**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.

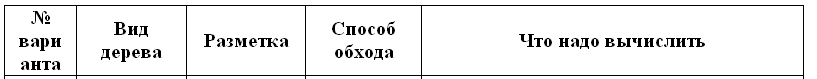
Санкт-Петербург

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.

Обоснование использования

**Результаты работы программы** **shared\_ptr**

Использование std::shared\_ptr в классе Tree оправдано несколькими факторами, которые обеспечивают удобство управления памятью и безопасность при многократном доступе к узлам дерева. Рассмотрим основные причины:

1. **Автоматическое управление памятью**: std::shared\_ptr является умным указателем, который использует счетчик ссылок для отслеживания количества ссылок на объект. Когда этот счетчик становится равным нулю (то есть когда на объект больше не указывает ни один указатель), память автоматически освобождается. Это помогает избежать утечек памяти, которые могут возникнуть при ручном управлении памятью, что особенно важно в сложных структурах данных, таких как деревья.
2. **Многократное владение объектами**: В дереве каждый узел может быть связан с несколькими другими узлами. Например, каждый узел может быть родительским для двух дочерних. Использование std::shared\_ptr позволяет нескольким частям программы безопасно владеть и использовать один и тот же объект узла дерева, не беспокоясь о его освобождении, пока все ссылки на этот узел не будут уничтожены. Это особенно полезно в случае с деревьями, где один узел может быть использован в нескольких контекстах (например, в левой или правой ветке).
3. **Предотвращение ошибок при удалении объектов**: В традиционном C++ без умных указателей можно столкнуться с ситуацией, когда объект удаляется несколько раз (например, если один и тот же указатель передан в несколько мест), что приводит к неопределенному поведению. std::shared\_ptr решает эту проблему, гарантируя, что объект будет удален только один раз, когда на него больше не будет ссылок.
4. **Простота и безопасность**: Использование std::shared\_ptr упрощает код и делает его более читаемым, так как не нужно вручную отслеживать, когда объект должен быть удален. В случае с деревом это особенно полезно, поскольку дерево имеет сложную структуру и удаление элементов вручную может быть очень трудоемким и подверженным ошибкам процессом.

Таким образом, использование std::shared\_ptr в данном контексте позволяет значительно упростить управление памятью, повысить безопасность кода и избежать утечек памяти или ошибок, связанных с многократным удалением объектов.



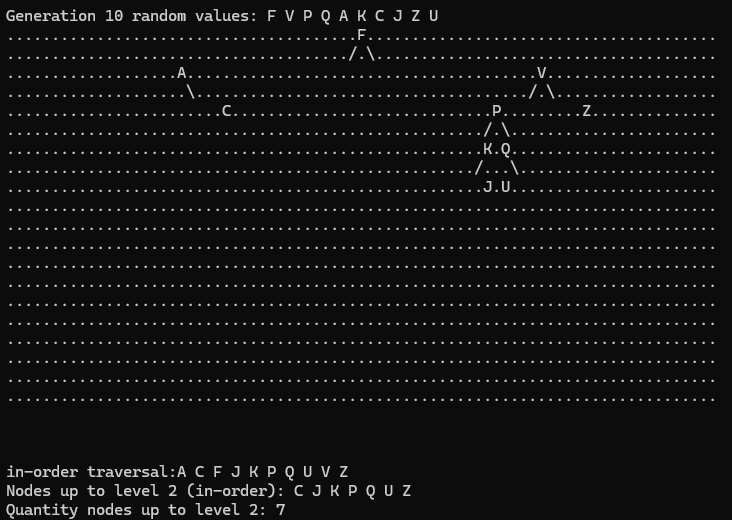


Рисунок -2. Авто-генерация дерева



Рисунок 3. Ввод вручную

# 

# **Оценка временной сложности алгоритма**

**Временная сложность для создания дерева:**

Добавление одного узла в дерево:

**Временная сложность**: зависит от высоты дерева.

* + Для сбалансированного дерева: O(logn), где n — количество узлов в дереве.
  + Для несбалансированного дерева (в худшем случае, например, если элементы добавляются по порядку): O(n).

Добавление k узлов в дерево:

**Временная сложность**: O(k⋅h), где h — средняя высота дерева.

* + В среднем (для сбалансированного дерева): O(k⋅logn).
  + В худшем случае (для несбалансированного дерева): O(k⋅n).

Генерация случайных значений и добавление:

* Генерация nnn случайных значений с уникальностью: O(n) для вставки в std::unordered\_set (в среднем).
* Добавление каждого значения в дерево: O(log n) в среднем.

**Итого**: O(n logn) в среднем.

**Временная сложность для обработки дерева:**

**Обход в симметричном порядке (**inOrderTraversal **и** inOrderTraversalHelper**)**

Обход дерева посещает каждый узел ровно один раз:

* **Временная сложность**: O(n), где n — количество узлов.

**Получение узлов со 2-го уровня и ниже (**getNodesFromSecondLevelToBottom**)**

* **Получение высоты дерева (**getHeight**)**:
  + Рекурсивный вызов для каждого узла дерева.
  + **Временная сложность**: O(n).
* **Сбор узлов начиная со 2-го уровня (**collectInOrderNodesFromLevelToBottom**)**:
  + Посещение каждого узла: O(n).

**Итого**: O(n).

**Временная сложность для вывода дерева:**

**Графический вывод дерева (**OutTree **и** OutNodes**)**

1. Очистка экрана (clrscr):
   * O(maxrow⋅maxcol).
2. Рекурсивное заполнение дерева (OutNodes):
   * Посещение каждого узла: O(n).

**Итого**: O(n).

**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).

https://en.cppreference.com/w/cpp/memory/shared\_ptr

**Приложение. Исходные тексты программ.**

#include <iostream>

#include <random>

#include "Tree.h"

#include "TreeNode.h"

int main()

{

int choise = 0;

Tree<char> 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, 65, 90);

break;

default:

system("cls");

break;

}

}

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

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

auto nodes = tree.getNodesFromSecondLevelToBottom();

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

for (char 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 <unordered\_set>

#include <iomanip>

#include <queue>

#include <cmath>

#include <random>

#include "TreeNode.h"

template <typename T>

class Tree

{

private:

const int maxrow = 20;

const int maxcol = 80;

const int offset = 40;

std::shared\_ptr<TreeNode<T>> root;

int nodeCounter;

char\*\* SCREEN;

public:

Tree();

Tree(const Tree&) = delete;

Tree(Tree&&) = delete;

void addNode(int value);

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

void inOrderTraversal() const;

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

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

void OutTree();

void clrscr();

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

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

std::vector<T> getNodesFromSecondLevelToBottom();

void manualInput();

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

~Tree();

private:

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

};

template <typename T>

Tree<T>::Tree() : root(nullptr), nodeCounter(0)

{

SCREEN = new char\* [maxrow];

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

SCREEN[i] = new char[maxcol];

}

template <typename T>

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

{

if (!node)

{

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

return;

}

if (value < node->value)

addNodeHelper(node->left, value);

else if (value > node->value)

addNodeHelper(node->right, value);

}

template <typename T>

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

{

if (node)

{

inOrderTraversalHelper(node->left);

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

inOrderTraversalHelper(node->right);

}

}

template <typename T>

void Tree<T>::printTreeHelper(std::ostream& sys, std::shared\_ptr<TreeNode<T>> 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<T>>> 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<T>> 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;

}

}

template <typename T>

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

{

if (!node)

return 0;

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

return 1 + std::max(leftHeight, rightHeight);

}

template <typename T>

void Tree<T>::addNode(int value)

{

addNodeHelper(root, value);

}

template <typename T>

void Tree<T>::inOrderTraversal() const

{

inOrderTraversalHelper(root);

std::cout << std::endl;

}

template <typename T>

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

{

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

SCREEN[r - 1][c - 1] = 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));

}

}

template <typename T>

void Tree<T>::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";

}

template <typename T>

void Tree<T>::clrscr()

{

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

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

}

template <typename T>

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

{

if (!node || currentLevel > maxLevel)

return;

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

if (currentLevel >= startLevel)

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

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

}

template <typename T>

std::vector<T> Tree<T>::getNodesFromSecondLevelToBottom()

{

std::vector<T> result;

int maxDepth = getHeight(root);

int startLevel = 2;

collectInOrderNodesFromLevelToBottom(root, 0, startLevel, maxDepth - 1, result);

return result;

}

template <typename T>

void Tree<T>::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)

{

T value;

std::cin >> value;

addNode(value);

}

}

template <typename T>

void Tree<T>::generateRandomValues(int n, int minValue, int maxValue)

{

std::unordered\_set<char> uniqueNumbers;

std::random\_device rd;

std::mt19937 gen(rd());

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

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

while (uniqueNumbers.size() != n)

uniqueNumbers.insert(distrib(gen));

for (char num : uniqueNumbers)

{

std::cout << num << " ";

addNode(num);

}

std::cout << std::endl;

}

template <typename T>

Tree<T>::~Tree()

{

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

delete[] SCREEN[i];

delete[] SCREEN;

}

**файл “TreeNode.h”**

#pragma once

#include <memory>

template <typename T>

class TreeNode

{

public:

T value;

std::shared\_ptr<TreeNode<T>> left, right;

TreeNode(T value);

TreeNode(const TreeNode&) = delete;

TreeNode(TreeNode&&) = delete;

~TreeNode();

};

template<typename T>

TreeNode<T>::TreeNode(T value) : value(value), left(nullptr), right(nullptr) {}

template<typename T>

TreeNode<T>::~TreeNode() {}