Санкт-Петербургский государственный политехнический университет
Институт машиностроения материалов и транспорта

Курсовой проект
Алгоритм «Красно-черное дерево»

Студент, гр. 3331506/20101 Ибрагимов А.Ф.

Преподаватель Ананьевский М.С.

Санкт-Петербург
2025 г

Оглавление

Введение 3

1. Определение общей структуры кода 3

2. Реализация кода 4

Заключение 8

Список литературы 9

Введение

Алгоритм красно-черного дерева – это усложненный алгоритм бинарного дерева поиска. Данный алгоритм реализует структуру для хранения данных типа «ключ»-«значение» в виде бинарного дерева, для которого поиск любого элемента будет реализовываться за log(n) операций. Сложность данного алгоритма заключается в реализации балансирования дерева после вставки или удаления очередного его элемента.

1. Определение общей структуры кода

В красно-черном дереве узлы располагаются особым образом – начальный узел – корень – не имеет родителей и является всегда черным. Если у узла нет дочерних узлов – то он ссылается на «нулевой» узел, который по определению является черным. У любого красного узла должно быть 2 дочерних черных узла (в качестве черного может выступать и нулевой узел).

Помимо этих правил сохраняются также условия для обычного бинарного дерева – «ключ» левого дочернего узла должен быть меньше «ключа» родителя, а «ключ» правого больше.

Также важно – что для того чтоб дерево оставалось сбалансированным – число черный узлов от корня до любого листа (элемента без дочерних узлов) должно быть одинаковым, при этом корень не учитывается, а нулевые конечные листы считаются.

Для реализации данной структуры будет удобно ввести два класса – класс «Node» – отвечающий за конкретный узел, в котором хранится его «ключ», «значение», а также ссылки на дочерние узлы. Для реализации удобного взаимодействия с деревом введем еще один класс «tree» который будет содержать указатель на корень дерева, а также все основные функции.

2. Реализация кода

Объявление классов «Node» и «tree» представлены на рисунке 1.

```
class Node {
public:
    int key;
    bool color;
    std::string name;

    Node *left = nullptr;
    Node *right = nullptr;

    Node();
    Node(int key, std::string name);
};
```

```
class Tree {
private:
    void display_subtree(Node* node);

    Node* get_sub_min(Node *node);
    Node* get_sub_min_parent(Node *node);
    Node* get_sub_max(Node *node);

    int black_child_count(Node *node);
    int child_count(Node *node);

    void swap_node(Node *node1, Node *node2);

    void right_rotate(Node *node);
    void left_rotate(Node *node);
public:
    Node *root;

    Tree() : root(nullptr) {}

    void add_node(int key, const std::string& name);
    void delete_node(int d_key);
    void display();

    Node* search_node(int key);
    Node search(int key);
    Node* get_min();
    Node* get_max();
};
```

Рисунок 1 – Объявление классов

Реализация добавления нового узла в дерево будет осуществляться следующим образом – ищем место для вставки, вставляем туда новый узел, при необходимости производим балансировку дерева. Реализация данного алгоритма представлена на рисунке 2.

```cpp
void Tree::add_node(int key, const std::string& name) {
    Node *newNode = new Node(key, name);

    //создание корня
    if (root == nullptr) {
        root = newNode;
        root->color = 0;
        return;
    }

    Node* current = root;
    Node* parent = nullptr;
    Node* grandpa = nullptr;
    Node* uncle = nullptr;

    //поиск места для вставки узла
    while(current != nill){
        grandpa = parent;
        parent = current;
        if(grandpa == nullptr)uncle = nullptr;
        else uncle = grandpa->right == parent ? grandpa->left : grandpa->right;
        if(key == current->key) {  //замена значения если ключ одинаковый
            current->name = name;
            return;
        }
        current = key < current->key ? current->left : current->right;
    }

    //вставка
    key < parent->key ? parent->left = newNode : parent->right = newNode;

    //балансировка
    while(parent->color == 1){
        if(parent == grandpa->left){
            if(uncle != nullptr && uncle->color == 1){
                parent->color = 0;
                uncle->color = 0;
                grandpa->color = 1;
            }
            else {
                if(newNode == parent->right){
                    this->left_rotate(parent);
                }
                parent->color = 0;
                grandpa->color = 1;
                this->right_rotate(grandpa);
                Node* buffer = parent;
                parent = grandpa;
                grandpa = buffer;
            }
        }
        else{
            if(uncle != nullptr && uncle->color == 1){
                parent->color = 0;
                uncle->color = 0;
                grandpa->color = 1;
            }
            else {
                if(newNode == parent->left){
                    this->right_rotate(parent);
                }
                parent->color = 0;
                grandpa->color = 1;
                this->left_rotate(grandpa);
                Node* buffer = parent;
                parent = grandpa;
                grandpa = buffer;
            }
```

Рисунок 2 – Реализация вставки узла

Реализация удаления узла будет осуществляться следующим образом – удаляемый узел, удаляем его, при необходимости производим балансировку дерева. Реализация данного алгоритма представлена на рисунке 3.

```
void Tree::delete_node(int d_key){
    Node* current = root;
    Node* parent = nullptr;
    Node* grandpa = nullptr;
    Node* uncle = nullptr;
    Node* brother = nullptr;

    //поиск удаляемого элемента и запись его родственников
    while(current->key != d_key){
        grandpa = parent;
        parent = current;

        if(grandpa == nullptr)uncle = nullptr;
        else uncle = grandpa->right == parent ? grandpa->left : grandpa->right;
        if(parent == nullptr)brother = nullptr;
        else brother = parent->right == current ? parent->left : parent->right;
        if(d_key < current->key){
            current = current->left;
        }
        else {
            current = current->right;
        }
        if(current == nill)return;
    }


    // само удаление
    //удаление красного узла без детей
    if(this->child_count(current) == 0 && current->color == 1){
        if(current == parent->left)parent->left = nill;
        else parent->right = nill;
        return;
    }
    //удаление черного узла с 1 красным ребенком
    if(this->child_count(current) == 1 && current->color == 0 && (current->right->color == 1 || current->left->color == 1)){
        if(current->right->color == 1){
            current->key = current->right->key;
            current->name = current->right->name;
            current->right = nill;
        }
        else {
            current->key = current->left->key;
            current->name = current->left->name;
            current->left = nill;
        }
        return;
    }
    //удаление черного узла и балансировка
    if(this->child_count(current) == 0 && current->color == 0){
        if(brother->color == 1){
            parent->color = 1;
            brother->color = 1;
            if(current == parent->left)this->left_rotate(parent);
            else this->right_rotate(parent);
        }
        else{
            if(this->black_child_count(brother) == 2){
                if(parent == root){
                    brother->color = 1;
                }
                else{
                    if(parent->color == 1){
                        brother->color = 1;
                        parent->color == 0;
                    }
                    else{
                        parent->color == 1;
                        if(current == parent->left)this->left_rotate(parent);
                        else this->right_rotate(parent);
                    }
                }
            }
            else{
                if(current == parent->left){
                    if(brother->right->color == 0){
                        brother->color = 1;
                        brother->left->color = 0;
                        this->right_rotate(brother);
                        brother = parent->right;
                    }
                    brother->color = parent->color;
                    brother->right->color = 0;
                    parent->color = 0;
                    this->left_rotate(parent);
                }
                else{
                    if(brother->left->color == 0){
                        brother->color = 1;
                        brother->right->color = 0;
                        this->left_rotate(brother);
                        brother = parent->left;
                    }
                    brother->color = parent->color;
                    brother->left->color = 0;
                    parent->color = 0;
                    this->right_rotate(parent);
                }
            }
        }
        return;
    }
    //замена узла с 2 детьми на минимальный правый и рекурсивное удаление минимального правого
    if(this->child_count(current) == 2){
        Node *last = current;
        Node *min_node = get_sub_min(current->right);
        int replace_key = min_node->key;
        std::string replace_name = min_node->name;
        this->delete_node(min_node->key);
        current->key = replace_key;
        current->name = replace_name;
    }
}
```

Рисунок 3 – Реализация удаления узла

Также реализуем вспомогательные функции для удаления узла – замена значений двух узлов, правый левый поворот дерева (рисунок 4)

```
void Tree::swap_node(Node *node1, Node *node2){
    int key1 = node1->key;
    std::string name1 = node1->name;
    bool color1 = node1->color;
    node1->key = node2->key;
    node1->name = node2->name;
    node1->color = node2->color;
    node2->key = key1;
    node2->name = name1;
    node2->color = color1;
}


void Tree::right_rotate(Node *node){
    this->swap_node(node, node->left);
    Node *buffer = node->right;
    node->right = node->left;
    node->left = node->right->left;
    node->right->left = node->right->right;
    node->right->right = buffer;
}


void Tree::left_rotate(Node *node){
    this->swap_node(node, node->right);
    Node *buffer = node->left;
    node->left = node->right;
    node->right = node->left->right;
    node->left->right = node->left->left;
    node->left->left = buffer;
}
```

Рисунок 4 – Реализация замены, правого левого поворота

Также реализуем алгоритмы по поиску указателя на узел и поиску и копирование значений узла (рисунок 5).

```
Node* Tree::search_node(int f_key){
    Node *node = root;
    while(node != nill){
        if(node->key == f_key)return node;
        f_key < node->key ? node = node->left : node = node->right;
    }
    return nullptr;
}


Node Tree::search(int f_key){
    Node newNode;
    Node *node = root;
    while(node != nill){
        if(node->key == f_key){
            newNode = *node;
            return newNode;
        }
        f_key < node->key ? node = node->left : node = node->right;
    }
    return newNode;
}
```

Рисунок 5 – Реализация поиска узла

Заключение

В ходе работы реализовали алгоритм красно-черного дерева для хранения данных типа «числовой ключ» - «строка». Алгоритм сохраняет дерево сбалансированным при вставке или удалении узла.

Список литературы

1. «Красно-черные деревья» [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/330644/

2. «Красно-чёрное дерево. Свойства, принципы организации, механизм вставки.» [Электронный ресурс]. – Режим доступа: https://javarush.com/groups/posts/4165-krasno-chjernoe-derevo-svoystva-principih-organizacii-mekhanizm-vstavki

3. Пышкин Е.В. «Структуры данных и алгоритмы: реализация на C/C++». - СПб.: ФТК СПБГПУ, 2009.- 200 с., ил

4. Давыдов В.Г. Программирование и основы алгоритмизации: Учебное пособие. - М.: Высшая школа, 2003.- 447 с., ил.

5. [Давыдов, 2005] Давыдов В.Г. Технологии программирования. C++. Учебное пособие. – СПб.: БХВ-Петербург, 2005.- 672 с., ил.