal are not mirrored are not symmetrical

Разберем более подробно работу программы на примере первого теста.

Было введено: (a(k##)(i##)), (a(k##)(i##))

Для подобия и равенства: Сначала проверяются корни двух деревьев, если они оба равны `nullptr`, то эти деревья подобны и равны. Если один из них `nullptr`, то деревья не подобны и не равны. В нашем случае корни обоих деревьев не равны `nullptr`, следовательно, проверяется левое и правое поддеревья этих деревьев. Структура левого (правового) поддерева первого и второго дерева равны, следовательно, деревья подобны, а так как и значения соответствующих узлов равны, то деревья равны.

Для зеркальности и симметричности: Сначала проверяются корни двух деревьев, если они оба равны `nullptr`, то эти деревья зеркальны и симметричны. Если один из них `nullptr`, то деревья не зеркальный и не симметричны. В нашем случае корни обоих деревьев не равны `nullptr`, следовательно, сравнивается структуры левого (правого) поддерева первого дерева и правового (левого) поддерева второго дерева. Левое (правое) поддерево первого дерева и правое (левое) поддерево второго дерева имеют одинаковую структуру, значит, деревья зеркальны, а так как и значения узлов раны, то деревья симметричны.

Выводы.

В ходе работы была закреплена тема бинарных деревьев, изучено представление бинарного дерева с помощью динамической памяти, а также работа с ним.

ПРИЛОЖЕНИЕ А
КОД MAIN.CPP

```
#include <algorithm>
#include <string>
#include <iostream>
#include "BinTree.hpp"
bool analyzer(std::string& inputStr, int& index);
bool treatment(std::string& inputStr, int& index);

bool treatment(std::string& inputStr, int& index){
    inputStr.erase(std::remove_if(inputStr.begin(),     inputStr.end(),
isspace), inputStr.end());//delete spaces

    if(inputStr.empty()){
        std::cout << "Input is empty!" << std::endl;
        return false;
    }
    if(analyzer(inputStr, index)){
        if(inputStr[index+1] != '\0'){
            std::cout << "Extra characters in the input!" << std::endl;
            return false;
        }
    }
    else {
        std::cout << "Binary tree record is wrong!" <<std::endl;
        return false;
    }
    index = 0;
     return true;
}

bool analyzer(std::string& inputStr, int& index){
    if(inputStr[index] == '('){
        ++index;
        if (isprint(inputStr[index]) && inputStr[index] != '#') {
```

```
            if(analyzer(inputStr,   ++index)   &&   analyzer(inputStr,
++index)){
                if(inputStr[++index] == ')')
                    return true;
                else return false;
            }
            else return false;
        }
        else return false;
    }
    else if(inputStr[index] == '#')
        return true;
    else return false;
}

int main(){
    int index = 0;
    std::string inputStr1;
    std::string inputStr2;

    std::cout << "Enter binary tree1: mark empty as '#'" << std::endl
<< "Example: (a(b##)(c##))" << std::endl;
    std::getline(std::cin, inputStr1);
    if(!treatment(inputStr1, index))
        return 0;

    std::cout << "Enter binary tree2: mark empty as '#'" << std::endl
<< "Example: (a(b##)(c##))" << std::endl;
    std::getline(std::cin, inputStr2);
    if(!treatment(inputStr2, index))
        return 0;

    std::cout << "Binary tree1..." << std::endl;
    binTree tree1(inputStr1, index);
    tree1.print();
    index = 0;
```

```
    std::cout << std::endl << std::endl;

    std::cout << "Binary tree2..." << std::endl;
    binTree tree2(inputStr2, index);
    tree2.print();
    std::cout << std::endl << std::endl;

    std::cout << std::endl << "Cheking for similarity and equality..."
<< std::endl;
    if (binTree::similarityTree(&tree1, &tree2))
        std::cout << "Binary trees are similar" << std::endl;
    else std::cout << "Binary trees are not similar" << std::endl;

    if (binTree::equalTree(&tree1, &tree2))
        std::cout << "Binary trees are equal" << std::endl;
    else std::cout << "Binary trees are not equal" << std::endl;

    std::cout << std::endl << "Cheking for mirror similarity and
symmetry..." << std::endl;
    if (binTree::isMirrowTree(&tree1, &tree2))
        std::cout << "Binary trees are mirrored" << std::endl;
    else std::cout << "Binary trees are not mirrored" << std::endl;

    if(binTree::isMirrowEqualTree(&tree1, &tree2))
         std::cout << "Binary trees are symmetrical" << std::endl;
    else std::cout << "Binary trees are not symmetrical" << std::endl;
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

ФАЙЛ BINTREE.HPP

```
#include <iostream>

void print_tabs(size_t tabs_count);
enum side_t { LEFT, RIGHT };

void print_tabs(size_t tabs_count) {
    for (size_t i = 0; i < tabs_count; ++i)
        std::cout << '\t';
}

class binTree{
    private:
        binTree* root;
        binTree* left;
        binTree* right;
        char data;
    public:
        binTree();
        binTree(const std::string& inputStr, int& index);
        ~binTree();
        binTree * leftTree();
        binTree * rightTree();
        void destroy();
        binTree  *  read_non_empty_binary_tree(const  std::string&
inputStr, int& index);
        void  read_binary_tree(const  std::string&  inputStr,  int&
index, side_t side);
        void _print(size_t tabs_count);
        void print();
```

```
            size_t height() const;
            static int checkRoot(binTree * tree1, binTree * tree2);
            static bool similarityNode(binTree * tree1, binTree *
tree2, size_t tabs_count);
            static bool similarityTree(binTree * tree1, binTree *
tree2);
            static bool equalTree(binTree * tree1, binTree * tree2);
            static bool equalNode(binTree * tree1, binTree * tree2);
            static bool isMirrowTree(binTree * tree1, binTree * tree2);
            static bool isMirrowNode(binTree * tree1, binTree * tree2,
size_t tabs_count);
            static bool isMirrowEqualTree(binTree * tree1, binTree *
tree2);
            static bool isMirrowEqualNode(binTree * tree1, binTree *
tree2);
     };

     binTree::binTree() : root(nullptr), left(nullptr), right(nullptr)
{ }

     binTree::binTree(const std::string& inputStr, int& index) :
binTree() {
         root = read_non_empty_binary_tree(inputStr, index);
     }

     binTree *  binTree::read_non_empty_binary_tree(const std::string&
inputStr, int& index){//create tree
         if(inputStr[index] == '#')
             return nullptr;
         ++index;
         data = inputStr[index];
         left = nullptr;
          right = nullptr;
         ++index;
```

```
        read_binary_tree(inputStr, index, LEFT);
        ++index;
        read_binary_tree(inputStr, index, RIGHT);
        ++index;
        return this;
    }

    void binTree::read_binary_tree(const std::string& inputStr, int&
index, side_t side) {//create branches
        if (inputStr[index] == '#')
            return;
        else {
            if (side == LEFT) {
                left = new binTree;
                left->read_non_empty_binary_tree(inputStr, index);
            }
            else {
                right = new binTree;
                right->read_non_empty_binary_tree(inputStr, index);
            }
        }
    }

    void binTree::_print(size_t tabs_count) {//display tree
        print_tabs(tabs_count);
        std::cout << data << std::endl;
        if (right)
            right->_print(tabs_count + 1);
        else {
            print_tabs(tabs_count + 1);
            std::cout << '#'<<std::endl;
        }
        if (left)
            left->_print(tabs_count + 1);
```

```
    else {
        print_tabs(tabs_count + 1);
        std::cout << '#' <<std::endl;
    }

}

void binTree::print() {//display tree
    size_t h = height();
    if(!root){
        std:: cout << '#' << std::endl;
        return;
    }
    for (size_t i = 0; i < h + 1; ++i)
        std::cout << i << '\t';
    std::cout << std::endl;
    _print(0);
    for (size_t i = 0; i < h + 1; ++i)
        std::cout << i << '\t';
    std::cout << std::endl;
}

size_t binTree::height() const {
    size_t height_left = 0;
    if (left)
        height_left = left->height();
    size_t height_right = 0;
    if (right)
        height_right = right->height();
    return 1 + std::max(height_left, height_right);
}

int binTree::checkRoot(binTree * tree1, binTree * tree2){
    if((tree1->root) == (tree2->root)){
```

```
            std::cout << "comparing: [#]  [#]"<< std::endl;
            return 1;
        }
        else if((tree1->root == nullptr) || (tree2->root == nullptr))
            return 2;
        else return 3;
    }

    bool binTree::similarityTree(binTree * tree1, binTree * tree2){
        int result = checkRoot(tree1, tree2);
        if(result == 1)
            return true;
        else if(result == 3){
            if(binTree::similarityNode(tree1, tree2, 0))
                return true;
            return false;
        }
        else return false;
    }
    bool  binTree::similarityNode(binTree  *  tree1,  binTree  *  tree2,
size_t tabs_count){
        if(tree1 == tree2)//if tree1 = tree2 = nullptr
            return true;

        if(tree1 == nullptr){
            print_tabs(tabs_count);
            std::cout << "comparing: " <<"[#]  " << '[' << tree2->data
<< ']' << std::endl;
            return false;
        }
        if(tree2 == nullptr){
            print_tabs(tabs_count);
            std::cout << "comparing: " <<'[' << tree1->data << "]  "
<< "[#]" << std::endl;
```

```
            return false;
        }
        print_tabs(tabs_count);
        std::cout << "comparing: " <<'[' << tree1->data << "]   " <<
'[' << tree2->data << ']' << std::endl;
        return   (binTree::similarityNode(tree1->left,   tree2->left,
tabs_count+1)  &&  binTree::similarityNode(tree1->right,  tree2->right,
tabs_count+1));
    }

    bool binTree::equalTree(binTree * tree1, binTree * tree2){
        int result = checkRoot(tree1, tree2);
        if(result == 1)
            return true;
        else if(result == 3){
            if(binTree::equalNode(tree1, tree2))
                return true;
            return false;
        }
        else return false;
    }

    bool binTree::equalNode(binTree * tree1, binTree * tree2){
        if(tree1 == tree2)//if tree1 = tree2 = nullptr
            return true;
        if(tree1 == nullptr)
            return false;
        if(tree2 == nullptr)
            return false;
        if(tree1->data != tree2->data)
            return false;
        return   (binTree::equalNode(tree1->left,   tree2->left)   &&
binTree::equalNode(tree1->right, tree2->right));
    }
```

```
bool binTree::isMirrowTree(binTree * tree1, binTree * tree2){
    int result = checkRoot(tree1, tree2);
    if(result == 1)
        return true;
    else if(result == 3){
        if(binTree::isMirrowNode(tree1, tree2, 0))
            return true;
        return false;
    }
    else return false;
}

bool binTree::isMirrowNode(binTree * tree1, binTree * tree2, size_t
tabs_count){
    if(tree1 == tree2)//if tree1 = tree2 = nullptr
        return true;
    if(tree1 == nullptr){
        print_tabs(tabs_count);
        std::cout << "comparing: " <<"[#]  " << '[' << tree2->data
<< ']' << std::endl;
        return false;
    }
    if(tree2 == nullptr){
        print_tabs(tabs_count);
        std::cout << "comparing: " <<'[' << tree1->data << "]  "
<< "[#]" << std::endl;
        return false;
    }
    print_tabs(tabs_count);
    std::cout << "comparing: " <<'[' << tree1->data << "]  " <<
'[' << tree2->data << ']' << std::endl;
```

```
        return((binTree::isMirrowNode(tree1->left,      tree2->right,
tabs_count+1))  &&  (binTree::isMirrowNode(tree1->right,  tree2->left,
tabs_count+1)));
    }

    bool binTree::isMirrowEqualTree(binTree * tree1, binTree * tree2){
        int result = checkRoot(tree1, tree2);
        if(result == 1)
            return true;
        else if(result == 3){
            if(binTree::isMirrowEqualNode(tree1, tree2))
                return true;
            return false;
        }
        else return false;
    }

    bool binTree::isMirrowEqualNode(binTree * tree1, binTree * tree2){
        if(tree1 == tree2)//if tree1 = tree2 = nullptr
            return true;
        if(tree1 == nullptr)
            return false;
        if(tree2 == nullptr)
            return false;
        if(tree1->data != tree2->data)
            return false;
        return((binTree::isMirrowEqualNode(tree1->left,
tree2->right))             &&(binTree::isMirrowEqualNode(tree1->right,
tree2->left)));
    }

    binTree::~binTree(){
        if (left)
               delete left;
```

```
        if (right)
            delete right;
    }
```

ПРИЛОЖЕНИЕ В
КОД ФАЙЛА RUNTESTS.BAT

```
echo off
cl /EHsc .\Source\main.cpp
echo Test 1:
type .\Tests\Test1.txt
echo.
main < .\Tests\Test1.txt
echo.
echo Test 2:
type .\Tests\Test2.txt
echo.
main < .\Tests\Test2.txt
echo.
echo Test 3:
type .\Tests\Test3.txt
echo.
main < .\Tests\Test3.txt
echo.
echo Test 4:
type .\Tests\Test4.txt
echo.
main < .\Tests\Test4.txt
echo.
echo Test 5:
type .\Tests\Test5.txt
echo.
main < .\Tests\Test5.txt
echo.
echo Test 6:
type .\Tests\Test6.txt
echo.
main < .\Tests\Test6.txt
echo.
```