**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»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Шарпинский Д. А. |  |
| Преподаватель | Манирагена В. |  |

Санкт-Петербург

2024

**Цель работы**

Исследование алгоритмов для работы с троичным деревом

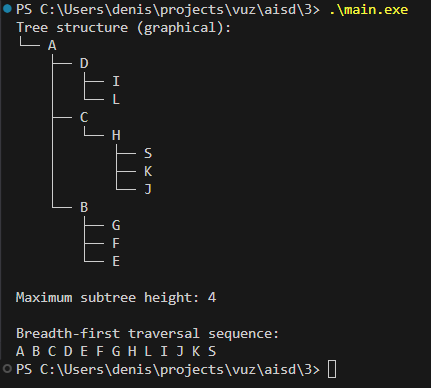
**Задание на работу с деревьями**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № варианта | Вид дерева | Разметка | Способ обхода | Что надо вычислить |
| 33 | Троичное | Прямая | В ширину | Высоту самого мощного поддерева для корня. |

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

По лабораторным работам №1 и №2 было принято решение о реализации лабораторной работе с использованием классов для обеспечения удобства написания кода.

**Тестовый пример**



Оценки временной сложности каждой функции обхода дерева, использованной в программе: создание дерева, обработка, вывод.

Создание дерева имеет сложность O(N)

Обработка дерева также имеет сложность O(N)

Вывод дерева аналогично имеет сложность O(N), т.к. при выводе мы проходим по каждому из узлов.

(где N — количество узлов в дереве)

**Ответы на контрольные вопросы:**

1. Чем отличаются алгоритмы для разных обходов дерева?

Алгоритмы обхода дерева различаются порядком обработки узлов:

* Обход в глубину (DFS):
  + *Прямой (pre-order)*: сначала обрабатывается текущий узел, затем его потомки.
  + *Симметричный (in-order)*: сначала обрабатываются потомки (слева направо), затем текущий узел.
  + *Обратный (post-order)*: сначала обрабатываются все потомки, затем текущий узел.
* Обход в ширину (BFS):
  + Узлы обрабатываются уровень за уровнем, начиная с корня. Используется очередь для хранения узлов текущего уровня.

Различия:

* DFS больше подходит для задач, где нужно углубляться в дерево (например, копирование структуры или вычисление значений поддеревьев).
* BFS полезен для задач, связанных с уровнями дерева (например, нахождение минимальной глубины или визуализация уровней).

2. Нужно ли сочетать ввод данных для построения дерева с его обходом?

Нет, ввод данных для построения дерева и его обход лучше разделять:

* Аргументы "за разделение":
  + Позволяет использовать дерево для разных задач (например, обход, модификация, удаление узлов) независимо от способа его построения.
  + Упрощает тестирование, так как можно проверять отдельно построение и алгоритмы обхода.
* Когда можно сочетать:
  + Если задача специфична и предполагает одноразовую обработку данных во время их ввода (например, вычисление суммы значений во время построения).

Разделение этих процессов делает код более универсальным и гибким.

3. Можно ли считать примененные алгоритмы обхода эффективными?

* Обход в ширину (BFS):
  + Использует очередь для хранения узлов, что делает его временную сложность O(n)O(n)O(n), где nnn — количество узлов.
  + По памяти: требует хранения всех узлов текущего уровня, что в худшем случае занимает O(w)O(w)O(w), где www — ширина дерева.
* Обход в глубину (DFS):
  + Временная сложность также O(n)O(n)O(n), так как каждый узел обрабатывается один раз.
  + По памяти: требуется стек вызовов для рекурсии, что занимает O(h)O(h)O(h), где hhh — высота дерева.

Алгоритмы считаются эффективными, так как имеют линейную временную сложность относительно числа узлов.

4. Нужно ли создавать отдельные классы для узла и дерева, или можно ограничиться одним универсальным, рассматривая любой узел как корень некоторого поддерева?

Это зависит от задачи:

* Отдельные классы для узла и дерева:
  + Рекомендуются в большинстве случаев, так как дерево содержит дополнительные свойства (например, количество узлов, указатель на корень).
  + Упрощает управление деревом в целом (например, добавление новых узлов, поиск, балансировка).
  + Обеспечивает лучшую читаемость и разделение ответственности.
* Единый класс "Узел":
  + Уместен для простых задач, где дерево — это просто набор связанных узлов.
  + Каждый узел можно рассматривать как корень поддерева, что позволяет реализовать операции на поддеревьях.

**Исходный текст программы**

#include <iostream>

#include <fstream>

#include <sstream>

#include <unordered\_map>

#include <queue>

#include <iomanip>

#include <string>

#include <vector>

#include <algorithm>

using namespace std;

/\*\*

\* @class Node

\* @brief Класс, представляющий узел троичного дерева.

\*/

class Node {

public:

char value; ///< Значение узла.

Node\* left; ///< Указатель на левого потомка.

Node\* middle; ///< Указатель на среднего потомка.

Node\* right; ///< Указатель на правого потомка.

Node\* parent; ///< Указатель на родительский узел.

/\*\*

\* @brief Конструктор узла дерева.

\* @param val Значение узла.

\* @param parentNode Указатель на родительский узел.

\*/

Node(char val, Node\* parentNode = nullptr)

: value(val), left(nullptr), middle(nullptr), right(nullptr), parent(parentNode) {}

};

/\*\*

\* @class TernaryTree

\* @brief Класс, представляющий троичное дерево.

\*/

class TernaryTree {

public:

Node\* root; ///< Указатель на корневой узел дерева.

size\_t nodeCount; ///< Общее количество узлов в дереве.

/\*\*

\* @brief Конструктор троичного дерева.

\*/

TernaryTree() : root(nullptr), nodeCount(0) {}

/\*\*

\* @brief Добавляет новый узел в дерево.

\* @param parent Указатель на родительский узел.

\* @param val Значение нового узла.

\* @param position Позиция нового узла относительно родителя ("left", "middle", "right").

\*/

Node\* addNode(Node\* parent, char val, const string& position) {

Node\* newNode = new Node(val, parent); // Передаем указатель на родителя в конструктор узла.

if (position == "left") {

parent->left = newNode;

} else if (position == "middle") {

parent->middle = newNode;

} else if (position == "right") {

parent->right = newNode;

}

nodeCount++; // Увеличиваем счетчик узлов.

return newNode;

}

/\*\*

\* @brief Выполняет обход дерева в ширину.

\* @return Вектор символов, представляющий последовательность узлов при обходе в ширину.

\*/

vector<char> breadthFirstTraversal() {

vector<char> result;

result.reserve(nodeCount); // Резервируем память под количество узлов.

if (!root) return result;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* current = q.front();

q.pop();

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

if (current->left) q.push(current->left);

if (current->middle) q.push(current->middle);

if (current->right) q.push(current->right);

}

return result;

}

/\*\*

\* @brief Рекурсивно выводит дерево в консоль в графическом виде с использованием стрелочек.

\* @param node Указатель на текущий узел.

\* @param prefix Префикс для текущей строки, используется для создания структуры дерева.

\* @param isLast Флаг, указывающий, является ли узел последним потомком.

\*/

void printTree(Node\* node, const string& prefix = "", bool isLast = true) {

if (!node) return;

cout << prefix;

// Выводим соединение: +-- для последнего ребенка, |-- для других

cout << (isLast ? "+-- " : "|-- ");

// Выводим значение узла

cout << node->value << endl;

// Формируем префикс для следующего уровня

string newPrefix = prefix + (isLast ? " " : "| ");

// Рекурсивно выводим потомков

if (node->left || node->middle || node->right) {

printTree(node->right, newPrefix, node->middle == nullptr && node->right == nullptr);

printTree(node->middle, newPrefix, node->right == nullptr);

printTree(node->left, newPrefix, true);

}

}

/\*\*

\* @brief Рекурсивно вычисляет высоту самого мощного поддерева для заданного узла.

\* @param node Указатель на текущий узел.

\* @return Высота поддерева.

\*/

int calculateSubtreeHeight(Node\* node) {

if (!node) return 0;

int leftHeight = calculateSubtreeHeight(node->left);

int middleHeight = calculateSubtreeHeight(node->middle);

int rightHeight = calculateSubtreeHeight(node->right);

return 1 + max({leftHeight, middleHeight, rightHeight});

}

/\*\*

\* @brief Возвращает высоту самого мощного поддерева для корня.

\* @return Высота поддерева.

\*/

int getMaxSubtreeHeight() {

return calculateSubtreeHeight(root);

}

/\*\*

\* @brief Загружает дерево из файла.

\* @param filename Имя файла, содержащего описание дерева.

\*/

void loadTreeFromFile(const string& filename) {

ifstream file(filename);

unordered\_map<string, Node\*> nodes; // Для быстрого поиска узлов по имени.

string line;

getline(file, line); // Пропускаем заголовок

while (getline(file, line)) {

stringstream ss(line);

string value, parentValue, position;

getline(ss, value, ',');

getline(ss, parentValue, ',');

getline(ss, position, ',');

Node\* parentNode = (parentValue == "null") ? nullptr : nodes[parentValue];

Node\* newNode = (parentNode == nullptr)

? (root = new Node(value[0])) // Устанавливаем корень

: addNode(parentNode, value[0], position);

nodes[value] = newNode; // Добавляем узел в карту

}

file.close();

}

/\*\*

\* @brief Добавляет узлы в дерево через ввод с клавиатуры.

\*/

void addNodesFromInput() {

string value, parentValue, position;

Node\* parentNode = nullptr;

while (true) {

cout << "Enter node value (or 'exit' to stop): ";

cin >> value;

if (value == "exit") break;

cout << "Enter parent value (or 'null' for root): ";

cin >> parentValue;

if (parentValue != "null") {

// Если родительский узел не равен "null", то ищем его

parentNode = findNodeByValue(parentValue);

if (!parentNode) {

cout << "Parent node not found!" << endl;

continue;

}

} else {

// Если родитель "null", это означает, что это корень дерева

if (root == nullptr) {

root = new Node(value[0]); // Создаем корень дерева

cout << "Root node created: " << value << endl;

continue; // Переходим к следующему узлу

} else {

cout << "Root already exists!" << endl;

continue; // Корень уже существует, продолжаем добавление других узлов

}

}

cout << "Enter position ('left', 'middle', 'right'): ";

cin >> position;

if (position == "left" || position == "middle" || position == "right") {

addNode(parentNode, value[0], position);

} else {

cout << "Invalid position!" << endl;

}

}

}

/\*\*

\* @brief Рекурсивно ищет узел по значению.

\* @param value Значение узла.

\* @return Указатель на узел, если найден, иначе nullptr.

\*/

Node\* findNodeByValue(const string& value) {

if (!root) return nullptr;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* current = q.front();

q.pop();

// Сравниваем полное значение, а не только первый символ

if (current->value == value[0]) return current;

if (current->left) q.push(current->left);

if (current->middle) q.push(current->middle);

if (current->right) q.push(current->right);

}

return nullptr; // Если узел не найден

}

};

/\*\*

\* @brief Точка входа в программу. Создает дерево, добавляет узлы, выводит дерево

\* и выполняет обход в ширину.

\* @return Код завершения программы.

\*/

int main() {

TernaryTree tree;

// Выбираем способ ввода дерева

string choice;

cout << "Choose input method (file/input): ";

cin >> choice;

if (choice == "file") {

string filename = "tree.csv";

tree.loadTreeFromFile(filename);

} else if (choice == "input") {

tree.addNodesFromInput();

} else {

cout << "Invalid choice!" << endl;

return 1;

}

// Выводим дерево

cout << "Tree structure (graphical):" << endl;

tree.printTree(tree.root);

// Высота самого мощного поддерева

int maxHeight = tree.getMaxSubtreeHeight();

cout << "\nMaximum subtree height: " << maxHeight << endl;

// Выполняем обход в ширину

vector<char> traversal = tree.breadthFirstTraversal();

cout << "\nBreadth-first traversal sequence:" << endl;

for (char c : traversal) {

cout << c << " ";

}

cout << endl;

return 0;

}