Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«**Пермский национальный исследовательский политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЕТ**

Дисциплина: «Информатика»

Тема: «Деревья»

Семестр 2

Выполнил работу

Студент группы РИС-22-1Б

Баженов Т.И.

Проверил

Доцент кафедры ИТАС

Полякова О.А.

Г. Пермь-2023

# Постановка задачи

Требуется реализовать алгоритмы для собственного варианта бинарного дерева поиска, имеющего не менее трёх уровней.

Задания:

1. Необходимо реализовать функции для редактирования дерева:

* Вставка узла.
* Удаление узла.
* Поиск элемента по ключу.

1. Реализовать алгоритмы обхода дерева:

* Прямой
* Симметричный
* Обратный

1. Реализовать алгоритм балансировки дерева.
2. Визуализация дерева должна быть выполнена с использованием любой доступной графической библиотеки – SFML, SDL, OpenGL и подобных.
3. Пользовательский интерфейс по усмотрению разработчика - с условием кроссплатформенности (поощряется использование Qt или иных фреймворков).

# Класс Tree

*Class* **Tree** {

int data;

Tree\* left;

Tree\* right;

Tree\* parent;

**Tree**(int val, Tree\* par, Tree\* l, Tree\* r) {

data = val;

left = l;

right = r;

parent = par;

}

**Tree**(int val) {

*this*->data = val;

*this*->left = *nullptr*;

*this*->right = *nullptr*;

*this*->parent = *nullptr*;

}

**Tree**() {

data = -2000;

left = *nullptr*;

right = *nullptr*;

parent = *nullptr*;

}

Tree\* **minTree**(Tree\* tree) {

*if* (tree->left == *nullptr*) *return* *this*;

*return* tree->left->minTree(*tree->left*);

}

Tree\* **maxTree**(Tree\* tree) {

*if* (tree->right == *nullptr*) *return* *this*;

*return* tree->right->minTree(*tree->right*);

}

void **insert**(int val) {

Tree\* temp\_tree = *this*;

*while* (temp\_tree != *nullptr*) {

*if* (val > temp\_tree->data) {

*if* (temp\_tree->right != *nullptr*)

temp\_tree = temp\_tree->right;

*else* {

Tree\* tmp = *new* Tree(val);

tmp->parent = temp\_tree;

temp\_tree->right = tmp;

*break*;

}

}

*else* *if* (val < temp\_tree->data) {

*if* (temp\_tree->left != *nullptr*)

temp\_tree = temp\_tree->left;

*else* {

Tree\* tmp = *new* Tree(val);

tmp->parent = temp\_tree;

temp\_tree->left = tmp;

*break*;

}

}

*else*

*break*;

}

}

Tree\* **find**(int val) {

*if* (*this* == *nullptr*)

*return* *nullptr*;

*if* (*this*->data == val)

*return* *this*;

*else* *if* (val < *this*->data)

*return* *this*->left->find(val);

*else* *if* (val > *this*->data)

*return* *this*->right->find(val);

}

bool **erase**(int val){

Tree\* node = *this*->find(val);

*if* (node == *nullptr*) *return* *false*;

*if* ( (node->left == *nullptr*) && (node->right == *nullptr*) ) {

Tree\* node\_par = node->parent;

*if* (node\_par->left == node)

node->parent->left = *nullptr*;

*else*

node->parent->right = *nullptr*;

*delete* node;

}

*else* *if* ( (node->left == *nullptr* && node->right != *nullptr*) || (node->left != *nullptr* && node->right == *nullptr*) ) {

Tree\* node\_par = node->parent;

*if* (node->left == *nullptr*) {

*if* (node\_par->left == node)

node->parent->left = node->right;

*else*

node->parent->right = node->right;

node->right->parent = node->parent;

}

*else* {

*if* (node\_par->left == node)

node->parent->left = node->left;

*else*

node->parent->right = node->left;

node->left->parent = node->parent;

}

*delete* node;

}

*else* {

Tree\* r\_tree\_min = node->right->minTree(*node->right*);

*if* (r\_tree\_min->left == *nullptr* && r\_tree\_min->right == *nullptr*) {

int tmp = r\_tree\_min->data;

*this*->erase(r\_tree\_min->data);

node->data = tmp;

}

*else* {

int tmp = r\_tree\_min->data;

*this*->erase(r\_tree\_min->data);

node->data = tmp;

}

}

*return* *true*;

}

void **clear**() {

*if*(*this* != *nullptr*) {

*this*->left->clear();

*this*->right->clear();

*delete* *this*;

*return*;

}

*else*

*return*;

}

void **balance**() {

std::vector<int> values;

inorderTraversal(*this*, *values*);

Tree\* balancedTree = buildBalancedTree(values, 0, values.size() - 1);

\**this* = \*balancedTree;

}

void **inorderTraversal**(Tree\* node, std::vector<int>& values){

*if* (node == *nullptr*)

*return*;

inorderTraversal(*node->left*, *values*);

values.push\_back(node->data);

inorderTraversal(*node->right*, *values*);

}

Tree\* **buildBalancedTree**(*const* std::vector<int>& values, int start, int end) {

*if* (start > end)

*return* *nullptr*;

int mid = (start + end) / 2;

Tree\* newNode = *new* Tree(values[mid]);

newNode->left = buildBalancedTree(values, start, mid - 1);

*if* (newNode->left != *nullptr*)

newNode->left->parent = newNode;

newNode->right = buildBalancedTree(values, mid + 1, end);

*if* (newNode->right != *nullptr*)

newNode->right->parent = newNode;

*return* newNode;

}

};

*struct* **BinaryTree** {

Tree\* root;

**BinaryTree**() {

root = *nullptr*;

}

**BinaryTree**(int val) {

root = *new* Tree(val, *nullptr*, *nullptr*, *nullptr*);

}

void **insert**(int val) {

*this*->root->insert(val);

}

Tree\* **find**(int val) {

*return* (*this*->root->find(val));

}

bool **erase**(int val) {

*return* (*this*->root->erase(val));

}

void **balance**() {

*if*(*this*->root != *nullptr*)

*this*->root->balance();

}

void **clear**() {

*this*->root->clear();

*this*->root = *nullptr*;

}

# };UML-диаграмма

# Работа программы



