**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: Обработка бинарных деревьев и лесов.**

Студент гр. 9381 _____ Семенов А. Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2020

**Цель работы.**

Ознакомление с одними из часто используемых на практике нелинейных конструкций, способами их организации и рекурсивной обработки; получение навыков решения задач обработки бинарных деревьев и лесов, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

**Задание.**

Вариант 5д (через динамическую память на базе указателей)

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

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

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

- 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);

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

**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 >= 0 попарно не пересекающихся множествах $T_1$, $T_2$, ..., $T_m$, каждое из которых, в свою очередь, является деревом. Деревья $T_1$, $T_2$, ..., $T_m$ называются поддеревьями данного дерева.

*Лес* – это множество (обычно упорядоченное), состоящее из некоторого

(быть может, равного нулю) числа непересекающихся деревьев.

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

На рисунке 1 представлен пример бинарного дерева.

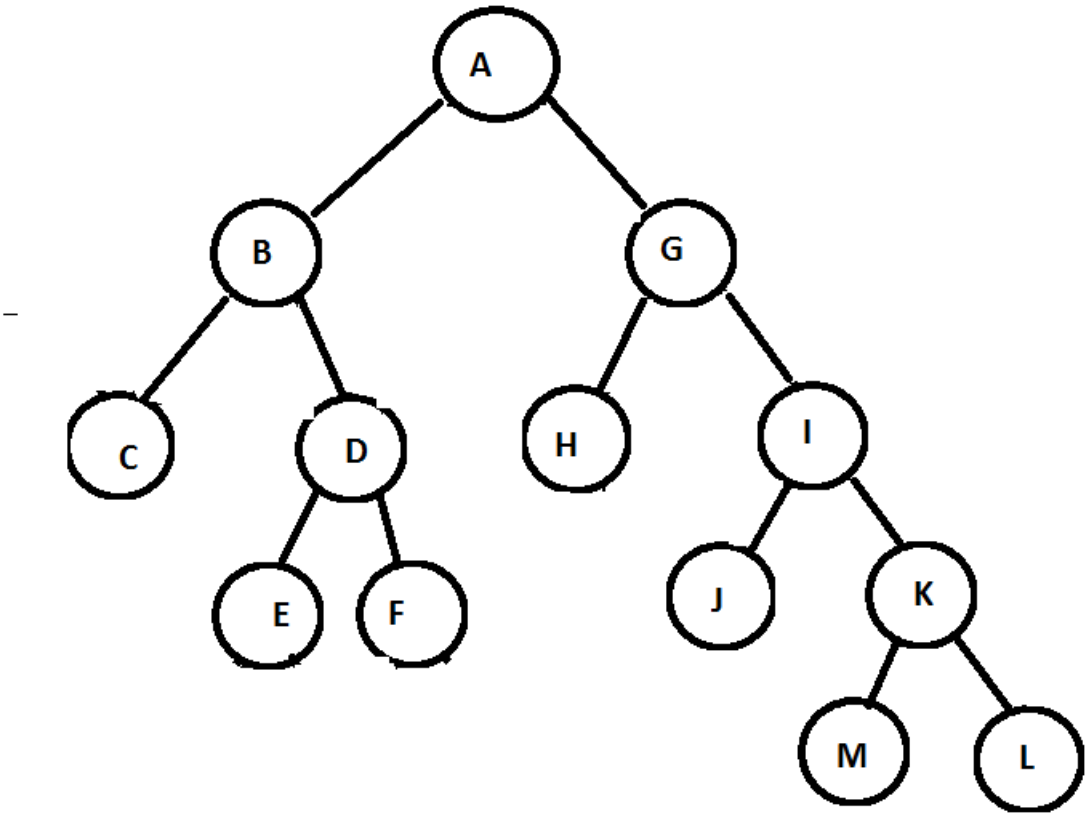


*Рис. 1*

**Описание структур данных и функций.**

1. `class BTnode` – структура элемента бинарного дерева. Имеет следующие поля:

`T data` – данные элемента списка в формате шаблонного типа;

`BTnode* left` – указатель на голову левого поддерева;

`BTnode* right` – указатель на голову правого поддерева;

Имеет следующие методы:

1) `BTnode(T data, BTnode* left, BTnode* right)` – конструктор

элемента бинарного дерева, заполняющий его поля соответствующими значениями.

2) `~BTnode()` – деструктор элемента бинарного дерева, зачищающий память под его левое и правое поддеревья в случае их существования, у которых в свою очередь также рекурсивно вызывается деструктор.

2. `class IterString` – итератор символов по строке. Служит для того, чтобы при каждом новом обращении к строке, возвращать ее очередной, следующий за предыдущим, символ. Имеет следующие поля:

`string data` – строка итератора;

`int index` – индекс текущего символа;

`int size` – размер строки-итератора;

Имеет следующие методы:

1) `IterString(string data)` – конструктор итератора, принимающий строку, и заполняющий ей соответствующее поле *data*. Также поле *index* обнуляется, так как первый текущий символ строки находится под нулевым индексом. Поле *size* заполняется размером строки, возвращаемой методом *length()*.

2) `char getChar()` – метод, возвращающий символ под текущим индексом строки: *data[index++]*, после чего текущий индекс инкрементируется для доступа уже к следующему символу строки. В случае совпадения текущего индекса с размером строки метод возвращает *'\0'*, как признак окончания строки.

3. Функции для печати информации в файл или в консоль:

`void print(string str, ofstream* fout)` – функция, производящая печать строки str в файл, адрес дескриптора которого передается аргументом fout, если он не нулевой. В противном случае печать строки производится на консоль. Данная функция используется при выводе всех промежуточных данных программы.

`void printStep(int step, ofstream* fout)` – функция, производящая печать табуляций в файл или на консоль в количестве step. Требуется для печати

отступов, соответствующих глубине рекурсии.

4. Функции для создания и печати бинарного дерева:

`BTnode<char>* createBT(IterString& input, bool& error)` – рекурсивная функция создания бинарного дерева. Принимает по ссылке итератор строки с записью бинарного дерева и флаг ошибки построения.

Возвращает указатель на голову бинарного дерева, данные которого представлены в символьном виде.

Вызовом метода getChar() выдается очередной символ строки с записью бинарного дерева. Далее проверяется, что строка еще не проитерировалась полностью, в противном случае бинарное дерево считается еще не достроенным до конца, а строка с его записью уже закончилась, что свидетельствует об ошибке записи бинарного дерева. В этом случае флаг *error* устанавливается в положительное значение, и функция возвращается нулевой указатель.

Далее проверяется, что выданный символ является слэшом, что свидетельствует о том, что поддерево, строящаяся этим вызовом функции – нулевое, в случае чего возвращается нулевой указатель.

Во всех остальных случаях данная функция рекурсивно с теми же переменными по ссылке вызывается для построения сначала левого, а потом правого поддерева данного дерева. Указатели на поддеревья, которые были возвращены рекурсивными вызовами этой функции, а также текущий символ отправляются в конструктор элемента бинарного дерева *BTnode()*, для которого выделяется память, указатель на которую функция возвращает.

`void printBT(BTnode<T>* root, ofstream* fout)` – функция, производящая печать бинарного дерева в файл или на консоль.

Принимает указатель на голову бинарного дерева, которое требуется распечатать и указатель на файловый поток, в который требуется распечатать. Для начала проверяется, что корень дерева существует, т. е. его указатель не нулевой, в случае чего вызовом функции *print()* в поток печатается символ-значение корня, затем данная функция рекурсивно вызывается сначала для

левого, потом для правого поддерева данного дерева. Если текущее дерево пусто, печатается символ *‘/’*.

Функция ничего не возвращает.

5. Функции для обработки бинарных деревьев и реализации задания:

`bool isSimilar(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream* fout)` – рекурсивная функция проверяющая два дерева b1 и b2 на предмет подобия, т. е. либо одновременной пустоты, либо подобия их правых и левых поддеревьев. Принимает указатели на корни соответствующих деревьев, символ *under*, служащий для определения, какие именно поддеревья по отношению к своим отцам проверяются на подобие для вывода информации о вызове и завершении функции, а также параметры *step* и *fout* для вывода в поток *fout* промежуточных данных с определенным числом отступов *step*, соответствующим глубине рекурсии.

Возвращает функция булевское значение *true* или *false*, отвечающее на вопрос подобны ли деревья.

`bool isEqual(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream* fout)` – рекурсивная функция проверяющая два дерева b1 и b2 на предмет равенства, т. е. подобия и равенства соответствующих значений элементов. Принимает указатели на корни соответствующих деревьев, символ *under*, служащий для определения, какие именно поддеревья по отношению к своим отцам проверяются на равенство для вывода информации о вызове и завершении функции, а также параметры *step* и *fout* для вывода в поток *fout* промежуточных данных с определенным числом отступов *step*, соответствующим глубине рекурсии.

Возвращает функция булевское значение *true* или *false*, отвечающее на вопрос равны ли деревья.

`bool isMirrorSimilar(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream* fout)` – рекурсивная функция проверяющая два дерева b1 и b2 на предмет зеркального подобия, т. е. либо одновременной пустоты, либо зеркального подобия левого поддерева каждого дерева с правым поддеревом

другого. Принимает указатели на корни соответствующих деревьев, символ *under*, служащий для определения, какие именно поддеревья по отношению к своим отцам проверяются на зеркальное подобие для вывода информации о вызове и завершении функции, а также параметры *step* и *fout* для вывода в поток *fout* промежуточных данных с определенным числом отступов *step*, соответствующим глубине рекурсии.

Возвращает функция булевское значение *true* или *false*, отвечающее на вопрос зеркально подобны ли деревья.

`bool isSymmetric(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream* fout)` – рекурсивная функция проверяющая два дерева b1 и b2 на предмет симметричности, т. е. зеркального подобия и равенства соответствующих значений элементов. Принимает указатели на корни соответствующих деревьев, символ *under*, служащий для определения, какие именно поддеревья по отношению к своим отцам проверяются на симметричность для вывода информации о вызове и завершении функции, а также параметры *step* и *fout* для вывода в поток *fout* промежуточных данных с определенным числом отступов *step*, соответствующим глубине рекурсии.

Возвращает функция булевское значение *true* или *false*, отвечающее на вопрос симметричны ли деревья.

6. Главная функция и функция печати результата:

`void writeRez(bool correct, string end, BTnode<T>* b1, BTnode<T>* b2, ofstream* fout)` – функция, производящая печать результата работы программы. Принимает булевское значение результата проверки двух деревьев *b1* и *b2*, строку *end*, которой заканчивается вывод и указатель на поток вывода.

Функция производит печать на консоль однозначно, а если адрес файлового потока не нулевой, то также и в файловый поток.

`int main(int argc, char* argv[])` – главная функция, выполняющая программу. Она принимает массив аргументов командной строки и их количество. Если аргументов не имеется, считывание исходного бинарного

дерева и печать информации производятся в консоль. Если аргумент один, то он воспринимается как имя файла, с которого будет считаны бинарные деревья. В случае же двух аргументов, первый из них – название файла со входными данными, а второй – с выходными. В любом случае результат всегда выводится на консоль, а одно из четырех возможных действий над бинарными деревьями считывается с консоли. Также в функции имеется проверка на корректность открытия файлов. В случае неудачного открытия файла с тем или иным названием, программа завершается с соответствующим сообщением и кодом ошибки 1.

Сперва производится чтение двух бинарных деревьев подряд, создание из прочитанных строк – строк-итераторов, и создание самих бинарных деревьев с помощью вызова функции *createBT()* от соответствующих строк-итераторов и переменных некорректного построения. После создания каждого дерева проверяется, что при его создании не произошла ошибка, и что вся строка итератора была прочитана до конца. В случае несоблюдения этих условий, построенные деревья зачищаются, и программа прерывается с ошибкой.

Далее с консоли вводится команда, задающая тип проверки двух бинарных деревьев:

*sim* – проверить на подобие;

*eq* – проверить на равенство;

*mirsim* – проверить на зеркальное подобие;

*symm* – проверить на симметричность.

В зависимости от введенной команды от двух деревьев вызывается соответствующая функция проверки, результат которой тут же передается функции печати результата.

**Описание алгоритма.**

В каждой функции проверки бинарных деревьев для начала проверяется, что они оба пусты. Это означает что они равны/подобны/зеркально подобны/симметричны. Тогда функция возвращает *true* и выводит информацию

об этом в промежуточные данные. Иначе проверяется что они оба не пусты, в случае чего есть смысл проверять конкретные поля бинарных деревьев. Тогда, если это функции isEqual() или isSymmetric(), где имеет значение четкое равенство данных, проверяется равенство значений корней этих деревьев. В случае успешной проверки рекурсивно вызываются подряд функции для проверки на подобие или равенство левых и правых поддеревьев / на зеркальное подобие или симметричность левого поддерева каждого дерева с правым поддеревом другого. Только если эти две проверки дали положительный результат, функция возвращает положительное значение, и выводит информацию об этом в промежуточные данные. Если какое-то условие не соблюдается, функция возвращает отрицательное значение и выводит в промежуточные данные информацию об этом и причину неудачи.

**Тестирование.**

Основной тест – 1:

**Входные данные:** *a/bcd///e//*

*n/qwe///g//*

*sim*

**Выходные данные:**

*Функция 'isSimilar()' для деревьев: 'a/bcd///e//' и 'n/qwe///g//' вызвана*

*Функция 'isSimilar()' для левых поддеревьев: '/' и '/' вызвана*

*Деревья: '/' и '/' подобны, так как пусты*

*Функция 'isSimilar()' для левых поддеревьев: '/' и '/' завершена*

*Функция 'isSimilar()' для правых поддеревьев: 'bcd///e//' и 'qwe///g//' вызвана*

*Функция 'isSimilar()' для левых поддеревьев: 'cd///' и 'we///' вызвана*

*Функция 'isSimilar()' для левых поддеревьев: 'd//' и 'e//' вызвана*

*Функция 'isSimilar()' для левых поддеревьев: '/' и '/' вызвана*

*Деревья: '/' и '/' подобны, так как пусты*

*Функция 'isSimilar()' для левых поддеревьев: '/' и '/' завершена*

*Функция 'isSimilar()' для правых поддеревьев: '/' и '/' вызвана*

*Деревья: '/' и '/' подобны, так как пусты*

*Функция 'isSimilar()' для правых поддеревьев: '/' и '/' завершена*

*Деревья: 'd//' и 'e//' подобны, так как подобны их правые и левые поддеревья*

*Функция 'isSimilar()' для левых поддеревьев: 'd//' и 'e//' завершена*

*Функция 'isSimilar()' для правых поддеревьев: '/' и '/' вызвана*

*Деревья: '/' и '/' подобны, так как пусты*

*Функция 'isSimilar()' для правых поддеревьев: '/' и '/' завершена*

*Деревья: 'cd///' и 'we///' подобны, так как подобны их правые и левые поддеревья*

*Функция 'isSimilar()' для левых поддеревьев: 'cd///' и 'we///' завершена*

*Функция 'isSimilar()' для правых поддеревьев: 'e//' и 'g//' вызвана*

*Функция 'isSimilar()' для левых поддеревьев: '/' и '/' вызвана*

*Деревья: '/' и '/' подобны, так как пусты*

*Функция 'isSimilar()' для левых поддеревьев: '/' и '/' завершена*

*Функция 'isSimilar()' для правых поддеревьев: '/' и '/' вызвана*

*Деревья: '/' и '/' подобны, так как пусты*

*Функция 'isSimilar()' для правых поддеревьев: '/' и '/' завершена*

*Деревья: 'e//' и 'g//' подобны, так как подобны их правые и левые поддеревья*

*Функция 'isSimilar()' для правых поддеревьев: 'e//' и 'g//' завершена*

*Деревья: 'bcd///e//' и 'qwe///g//' подобны, так как подобны их правые и левые поддеревья*

*Функция 'isSimilar()' для правых поддеревьев: 'bcd///e//' и 'qwe///g//' завершена*

*Деревья: 'a/bcd///e//' и 'n/qwe///g//' подобны, так как подобны их правые и левые поддеревья*

*Функция 'isSimilar()' для деревьев: 'a/bcd///e//' и 'n/qwe///g//' завершена*

*Результат: Деревья: 'a/bcd///e//' и 'n/qwe///g//' подобны*

Дополнительное тестирование:

| **Номер теста** | **Входные данные** | **Выходные данные** |
|---|---|---|
| **2** | /<br>/<br>eq | Результат: Деревья: '/' и '/' равны |
| **3** | a//<br>b//<br>mirsim | Результат: Деревья: 'a//' и 'b//' зеркально подобны |
| **4** | abc////<br>abc////<br>symm | Результат: Деревья: 'abc////' и 'abc////' не симметричны |
| **5** | ab//cd///<br>aq/t//r//<br>mirsim | Результат: Деревья: ' ab//cd///' и ' aq/t//r//' зеркально подобны |
| **6** | av//w//<br>av//v//<br>eq | Результат: Деревья: 'av//w//' и 'av//v//' не равны |
| **7** | abd//e//cp//g//<br>TTL///GS/P///<br>sim | Результат: Деревья: 'abd//e//cp//g//' и 'TTL///GS/P///' не подобны |
| **8** | f//<br>////<br>eq | Ошибка при чтении второго бинарного дерева |
| **9** | a/bcd//e///<br>ab/ce//d///<br>symm | Результат: Деревья: 'a/bcd//e///' и ' ab/ce//d///' симметричны |
| **10** | abc//<br>/<br>symm | Ошибка при чтении первого бинарного дерева |
| **11** | as/// | Не распознано действие |

| | | |
|---|---|---|
| | tp///<br>qwe | |

**Вывод.**

В ходе лабораторной работы было проведено ознакомление с бинарными деревьями, способами их представления, записи, а также рекурсивной обработки на языке программирования Си++.

## ПРИЛОЖЕНИЕ А
## ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл AiSD_lab_3.cpp:

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

using namespace std;

#define F(argc) (argc == 3 ? &fout: nullptr)

// Классы--------------------------------------------------------------------
template <class T>
class BTnode{
public:
    T data;
    BTnode* left;
    BTnode* right;
    BTnode(T data, BTnode* left, BTnode* right):data(data), left(left),
right(right){}
    ~BTnode(){
        if (this->left)
            delete this->left;
        if (this->right)
            delete this->right;
    }
};

class IterString{
```

```
private:
    string data;
    int index;
    int size;
public:
    IterString(string data):data(data), index(0), size(data.length()){}
    char getChar(){
        if (index == size)
            return '\0';
        else
            return data[index++];
    }
};
//---------------------------------------------------------------------------

// Функции для печати-------------------------------------------------------
template <class T>
void print(T str, ofstream* fout) {
    if (fout) *fout << str;
    else cout << str;
}
void printStep(int step, ofstream* fout){
    for (int i = 0; i < step; i++) print<char>('\t', fout);
}
//---------------------------------------------------------------------------

// Функции для создания и печати бинарного дерева---------------------------
BTnode<char>* createBT(IterString& input, bool& error){
    char c;
    BTnode<char> *left, *right;
    c = input.getChar();
    if (c == '\0') {
        error = true;
        return nullptr;
    }
    if (c == '/') return nullptr;
    else {
        left = createBT(input, error);
        right = createBT(input, error);
        return new BTnode<char>(c, left, right);
    }
}

template <class T>
```

```
void printBT(BTnode<T>* root, ofstream* fout){
    if (root){
        print<T>(root->data, fout);
        printBT<T>(root->left, fout);
        printBT<T>(root->right, fout);
    } else print<char>('/', fout);
}
//---------------------------------------------------------------------------

// Функции для анализа пар бинарных деревьев---------------------------------
template <class T>
bool isSimilar(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream*
fout){
    // Печать информации о вызове функции
    printStep(step, fout);
    switch (under) {
        case '0':
            print<string>("Функция 'isSimilar()' для деревьев: '", fout);
            break;
        case 'l':
            print<string>("Функция 'isSimilar()' для левых поддеревьев: '", fout);
            break;
        case 'r':
            print<string>("Функция 'isSimilar()' для правых поддеревьев: '",
fout);
            break;
    }
    printBT<T>(b1, fout);
    print<string>("' и '", fout);
    printBT<T>(b2, fout);
    print<string>("' вызвана\n", fout);
    bool flag = false; // Инициализация результирующего флага отрицательным
значением

    if (b1 == nullptr && b2 == nullptr){ // Проверка на пустоту обоих деревьев
        printStep(step, fout);
        print<string>("Деревья: '", fout);
        printBT<T>(b1, fout);
        print<string>("' и '", fout);
        printBT<T>(b2, fout);
        print<string>("' подобны, так как пусты\n", fout);
        flag = true; // Результирующий флаг принимает положительное значение
    } else {
        if (b1 && b2){ // Проверка на не пустоту обоих деревьев
```

```
            if (isSimilar<T>(b1->left, b2->left, 'l', step + 1, fout)){ //
Проверка на подобие левых поддеревьев
                print<string>("\n", fout);
                if (isSimilar<T>(b1->right, b2->right, 'r', step + 1, fout)){ //
Проверка на подобие правых поддеревьев
                    printStep(step, fout);
                    print<string>("Деревья: '", fout);
                    printBT<T>(b1, fout);
                    print<string>("' и '", fout);
                    printBT<T>(b2, fout);
                    print<string>("' подобны, так как подобны их правые и левые
поддеревья\n", fout);
                    flag = true; // Результирующий флаг принимает положительное
значение
                } else {
                    printStep(step, fout);
                    print<string>("Деревья: '", fout);
                    printBT<T>(b1, fout);
                    print<string>("' и '", fout);
                    printBT<T>(b2, fout);
                    print<string>("' не подобны, так как не подобны их правые
поддеревья\n", fout);
                }
            } else {
                printStep(step, fout);
                print<string>("Деревья: '", fout);
                printBT<T>(b1, fout);
                print<string>("' и '", fout);
                printBT<T>(b2, fout);
                print<string>("' не подобны, так как не подобны их левые
поддеревья\n", fout);
            }
        } else {
            printStep(step, fout);
            print<string>("Деревья: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            print<string>("' не подобны, так как одно из них пусто, а другое
нет\n", fout);
        }
    }
    // Печать информации о завершении функции
    printStep(step, fout);
```

```
    switch (under) {
        case '0':
            print<string>("Функция 'isSimilar()' для деревьев: '", fout);
            break;
        case 'l':
            print<string>("Функция 'isSimilar()' для левых поддеревьев: '", fout);
            break;
        case 'r':
            print<string>("Функция  'isSimilar()'  для  правых  поддеревьев:  '",
fout);
            break;
    }
    printBT<T>(b1, fout);
    print<string>("' и '", fout);
    printBT<T>(b2, fout);
    print<string>("' завершена\n", fout);
    return flag; // Возврат значения результирующего флага
}

template <class T>
bool isEqual(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream* fout){
    // Печать информации о вызове функции
    printStep(step, fout);
    switch (under) {
        case '0':
            print<string>("Функция 'isEqual()' для деревьев: '", fout);
            break;
        case 'l':
            print<string>("Функция 'isEqual()' для левых поддеревьев: '", fout);
            break;
        case 'r':
            print<string>("Функция 'isEqual()' для правых поддеревьев: '", fout);
            break;
    }
    printBT<T>(b1, fout);
    print<string>("' и '", fout);
    printBT<T>(b2, fout);
    print<string>("' вызвана\n", fout);
    bool  flag  =  false;  // Инициализация  результирующего  флага  отрицательным
значением

    if (b1 == nullptr && b2 == nullptr){ // Проверка на пустоту обоих деревьев
        printStep(step, fout);
        print<string>("Деревья: '", fout);
```

```
        printBT<T>(b1, fout);
        print<string>("' и '", fout);
        printBT<T>(b2, fout);
        print<string>("' равны, так как пусты\n", fout);
        flag = true; // Результирующий флаг принимает положительное значение
    } else {
        if (b1 && b2){ // Проверка на не пустоту обоих деревьев
            if (b1->data == b2->data){ // Проверка на равенство значений корней деревьев
                printStep(step, fout);
                print<string>("Значения корней деревьев: '", fout);
                printBT<T>(b1, fout);
                print<string>("' и '", fout);
                printBT<T>(b2, fout);
                print<string>("' равны '", fout);
                print<T>(b1->data, fout);
                print<string>("'\n", fout);
                if (isEqual<T>(b1->left, b2->left, 'l', step + 1, fout)){ // Проверка на равенство левых поддеревьев
                    print<string>("\n", fout);
                    if (isEqual<T>(b1->right, b2->right, 'r', step + 1, fout)){ // Проверка на равенство правых поддеревьев
                        printStep(step, fout);
                        print<string>("Деревья: '", fout);
                        printBT<T>(b1, fout);
                        print<string>("' и '", fout);
                        printBT<T>(b2, fout);
                        print<string>("' равны, так как равны значения их корней, их правые и левые поддеревья\n", fout);
                        flag = true; // Результирующий флаг принимает положительное значение
                    } else {
                        printStep(step, fout);
                        print<string>("Деревья: '", fout);
                        printBT<T>(b1, fout);
                        print<string>("' и '", fout);
                        printBT<T>(b2, fout);
                        print<string>("' не равны, так как не равны их правые поддеревья\n", fout);
                    }
                } else {
                    printStep(step, fout);
                    print<string>("Деревья: '", fout);
                    printBT<T>(b1, fout);
```

```
                    print<string>("' и '", fout);
                    printBT<T>(b2, fout);
                    print<string>("'  не  равны,  так  как  не  равны  их  левые
поддеревья\n", fout);
                }
            } else {
                printStep(step, fout);
                print<string>("Деревья: '", fout);
                printBT<T>(b1, fout);
                print<string>("' и '", fout);
                printBT<T>(b2, fout);
                print<string>("' не равны, так как не равны значения их корней\n",
fout);
            }
        } else {
            printStep(step, fout);
            print<string>("Деревья: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            print<string>("' не равны, так как одно из них пусто, а другое нет\n",
fout);
        }
    }
    // Печать информации о завершении функции
    printStep(step, fout);
    switch (under) {
        case '0':
            print<string>("Функция 'isEqual()' для деревьев: '", fout);
            break;
        case 'l':
            print<string>("Функция 'isEqual()' для левых поддеревьев: '", fout);
            break;
        case 'r':
            print<string>("Функция 'isEqual()' для правых поддеревьев: '", fout);
            break;
    }
    printBT<T>(b1, fout);
    print<string>("' и '", fout);
    printBT<T>(b2, fout);
    print<string>("' завершена\n", fout);
    return flag; // Возврат значения результирующего флага
}
```

```
template <class T>
bool isMirrorSimilar(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream*
fout){
    // Печать информации о вызове функции
    printStep(step, fout);
    switch (under) {
        case '0':
            print<string>("Функция 'isMirrorSimilar()' для деревьев: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            break;
        case 'l':
            print<string>("Функция  'isMirrorSimilar()'  для  левого  поддерева
первого дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и правого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
        case 'r':
            print<string>("Функция  'isMirrorSimilar()'  для  правого  поддерева
первого дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и левого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
    }
    print<string>("' вызвана\n", fout);
    bool  flag  =  false;  // Инициализация результирующего флага отрицательным
значением

    if (b1 == nullptr && b2 == nullptr){ // Проверка на пустоту обоих деревьев
        printStep(step, fout);
        print<string>("Деревья: '", fout);
        printBT<T>(b1, fout);
        print<string>("' и '", fout);
        printBT<T>(b2, fout);
        print<string>("' зеркально подобны, так как пусты\n", fout);
        flag = true; // Результирующий флаг принимает положительное значение
    } else {
        if (b1 && b2){ // Проверка на не пустоту обоих деревьев
            if (isMirrorSimilar<T>(b1->left, b2->right, 'l', step + 1, fout)){ //
Проверка на зеркальное подобие левого поддерева первого дерева и правого поддерева
второго
```

```
                print<string>("\n", fout);
                if (isMirrorSimilar<T>(b1->right, b2->left, 'r', step + 1, fout)){ // Проверка на зеркальное подобие правого поддерева первого дерева и левого поддерева второго
                    printStep(step, fout);
                    print<string>("Деревья: '", fout);
                    printBT<T>(b1, fout);
                    print<string>("' и '", fout);
                    printBT<T>(b2, fout);
                    print<string>("' зеркально подобны, так как правое поддерево каждого из них зеркально подобно с левым поддеревом другого\n", fout);
                    flag = true; // Результирующий флаг принимает положительное значение
                } else {
                    printStep(step, fout);
                    print<string>("Деревья: '", fout);
                    printBT<T>(b1, fout);
                    print<string>("' и '", fout);
                    printBT<T>(b2, fout);
                    print<string>("' не зеркально подобны, так как правое поддерево первого дерева не зеркально подобно с левым поддеревом второго\n", fout);
                }
            } else {
                printStep(step, fout);
                print<string>("Деревья: '", fout);
                printBT<T>(b1, fout);
                print<string>("' и '", fout);
                printBT<T>(b2, fout);
                print<string>("' не зеркально подобны, так как левое поддерево первого дерева не зеркально подобно с правым поддеревом второго\n", fout);
            }
        } else {
            printStep(step, fout);
            print<string>("Деревья: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            print<string>("' не зеркально подобны, так как одно из них пусто, а другое нет\n", fout);
        }
    }
    // Печать информации о завершении функции
    printStep(step, fout);
    switch (under) {
```

```
        case '0':
            print<string>("Функция 'isMirrorSimilar()' для деревьев: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            break;
        case 'l':
            print<string>("Функция 'isMirrorSimilar()' для левого поддерева
первого дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и правого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
        case 'r':
            print<string>("Функция 'isMirrorSimilar()' для правого поддерева
первого дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и левого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
    }
    print<string>("' завершена\n", fout);
    return flag; // Возврат значения результирующего флага
}

template <class T>
bool isSymmetric(BTnode<T>* b1, BTnode<T>* b2, char under, int step, ofstream*
fout){
    // Печать информации о вызове функции
    printStep(step, fout);
    switch (under) {
        case '0':
            print<string>("Функция 'isSymmetric()' для деревьев: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            break;
        case 'l':
            print<string>("Функция 'isSymmetric()' для левого поддерева первого
дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и правого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
```

```
        case 'r':
            print<string>("Функция 'isSymmetric()' для правого поддерева первого дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и левого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
    }
    print<string>("' вызвана\n", fout);
    bool flag = false; // Инициализация результирующего флага отрицательным значением

    if (b1 == nullptr && b2 == nullptr){ // Проверка на пустоту обоих деревьев
        printStep(step, fout);
        print<string>("Деревья: '", fout);
        printBT<T>(b1, fout);
        print<string>("' и '", fout);
        printBT<T>(b2, fout);
        print<string>("' симметричны, так как пусты\n", fout);
        flag = true; // Результирующий флаг принимает положительное значение
    } else {
        if (b1 && b2){ // Проверка на не пустоту обоих деревьев
            if (b1->data == b2->data){ // Проверка на равенство значений корней деревьев
                printStep(step, fout);
                print<string>("Значения корней деревьев: '", fout);
                printBT<T>(b1, fout);
                print<string>("' и '", fout);
                printBT<T>(b2, fout);
                print<string>("' равны '", fout);
                print<T>(b1->data, fout);
                print<string>("'\n", fout);
                if (isSymmetric<T>(b1->left, b2->right, 'l', step + 1, fout)){ // Проверка на симметричность левого поддерева первого дерева и правого поддерева второго
                    print<string>("\n", fout);
                    if (isSymmetric<T>(b1->right, b2->left, 'r', step + 1, fout)){ // Проверка на симметричность правого поддерева первого дерева и левого поддерева второго
                        printStep(step, fout);
                        print<string>("Деревья: '", fout);
                        printBT<T>(b1, fout);
                        print<string>("' и '", fout);
                        printBT<T>(b2, fout);
```

```
                        print<string>("' симметричны, так как равны значения их
корней, и правое поддерево каждого из них симметрично левому поддереву другого\n",
fout);
                        flag = true; // Результирующий флаг принимает
положительное значение
                    } else {
                        printStep(step, fout);
                        print<string>("Деревья: '", fout);
                        printBT<T>(b1, fout);
                        print<string>("' и '", fout);
                        printBT<T>(b2, fout);
                        print<string>("' не симметричны, так как правое поддерево
первого дерева не симметрично левому поддереву второго\n", fout);
                    }
                } else {
                    printStep(step, fout);
                    print<string>("Деревья: '", fout);
                    printBT<T>(b1, fout);
                    print<string>("' и '", fout);
                    printBT<T>(b2, fout);
                    print<string>("' не симметричны, так как левое поддерево
первого дерева не симметрично правому поддереву второго\n", fout);
                }
            } else {
                printStep(step, fout);
                print<string>("Деревья: '", fout);
                printBT<T>(b1, fout);
                print<string>("' и '", fout);
                printBT<T>(b2, fout);
                print<string>("' не симметричны, так как не равны значения их
корней\n", fout);
            }
        } else {
            printStep(step, fout);
            print<string>("Деревья: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            print<string>("' не симметричны, так как одно из них пусто, а другое
нет\n", fout);
        }
    }
    // Печать информации о завершении функции
    printStep(step, fout);
```

```
    switch (under) {
        case '0':
            print<string>("Функция 'isSymmetric()' для деревьев: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и '", fout);
            printBT<T>(b2, fout);
            break;
        case 'l':
            print<string>("Функция 'isSymmetric()' для левого поддерева первого
дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и правого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
        case 'r':
            print<string>("Функция 'isSymmetric()' для правого поддерева первого
дерева: '", fout);
            printBT<T>(b1, fout);
            print<string>("' и левого поддерева второго: '", fout);
            printBT<T>(b2, fout);
            break;
    }
    print<string>("' завершена\n", fout);
    return flag; // Возврат значения результирующего флага
}

//---------------------------------------------------------------------------
template <class T>
void writeRez(bool correct, string end, BTnode<T>* b1, BTnode<T>* b2, ofstream*
fout){
    print<string>("\nРезультат: Деревья: '", fout);
    if (fout) cout << "\nРезультат: Деревья: '";
    printBT<T>(b1, fout);
    print<string>("' и '", fout);
    printBT<T>(b2, fout);
    if (fout){
        printBT<T>(b1, nullptr);
        print<string>("' и '", nullptr);
        printBT<T>(b2, nullptr);
    }
    print<string>("'", fout);
    if (fout) cout << "'";
    if (!correct){
        print<string>(" не", fout);
```

```
        if (fout) cout << " не";
    }
    print<string>(end, fout);
    if (fout) cout << end;
}

int main(int argc, char* argv[]){
    if (argc > 3){
        cout << "Слишком много аргументов программы\n";
        return 1;
    }

    ifstream fin;
    ofstream fout;
    if (argc > 1){
        fin.open(argv[1]);
        if (!fin){
            cout << "Ошибка открытия файла: " << argv[1] << endl;
            return 1;
        }

        if (argc == 3){
            fout.open(argv[2]);
            if (!fout){
                cout << "Ошибка открытия файла: " << argv[2] << endl;
                fin.close();
                return 1;
            }
        }
    }

    string str1;
    if (argc == 1){
        cout << "Введите первое бинарное дерево: ";
        cin >> str1;
    } else fin >> str1;
    IterString input = IterString(str1);
    bool error = false;
    BTnode<char>* b1 = createBT(input, error);
    if ((error) || (input.getChar() != '\0')){
        delete b1;
        cout << "Ошибка при чтении первого бинарного дерева\n";
        fin.close();
        fout.close();
```

```
        return 1;
    }

    string str2;
    if (argc == 1){
        cout << "Введите второе бинарное дерево: ";
        cin >> str2;
    } else fin >> str2;
    input = IterString(str2);
    error = false;
    BTnode<char>* b2 = createBT(input, error);
    if ((error) || (input.getChar() != '\0')){
        delete b1;
        delete b2;
        cout << "Ошибка при чтении второго бинарного дерева\n";
        fin.close();
        fout.close();
        return 1;
    }

    string option;
    cout << "Введите действие: ";
    cin >> option;
    if (option == "sim")
        writeRez(isSimilar<char>(b1, b2, '0', 0, F(argc)), " подобны\n", b1, b2,
F(argc));
    else {
        if (option == "mirsim")
            writeRez(isMirrorSimilar<char>(b1, b2, '0', 0, F(argc)), " зеркально
подобны\n", b1, b2, F(argc));
        else {
            if (option == "eq")
                writeRez(isEqual<char>(b1, b2, '0', 0, F(argc)), " равны\n", b1,
b2, F(argc));
            else {
                if (option == "symm")
                    writeRez(isSymmetric<char>(b1,  b2,  '0',  0,  F(argc)),  "
симметричны\n", b1, b2, F(argc));
                else
                    cout << "Не распознано действие\n";
            }
        }
    }
```

```
    fin.close();
    fout.close();
    delete b1;
    delete b2;
    return 0;
}
```