**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра Вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «ДЕРЕВЬЯ»**

Студенты гр. 3312 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Поляков А.И. Половникова А.С.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Колинько П.Г.

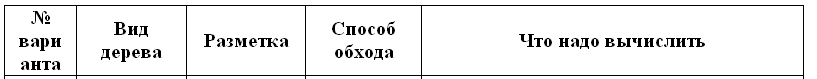
Санкт-Петербург

2024

**Цель работы.**

Исследование алгоритмов для работы с двоичным деревом.

**Задание 9.**

****

****

# **Обоснование выбора способа представления деревьев в памяти ЭВМ**

Дерево и его листья реализованы при помощи списков они обеспечивают эффективную работу с динамическими структурами данных. Списки позволяют хранить элементы, связанные между собой указателями, что упрощает добавление и удаление элементов, а также поиск и обход элементов дерева.

**Результаты работы программы**





Рисунок 1-2. Авто-генерация дерева



Рисунок 3. Ввод вручную

# 

# **Оценка временной сложности алгоритма**

Временная сложность для создания дерева – O(1)

Временная сложность для обработки дерева – O(1)

Временная сложность для вывода дерева – O(1)

**Вывод.**

В данной лабораторной работе мы получили практические навыки работы с двоичным деревом и алгоритмом прямой разметки и внутреннего обхода дерева.

**Список использованных источников.**

Колинько П. Г. Пользовательские структуры данных: Методические указания по дисциплине “Алгоритмы и структуры данных, часть 1”. - СПб.: СПбГЭТУ “ЛЭТИ”, 2023. - 64 c. (вып.2309).

**Приложение. Исходные тексты программ.**

**Основная программа**

#include <iostream>

#include <random>

#include "Tree.h"

#include "TreeNode.h"

int main()

{

int choise = 0;

Tree tree;

while (!choise)

{

std::cout << "#==========================#" << std::endl;

std::cout << "1 - Create a tree manually" << std::endl;

std::cout << "2 - Generate tree" << std::endl;

std::cout << "#==========================#" << std::endl;

std::cin >> choise;

switch (choise)

{

case 1:

system("cls");

tree.manualInput();

break;

case 2:

system("cls");

tree.generateRandomValues(10, 1, 9);

break;

default:

system("cls");

break;

}

}

tree.OutTree(); std::cout << std::endl << std::endl;

std::cout << "in-order traversal:"; tree.inOrderTraversal();

auto nodes = tree.getNodesInOrderUpToLevel2();

std::cout << "Nodes up to level 2 (in-order): ";

for (int val : nodes)

std::cout << val << " ";

std::cout << std::endl;

std::cout << "Quantity nodes up to level 2: " << nodes.size();

return 0;

}

**файл “Tree.h”**

#pragma once

#include <iostream>

#include <memory>

#include <iomanip>

#include <queue>

#include <cmath>

#include <random>

#include "TreeNode.h"

class Tree

{

protected:

const int maxrow = 20;

const int maxcol = 80;

char\*\* SCREEN;

const int offset = 40;

std::shared\_ptr<TreeNode> root;

void addNodeHelper(std::shared\_ptr<TreeNode>& node, int value);

void inOrderTraversalHelper(std::shared\_ptr<TreeNode> node) const;

void printTreeHelper(std::ostream& os, std::shared\_ptr<TreeNode> node) const;

static int getHeight(const std::shared\_ptr<TreeNode>& node);

void collectInOrderNodesUpToLevel(std::shared\_ptr<TreeNode> node, int currentLevel, int maxLevel, std::vector<int>& result);

public:

Tree();

Tree(const Tree&) = delete;

Tree(Tree&&) = delete;

void addNode(int value);

void inOrderTraversal() const;

friend std::ostream& operator<<(std::ostream& os, const Tree& tree);

void OutTree();

void clrscr();

void OutNodes(std::shared\_ptr<TreeNode> v, int r, int c);

std::vector<int> getNodesInOrderUpToLevel2();

void manualInput();

void generateRandomValues(int n, int minValue, int maxValue);

~Tree();

};

**файл “Tree.cpp”**

#include "Tree.h"

Tree::Tree() : root(nullptr)

{

SCREEN = new char\* [maxrow];

for (int i = 0; i < maxrow; ++i)

SCREEN[i] = new char[maxcol];

}

void Tree::addNodeHelper(std::shared\_ptr<TreeNode>& node, int value)

{

if (!node)

node = std::make\_shared<TreeNode>(value);

else if (value < node->value)

addNodeHelper(node->left, value);

else

addNodeHelper(node->right, value);

}

void Tree::inOrderTraversalHelper(std::shared\_ptr<TreeNode> node) const

{

if (node)

{

inOrderTraversalHelper(node->left);

std::cout << node->value << " ";

inOrderTraversalHelper(node->right);

}

}

void Tree::printTreeHelper(std::ostream& sys, std::shared\_ptr<TreeNode> node) const

{

if (!node) return;

int height = getHeight(node);

int maxNodes = std::pow(2, height) - 1;

int width = 4 \* maxNodes;

std::queue<std::shared\_ptr<TreeNode>> q;

q.push(node);

for (int level = 0; level < height; ++level)

{

int levelNodes = std::pow(2, level);

int spaceBetween = width / levelNodes;

for (int i = 0; i < levelNodes; ++i)

{

std::shared\_ptr<TreeNode> current = q.front();

q.pop();

if (current)

{

sys << std::setw(spaceBetween / 2) << ""

<< current->value

<< std::setw(spaceBetween / 2) << "";

q.push(current->left);

q.push(current->right);

}

else

{

sys << std::setw(spaceBetween) << "";

q.push(nullptr);

q.push(nullptr);

}

}

sys << std::endl;

}

}

int Tree::getHeight(const std::shared\_ptr<TreeNode>& node)

{

if (!node) return 0;

return 1 + std::max(getHeight(node->left), getHeight(node->right));

}

void Tree::addNode(int value)

{

addNodeHelper(root, value);

}

void Tree::inOrderTraversal() const

{

inOrderTraversalHelper(root);

std::cout << std::endl;

}

std::ostream& operator<<(std::ostream& sys, const Tree& tree)

{

tree.printTreeHelper(sys, tree.root);

return sys;

}

void Tree::OutTree()

{

clrscr();

OutNodes(root, 1, offset);

for (int i = 0; i < maxrow; i++)

{

SCREEN[i][maxcol - 1] = 0;

std::cout << SCREEN[i] << "\n";

}

std::cout << "\n";

}

void Tree::clrscr()

{

for (int i = 0; i < maxrow; i++)

memset(SCREEN[i], '.', maxcol);

}

void Tree::OutNodes(std::shared\_ptr<TreeNode> v, int r, int c)

{

if (!v || r >= maxrow || c < 1 || c >= maxcol) return;

SCREEN[r - 1][c - 1] = '0' + v->value;

if (v->left)

{

SCREEN[r][c - 2] = '/';

OutNodes(v->left, r + 2, c - (offset >> r));

}

if (v->right)

{

SCREEN[r][c] = '\\';

OutNodes(v->right, r + 2, c + (offset >> r));

}

}

void Tree::collectInOrderNodesUpToLevel(std::shared\_ptr<TreeNode> node, int currentLevel, int maxLevel, std::vector<int>& result)

{

if (!node || currentLevel > maxLevel)

return;

collectInOrderNodesUpToLevel(node->left, currentLevel + 1, maxLevel, result);

if (currentLevel <= maxLevel)

result.push\_back(node->value);

collectInOrderNodesUpToLevel(node->right, currentLevel + 1, maxLevel, result);

}

std::vector<int> Tree::getNodesInOrderUpToLevel2()

{

std::vector<int> result;

collectInOrderNodesUpToLevel(root, 0, 2, result);

return result;

}

void Tree::manualInput()

{

int n;

std::cout << "Enter the number of nodes: ";

std::cin >> n;

if (n < 0 || n == 0)

return;

std::cout << "Enter node values: ";

for (int i = 0; i < n; ++i)

{

int value;

std::cin >> value;

addNode(value);

}

}

void Tree::generateRandomValues(int n, int minValue, int maxValue)

{

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> distrib(minValue, maxValue);

std::cout << "Generation " << n << " random values: ";

for (int i = 0; i < n; ++i)

{

int value = distrib(gen);

std::cout << value << " ";

addNode(value);

}

std::cout << std::endl;

}

Tree::~Tree()

{

for (int i = 0; i < maxrow; ++i)

delete[] SCREEN[i];

delete[] SCREEN;

}

**файл “TreeNode.h”**

#pragma once

#include <memory>

class TreeNode

{

public:

int value;

std::shared\_ptr<TreeNode> left;

std::shared\_ptr<TreeNode> right;

TreeNode(int val);

TreeNode(const TreeNode&) = delete;

TreeNode(TreeNode&&) = delete;

~TreeNode();

};

**файл “TreeNode.cpp”**

#include "TreeNode.h"

TreeNode::TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}

TreeNode::~TreeNode() {}