Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Пермский национальный** **исследовательский политехнический университет»**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

направление подготовки: 09.03.01– «Информатика и вычислительная техника»

**Лабораторная работа**

**по теме**

**«Бинарные деревья»**

Выполнил студент гр. ИВТ-24-1б

Оглезнев Никита Михайлович

Проверил:

доц. каф. ИТАС

Яруллин Денис Владимирович

(оценка) (подпись)

(дата)

г. Пермь, 2024

1. **Постановка задачи**

Целью данной работы является разработать приложение для работы с бинарными деревьями, включающее создание идеально сбалансированного бинарного дерева, визуализацию, выполнение операций обработки (обходы, балансировка), преобразование идеально сбалансированного дерева в дерево поиска (BST), отображение результатов операций.

Реализовать графический интерфейс (GUI) на QT, позволяющий:

* Вручную добавлять/удалять узлы.
* Автоматически балансировать дерево.
* Отображать результаты обходов (прямой, симметричный, обратный).
* Визуализировать структуру дерева.

1. **Анализ задачи**

Для реализации поставленных задач код разделен на логические компоненты:

* Класс Node - узел дерева (ключ, указатели на левого и правого потомка).
* Класс BinaryTree – реализация основных операций (вставка, удаление, поиск, обходы, балансировка).
* Класс TreeWidget - отрисовка дерева на экране.
* Класс MainWindow - создание GUI, связка логики с интерфейсом

1. **Формирование идеально сбалансированного дерева**

Метод balance(). Собирает ключи в отсортированный список через обход in-order. Рекурсивно строит сбалансированное дерево из середины списка.

1. **Визуализация дерева**

Класс TreeWidget. Рассчитывает позиции узлов на основе их уровня в дереве. Рисует узлы и связи между ними с помощью QPainter.

1. **Обработка дерева**

Обходы (PreOrder, InOrder, PostOrder). Балансировка - вызывается метод balance(), который перестраивает дерево.

1. **Преобразование в дерево поиска**

В коде изначально используется BST. Метод insert() размещает узлы по правилам BST, балансировка сохраняет свойства BST, делая его идеально сбалансированным.

1. **Интерфейс пользователя**

Поля ввода и кнопки для управления деревом. Выпадающий список для выбора типа обхода. Отображение результатов обходов.

1. **Код программы**

#include <QApplication>

#include <QWidget>

#include <QPushButton>

#include <QLineEdit>

#include <QVBoxLayout>

#include <QHBoxLayout>

#include <QPainter>

#include <QLabel>

#include <QComboBox>

#include <QMap>

class Node {

public:

    int key;

    Node \*left, \*right;

    Node(int key) : key(key), left(nullptr), right(nullptr) {}

};

class BinaryTree {

public:

    enum TraversalType { PreOrder, InOrder, PostOrder };

    BinaryTree() : root(nullptr) {}

    ~BinaryTree() { clearTree(root); }

    void insert(int key) { root = insert(root, key); }

    void deleteNode(int key) { root = deleteNode(root, key); }

    Node\* search(int key) { return search(root, key); }

    QList<int> traverse(TraversalType type) {

        QList<int> result;

        switch (type) {

        case PreOrder: preOrder(root, result); break;

        case InOrder: inOrder(root, result); break;

        case PostOrder: postOrder(root, result); break;

        }

        return result;

    }

    void balance() {

        QList<int> keys;

        inOrder(root, keys);

        clearTree(root);

        root = buildBalancedTree(keys, 0, keys.size() - 1);

    }

    Node\* getRoot() { return root; }

private:

    Node\* root;

    Node\* insert(Node\* node, int key) {

        if (!node) return new Node(key);

        if (key < node->key)

            node->left = insert(node->left, key);

        else

            node->right = insert(node->right, key);

        return node;

    }

    Node\* deleteNode(Node\* node, int key) {

        if (!node) return nullptr;

        if (key < node->key) {

            node->left = deleteNode(node->left, key);

        } else if (key > node->key) {

            node->right = deleteNode(node->right, key);

        } else {

            if (!node->left) {

                Node\* temp = node->right;

                delete node;

                return temp;

            } else if (!node->right) {

                Node\* temp = node->left;

                delete node;

                return temp;

            }

            Node\* temp = minValueNode(node->right);

            node->key = temp->key;

            node->right = deleteNode(node->right, temp->key);

        }

        return node;

    }

    Node\* minValueNode(Node\* node) {

        Node\* current = node;

        while (current && current->left) current = current->left;

        return current;

    }

    Node\* search(Node\* node, int key) {

        if (!node || node->key == key) return node;

        return search(key < node->key ? node->left : node->right, key);

    }

    void clearTree(Node\* node) {

        if (node) {

            clearTree(node->left);

            clearTree(node->right);

            delete node;

        }

    }

    void preOrder(Node\* node, QList<int>& result) {

        if (node) {

            result.append(node->key);

            preOrder(node->left, result);

            preOrder(node->right, result);

        }

    }

    void inOrder(Node\* node, QList<int>& result) {

        if (node) {

            inOrder(node->left, result);

            result.append(node->key);

            inOrder(node->right, result);

        }

    }

    void postOrder(Node\* node, QList<int>& result) {

        if (node) {

            postOrder(node->left, result);

            postOrder(node->right, result);

            result.append(node->key);

        }

    }

    Node\* buildBalancedTree(const QList<int>& keys, int start, int end) {

        if (start > end) return nullptr;

        int mid = (start + end) / 2;

        Node\* node = new Node(keys[mid]);

        node->left = buildBalancedTree(keys, start, mid - 1);

        node->right = buildBalancedTree(keys, mid + 1, end);

        return node;

    }

};

class TreeWidget : public QWidget {

public:

    TreeWidget(QWidget\* parent = nullptr) : QWidget(parent) {}

    void setTree(BinaryTree\* tree) {

        this->tree = tree;

        update();

    }

protected:

    void paintEvent(QPaintEvent\*) override {

        QPainter painter(this);

        painter.setRenderHint(QPainter::Antialiasing);

        painter.fillRect(rect(), Qt::white);

        if (!tree || !tree->getRoot()) return;

        QMap<Node\*, QPoint> positions;

        calculatePositions(tree->getRoot(), positions, width() / 2, 20, width() / 4);

        painter.setPen(Qt::black);

        for (auto it = positions.begin(); it != positions.end(); ++it) {

            Node\* node = it.key();

            QPoint pos = it.value();

            if (node->left && positions.contains(node->left)) {

                QPoint leftPos = positions[node->left];

                painter.drawLine(pos, leftPos);

            }

            if (node->right && positions.contains(node->right)) {

                QPoint rightPos = positions[node->right];

                painter.drawLine(pos, rightPos);

            }

        }

        painter.setBrush(Qt::lightGray);

        for (auto it = positions.begin(); it != positions.end(); ++it) {

            QPoint pos = it.value();

            painter.drawEllipse(pos, 20, 20);

            painter.drawText(pos.x() - 15, pos.y() - 10, 30, 20,

                             Qt::AlignCenter, QString::number(it.key()->key));

        }

    }

private:

    BinaryTree\* tree = nullptr;

    void calculatePositions(Node\* node, QMap<Node\*, QPoint>& positions,

                            int x, int y, int hOffset) {

        if (!node) return;

        positions[node] = QPoint(x, y);

        calculatePositions(node->left, positions, x - hOffset, y + 60, hOffset / 2);

        calculatePositions(node->right, positions, x + hOffset, y + 60, hOffset / 2);

    }

};

class MainWindow : public QWidget {

public:

    MainWindow() {

        setupUI();

        connectButtons();

    }

private:

    BinaryTree tree;

    TreeWidget\* treeWidget;

    QLineEdit\* keyEdit;

    QLabel\* traversalResult;

    void setupUI() {

        QVBoxLayout\* layout = new QVBoxLayout(this);

        layout->setContentsMargins(5, 5, 5, 5);

        QWidget\* controlPanel = new QWidget;

        controlPanel->setFixedHeight(50);

        QHBoxLayout\* controls = new QHBoxLayout(controlPanel);

        controls->setContentsMargins(0, 0, 0, 0);

        controls->setSpacing(5);

        keyEdit = new QLineEdit;

        keyEdit->setFixedHeight(30);

        QPushButton\* insertBtn = new QPushButton("Вставить");

        insertBtn->setObjectName("Insert");

        insertBtn->setFixedSize(80, 30);

        QPushButton\* deleteBtn = new QPushButton("Удалить");

        deleteBtn->setObjectName("Delete");

        deleteBtn->setFixedSize(80, 30);

        QPushButton\* balanceBtn = new QPushButton("Балансировать");

        balanceBtn->setObjectName("Balance");

        balanceBtn->setFixedSize(120, 30);

        QComboBox\* traversalCombo = new QComboBox;

        traversalCombo->setObjectName("TraversalCombo");

        traversalCombo->addItems({"Прямой", "Симметричный", "Обратный"});

        traversalCombo->setFixedHeight(30);

        traversalResult = new QLabel("Результат обхода: ");

        traversalResult->setFixedHeight(20);

        controls->addWidget(keyEdit);

        controls->addWidget(insertBtn);

        controls->addWidget(deleteBtn);

        controls->addWidget(balanceBtn);

        controls->addWidget(traversalCombo);

        treeWidget = new TreeWidget;

        treeWidget->setTree(&tree);

        treeWidget->setMinimumSize(800, 600);

        layout->addWidget(controlPanel);

        layout->addWidget(traversalResult);

        layout->addWidget(treeWidget);

    }

    void connectButtons() {

        connect(findChild<QPushButton\*>("Insert"), &QPushButton::clicked,

                this, &MainWindow::insertNode);

        connect(findChild<QPushButton\*>("Delete"), &QPushButton::clicked,

                this, &MainWindow::deleteNode);

        connect(findChild<QPushButton\*>("Balance"), &QPushButton::clicked,

                this, &MainWindow::balanceTree);

        connect(findChild<QComboBox\*>("TraversalCombo"), QOverload<int>::of(&QComboBox::currentIndexChanged),

                this, &MainWindow::updateTraversalResult);

    }

    void insertNode() {

        bool ok;

        int key = keyEdit->text().toInt(&ok);

        if (ok) {

            tree.insert(key);

            treeWidget->update();

            updateTraversalResult();

        }

    }

    void deleteNode() {

        bool ok;

        int key = keyEdit->text().toInt(&ok);

        if (ok) {

            tree.deleteNode(key);

            treeWidget->update();

            updateTraversalResult();

        }

    }

    void balanceTree() {

        tree.balance();

        treeWidget->update();

        updateTraversalResult();

    }

    void updateTraversalResult() {

        QComboBox\* combo = findChild<QComboBox\*>("TraversalCombo");

        BinaryTree::TraversalType type = static\_cast<BinaryTree::TraversalType>(combo->currentIndex());

        QList<int> result = tree.traverse(type);

        QString text;

        for (int key : result) text += QString::number(key) + " ";

        traversalResult->setText("Результат обхода: " + text);

    }

};

int main(int argc, char \*argv[]) {

    QApplication app(argc, argv);

    MainWindow window;

    window.setWindowTitle("binary tree lab");

    window.show();

    return app.exec();

}

1. **UML и результат**



