Санкт-Петербургский Политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Кафедра «Мехатроника и роботостроение (при ЦНИИ РТК)»

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Красно-черное дерево. Добавление

Разработал: ст. гр. 3331506/80401 Калясова С. С.

Преподаватель Кузнецова Е. М.

Санкт-Петербург

2021 г

Содержание

Введение

Красно-чёрное дерево - двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный". Такие деревья являются наиболее часто используемыми сбалансированными деревьями двоичного поиска, и они гарантированно поддерживают вставку, удаление и поиск.

Красно-чёрное дерево используется для организации сравнимых данных, таких как фрагменты текста или числа.

Применение:

- Ява: java.util.TreeMap ,java.util.TreeSet
- C ++ STL (в большинстве реализаций): map, multimap, multiset
- Ядро Linux: полностью честный планировщик, linux / rbtree.h

Описание алгоритма

Красно-черное дерево является "достаточно сбалансированным". Т.е. путь от вершины дерева до любого листа дерева занимает примерно одинаковое время. Длины поддеревьев могут отличаться больше, чем на единицу, но никогда не больше, чем в два раза.

Для красно-черного дерева должны выполняться следующие свойства:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

При этом для удобства, листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных. Пример данного дерева представлен на рисунке 1.

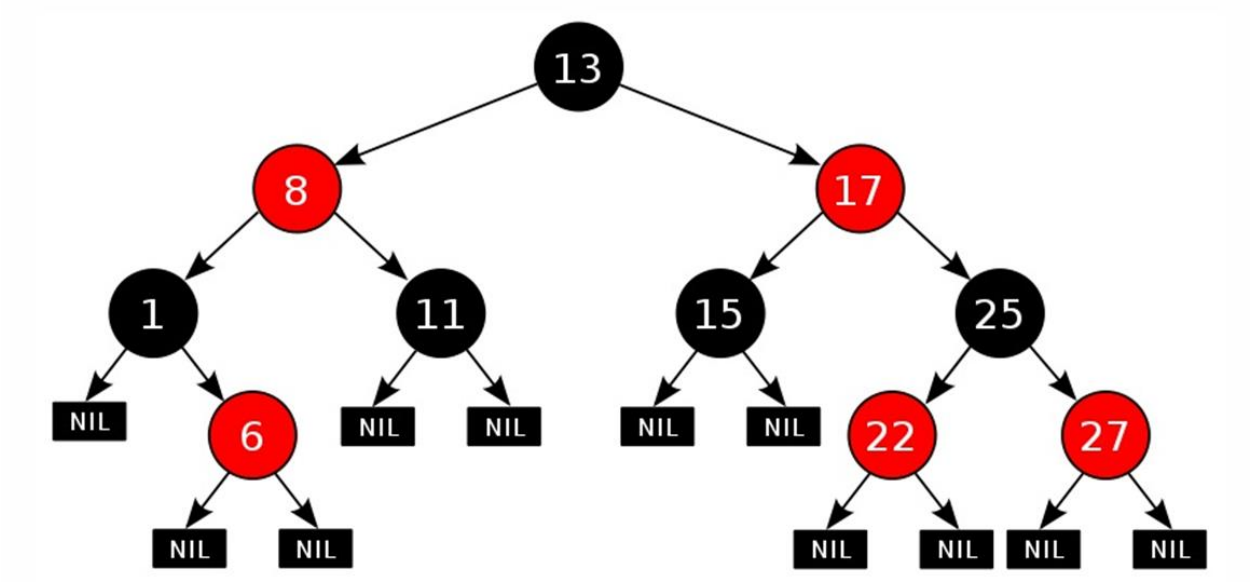

Рисунок 1 — Красно-черное дерево

При добавлении в дерево новый узел окрашивается в «красный» цвет. Далее проверяются правила, приведённые выше; если они нарушаются, дерево балансируется.

Количество черных узлов на ветви от корня до листа называется черной высотой дерева.

Сложность алгоритма

Красно-чёрные деревья имеют вычислительную сложность поиска, одинаковую с АВЛ-деревьями, а именно O(ln N). Причём поддержание структуры в «оптимальной форме» требует на порядок меньших усилий. Для данной структуры требуется память, кратная количеству элементов, следовательно, пространственная сложность алгоритма линейна — O(N), как и в случае бинарного дерева поиска.

Исследование алгоритма

Если посчитать время заполнения красно-черного дерева различным количеством данных, то получим результаты, приведенные в таблице 1.

Таблица 1- Время выполнения алгоритма

Число элементов, шт.	10	10^2	$5 \cdot 10^2$	10^3	$2 \cdot 10^3$	$5 \cdot 10^3$	$7 \cdot 10^3$	10^4
Время выполнения, мс.	24	323	546	1190	2266	4502	7397	11970

На рисунке 2 приведен график в логарифмическом масштабе времени выполнения алгоритма в зависимости от количества данных.

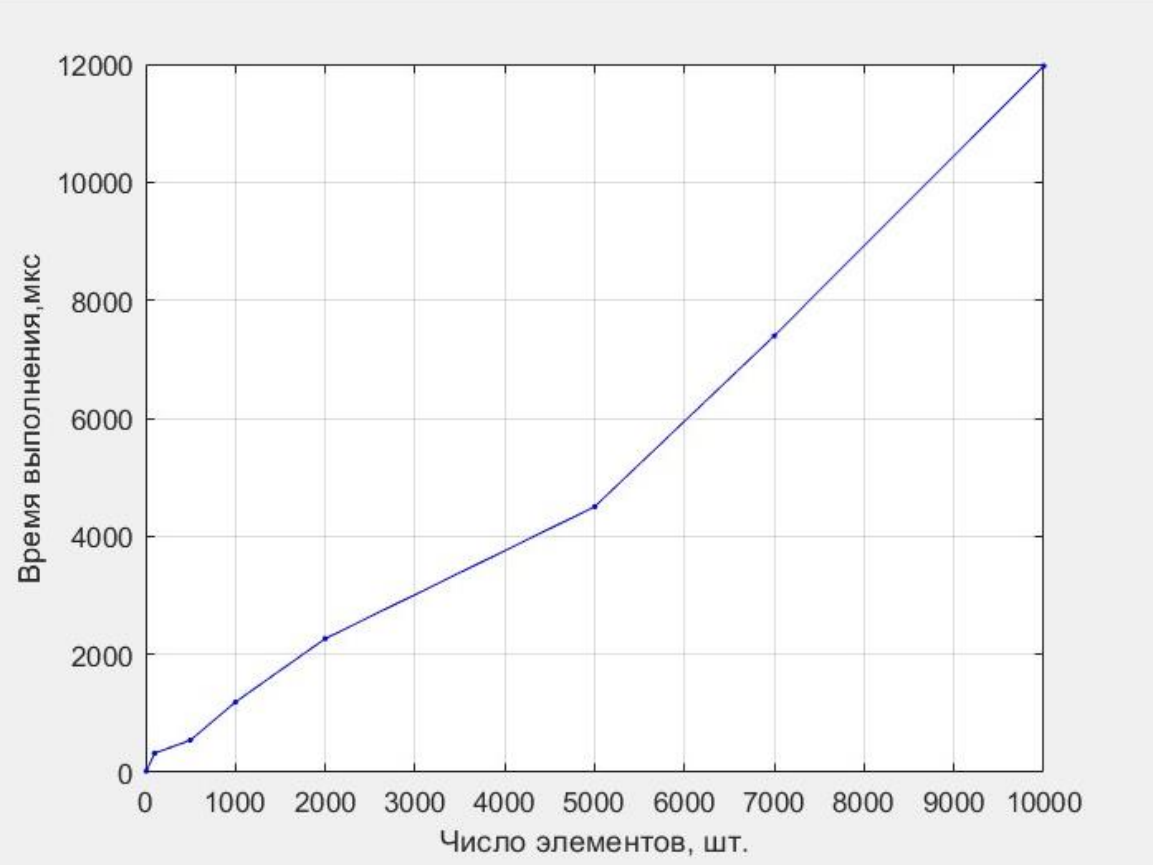

Рисунок 2 – Численный анализ

По графику видно, что в моей реализации алгоритма при больших значениях сложность приближается к линейной. Это происходит из-за рекурсивных вызовов родителя и дедушки в алгоритме.

Список литературы

1. Роберт Седжвик, Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных / пер. с англ. (Algorithms in C++) — М.: Вильямс. — 2011 г. — 1056 с
2. Томас Кормен, Алгоритмы: построение и анализ / пер. с англ. (Introduction to Algorithms) — А.: Шеня. — 2002 г. — 955 с

Приложение 1. Код программы

```cpp
#include <iostream>
#include <string>
#include <chrono>


class Node {
public:
    int key;
    std::string data;
    bool red;
public:
    Node *left;
    Node *right;
public:
    Node();
    Node(int key, std::string data);
    ~Node();
};


class Tree {
private:
    Node *root;
    Node *findNode(Node* root, int key);
    Node *findParent(Node * root, int key);
    static Node *findUncle(Node *root, int childKey, Node* parent, Node* grParent);
    void recurseAdding (Node* child, Node* root);
    void rotateRight (Node* node);
    void rotateLeft (Node* node);
    void outputTree (Node *root, int totalSpace);
public:
    void printTree();
    bool add(const int key, std::string data);
    std::string find(const int key);
public:
    Tree();
    ~Tree();
};

Node::Node() {
    left = nullptr;
    right = nullptr;
    red = true;
}

Node::Node(int key, std::string data) {
    this->key = key;
    this->data = data;
    left = nullptr;
    right = nullptr;
    red = true;
}

```

```cpp
Node::~Node() {
    delete left;
    delete right;
}

Tree::Tree() {
    root = nullptr;
}

Tree::~Tree() {
    delete root;
}

void Tree::outputTree(Node *root, int level) {
    if(root)
    {
        outputTree(root->left, level: level + 1);
        for(int i = 0;i< level;i++) std::cout<<"   ";
        if (root->red == true)
            printf( format: "r");
        if (root->red == false)
            printf( format: "b");
        std::cout << root->key << std::endl;
        outputTree(root->right, level: level + 1);
    }
}


void Tree::printTree() {
    outputTree(root, level: 0);
}

void Tree::rotateRight (Node *node) {
    Node *grgrParent = findParent(root, node->key);
    Node *parent = node->left;
    if (grgrParent != nullptr){
        if (grgrParent->right == node)
            grgrParent->right = parent;
        else
            grgrParent->left = parent;
    }
    node->left = parent->right;
    parent->right = node;
}

void Tree::rotateLeft(Node *node) {
    Node *grgrParent = findParent(root, node->key);
    Node *parent = node->right;
    if (grgrParent != nullptr){
        if (grgrParent->right == node)
            grgrParent->right = parent;
        else
            grgrParent->left = parent;
    }
```

```
108         node->right = parent->left;
109         parent->left = node;
110     }
111
112     Node* Tree::findNode(Node *root, int key) {
113         if (root == nullptr) return root;
114         if (root->key == key) return root;
115         if (root->key > key) return findNode(root->left, key);
116         return findNode(root->right, key);
117     }
118
119     Node* Tree::findParent(Node *root, int key){
120         if (root->key < key) {
121             if (root->right == nullptr)
122                 return root;
123             if (root->right->key == key)
124                 return root;
125             return findParent(root->right, key);
126         }
127
128         if (root->key > key) {
129             if (root->left == nullptr)
130                 return root;
131             if (root->left->key == key)
132                 return root;
133             return findParent(root->left, key);
134         }
135         return nullptr;
136     }
137
138     Node* Tree::findUncle(Node *root, int childKey, Node* parent, Node *grParent) {
139         if (parent == nullptr)
140             return nullptr;
141         if (grParent == nullptr)
142             return nullptr;
143         if (grParent->left == parent)
144             return grParent->right;
145         else
146             return grParent->left;
147     }
148
149     std::string Tree::find(const int key){
150         Node* temp = root;
151         temp = findNode(temp, key);
152         if (temp == nullptr)
153             return "nothing find";
154         return temp->data;
155     }
156
157     void Tree::recurseAdding(Node* child, Node* root) {
158         root->red = false;
159         child->red = true;
160         Node* parent = findParent(root, child->key);
161
162         if (parent == nullptr){
163             child->red = false;
```

```
        return;
    }

    if (!(parent->red)) {
        return;
    }

    Node* grParent = findParent(root, parent->key);
    Node* uncle = findUncle(root, child->key, parent, grParent);

    if (parent->red) {
        if ((uncle != nullptr) && uncle->red) {
            parent->red = false;
            uncle->red = false;
            grParent->red = true;
            return recurseAdding(grParent, root);
        }

        if ((uncle == nullptr) || !(uncle->red)) {
            if (grParent->left == parent){
                if (parent->right == child){
                    rotateLeft(parent);
                    child = grParent->left->left;
                    parent = findParent(root, child->key);
                    grParent = findParent(root, parent->key);
                }
                parent->red = false;
                grParent->red = true;
                rotateRight(grParent);
                root->red = false;
                return;
            }

            if (grParent->right == parent){
                if (parent->left == child){
                    rotateRight(parent);
                    child = grParent->right->right;
                    parent = findParent(root, child->key);
                    grParent = findParent(root, parent->key);
                }
                parent->red = false;
                grParent->red = true;
                rotateLeft(grParent);
                root->red = false;
                return;
            }
        }
    }
}

bool Tree::add(const int key, std::string data) {
    Node* child = new Node (key,data);

    if (findNode(root, key) != nullptr)
        return false;

```

```
220         if (root == nullptr){
221             child->red = false;
222             root = child;
223             return true;
224         }
225
226         Node* parent = findParent(root, key);
227
228         if (key <= parent->key)
229             parent->left = child;
230         else
231             parent->right = child;
232
233         recurseAdding(child, root);
234         return true;
235     }
236
237     int main() {
238         Tree testTree;
239         testTree.add( key: 20,  data: "1");
240         testTree.add( key: 86,  data: "2");
241         testTree.add( key: 35,  data: "3");
242         testTree.add( key: 13,  data: "4");
243         testTree.add( key: 62,  data: "5");
244         testTree.add( key: 100,  data: "6");
245         testTree.add( key: 48,  data: "7");
246         testTree.add( key: -15,  data: "8");
247         testTree.add( key: 0,  data: "9");
248         testTree.add( key: 5,  data: "10");
249         testTree.printTree();
250     }
```