МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНЖЕНЕРНО-ТЕХНОЛОГИЧЕСКАЯ АКАДЕМИЯ

Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

**Отчет по лабораторной работе № 2**

по курсу «Структуры и алгоритмы обработки данных»

**«ЗАДАЧИ ОБХОДА ДЕРЕВЬЕВ»**

Выполнили:

студент гр. КТбо2-1

Самардак А.В

Проверил:

Доцент кафедры МОП ЭВМ

Дроздов С.Н

Техническое задание

Вариант 2

1. Реализовать структуру «дерево» (Рис. 1).
2. Реализовать базовые операции, а именно:

* Ввод дерева из файла;
* Отображение введённого дерева;
* Построение сцепленного представления;

1. Реализовать дополнительные операции, а именно:

* Проверить, является заданное дерево сортированным;

Ход работы

**Листинг программы на языке С++**

**Main.cpp**

#include <vld.h>

#include "Console.h"

int main()

{

setlocale(LC\_ALL, "Russian");

Console().start();

return 0;

}

**Console.h**

#pragma once

#include <iostream>

#include <fstream>

#include "Tree.h"

class Console

{

public:

void start();

private:

void create\_tree();

void menu();

void dialog\_check\_tree\_rec();

void dialog\_check\_tree\_no\_rec();

void dialog\_print\_tree();

int dialog\_get\_data();

void print\_menu();

Tree \_tree;

};

**Console.cpp**

#include "Console.h"

void Console::start()

{

try

{

create\_tree();

menu();

}

catch (const std::exception& ex)

{

std::cout << ex.what() << std::endl;

}

}

void Console::create\_tree()

{

std::ifstream fin("Tree.txt");

if (!fin.is\_open())

{

throw std::exception("Нет такого файла...");

}

else

{

std::string key;

int data;

while (fin >> key >> data)

{

\_tree.push(key, data);

}

}

fin.close();

}

void Console::menu()

{

bool operation = true;

while (operation)

{

print\_menu();

int check = dialog\_get\_data();

switch (check)

{

case 1:

dialog\_check\_tree\_rec();

break;

case 2:

dialog\_check\_tree\_no\_rec();

break;

case 3:

dialog\_print\_tree();

break;

case 4:

operation = false;

}

}

}

void Console::dialog\_check\_tree\_rec()

{

if (\_tree.check\_tree\_rec())

{

std::cout << "Да, отсортирован" << std::endl;

}

else

{

std::cout << "Нет, не отсортирован" << std::endl;

}

}

void Console::dialog\_check\_tree\_no\_rec()

{

if (\_tree.check\_tree\_no\_rec())

{

std::cout << "Да, отсортирован" << std::endl;

}

else

{

std::cout << "Нет, не отсортирован" << std::endl;

}

}

void Console::dialog\_print\_tree()

{

std::vector<int> vec;

\_tree.print\_tree(vec);

for (int i = 0; i < vec.size(); i++)

{

std::cout << vec.at(i) << std::endl;

}

}

int Console::dialog\_get\_data()

{

int data;

std::cin >> data;

return data;

}

void Console::print\_menu()

{

std::cout << "------------------" << std::endl;

std::cout << "1. Проверка дерева - рекурсивно" << std::endl;

std::cout << "2. Проверка дерева - не рекурсивно" << std::endl;

std::cout << "3. Вывод дерева" << std::endl;

std::cout << "4. Выход" << std::endl;

std::cout << "------------------" << std::endl;

}

**Tree.h**

#pragma once

#include <iostream>

#include <vector>

#include <string>

class Tree {

private:

struct Node;

public:

~Tree();

void push(std::string& key, int data);

void print\_tree(std::vector<int>& vec);

bool check\_tree\_rec();

bool check\_tree\_no\_rec();

private:

Node\* move\_on\_tree(std::string& key);

void create\_leaf\_tree(Node\* currentElement, std::string& key, int data);

void check\_tree\_rec(Node\* currentElement, bool& flag, int& data);

void check\_tree\_no\_rec(Node\* currentElement, std::vector<int>& vec);

void delete\_tree(Node\* currentElement);

struct Node {

int \_data;

Node\* \_pRight = nullptr;

Node\* \_pLeft = nullptr;

};

Node\* \_head = nullptr;

};

**Tree.cpp**

#include "Tree.h"

Tree::~Tree()

{

if (\_head)

{

delete\_tree(\_head);

}

}

void Tree::push(std::string& key, int data)

{

if (!\_head)

{

\_head = new Node();

\_head->\_data = data;

}

else

{

Node\* currentElement = move\_on\_tree(key);

create\_leaf\_tree(currentElement, key, data);

}

}

void Tree::print\_tree(std::vector<int>& vec)

{

check\_tree\_no\_rec(\_head, vec);

}

bool Tree::check\_tree\_rec()

{

bool flag = true;

int data;

check\_tree\_rec(\_head, flag, data);

return flag;

}

bool Tree::check\_tree\_no\_rec()

{

bool flag = true;

std::vector<int> vec;

check\_tree\_no\_rec(\_head, vec);

for (int i = 0; i < vec.size() - 1; i++)

{

if (vec.at(i) > vec.at(i + 1))

{

flag = !flag;

break;

}

}

return flag;

}

Tree::Node\* Tree::move\_on\_tree(std::string& key)

{

Node\* currentElement = \_head;

for (int i = 1; i < key.size() - 1; i++)

{

if (key[i] == '0')

{

currentElement = currentElement->\_pLeft;

}

else

{

currentElement = currentElement->\_pRight;

}

}

return currentElement;

}

void Tree::create\_leaf\_tree(Node\* currentElement, std::string& key, int data)

{

if (key[key.size() - 1] == '0')

{

currentElement->\_pLeft = new Node();

currentElement->\_pLeft->\_data = data;

}

else

{

currentElement->\_pRight = new Node();

currentElement->\_pRight->\_data = data;

}

}

void Tree::check\_tree\_rec(Node\* currentElement, bool& flag, int& data)

{

if (!currentElement)

{

return;

}

check\_tree\_rec(currentElement->\_pLeft, flag, data);

if (!flag)

{

return;

}

if (!currentElement->\_pLeft && !currentElement->\_pRight)

{

data = currentElement->\_data;

return;

}

else if (currentElement->\_data > data)

{

data = currentElement->\_data;

}

else

{

flag = !flag;

}

check\_tree\_rec(currentElement->\_pRight, flag, data);

}

void Tree::check\_tree\_no\_rec(Node\* currentElement, std::vector<int>& vec)

{

if (currentElement->\_pLeft)

{

check\_tree\_no\_rec(currentElement->\_pLeft, vec);

}

vec.push\_back(currentElement->\_data);

if (currentElement->\_pRight)

{

check\_tree\_no\_rec(currentElement->\_pRight, vec);

}

}

void Tree::delete\_tree(Node\* currentElement)

{

if (currentElement->\_pLeft)

{

delete\_tree(currentElement->\_pLeft);

}

if (currentElement->\_pRight)

{

delete\_tree(currentElement->\_pRight);

}

delete currentElement;

}