Санкт-Петербургский политехнический университет Петра великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая проект

Дисциплина: Программирование на языках высокого уровня

Тема: Красно-черное дерево.

Разработал:
студент гр. 3331506/70401 Водорезов Г.И.

Преподаватель Ананьевский М.С.

Санкт-Петербург
2020

Оглавление

Введение

Постановка задачи

Красно-черное дерево - один из видов самобалансирующихся двоичных деревьев поиска. В свою очередь двоичное дерево является решением «словарной проблемы». Подразумевается, что каждой вершине дерева соответствует элемент, имеющий некое ключевое значение, в дальнейшем именуемое просто ключем.

ДДП позволяет выполнять следующие основные операции:

▫ Поиск вершины по ключу.

▫ Переход к предыдущей или последующей вершине, в порядке, определяемом ключами.

▫ Вставка вершины.

▫ Удаление вершины.

Описание алгоритма

Эффективность выполнения операций с деревом напрямую связана с его сбалансированностью, то есть с максимальной разницей между глубиной левого и правого поддерева среди всех вершин. Красно-черное дерево является сбалансированным деревом, т.е. деревом, где путь от вершины дерево до любого листа дерева занимает примерно одинаковое время.

Красно-черное дерево обладает всеми свойствами и правилами организации бинарного дерева поиска. Из-за введения дополнительного атрибута узла – цвета, добавляется несколько новых свойств и правил:

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 — чёрные и не содержат данных.
4. Оба потомка каждого красного узла — чёрные.

5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Пример красно-черного дерева представлен на рисунке 1.

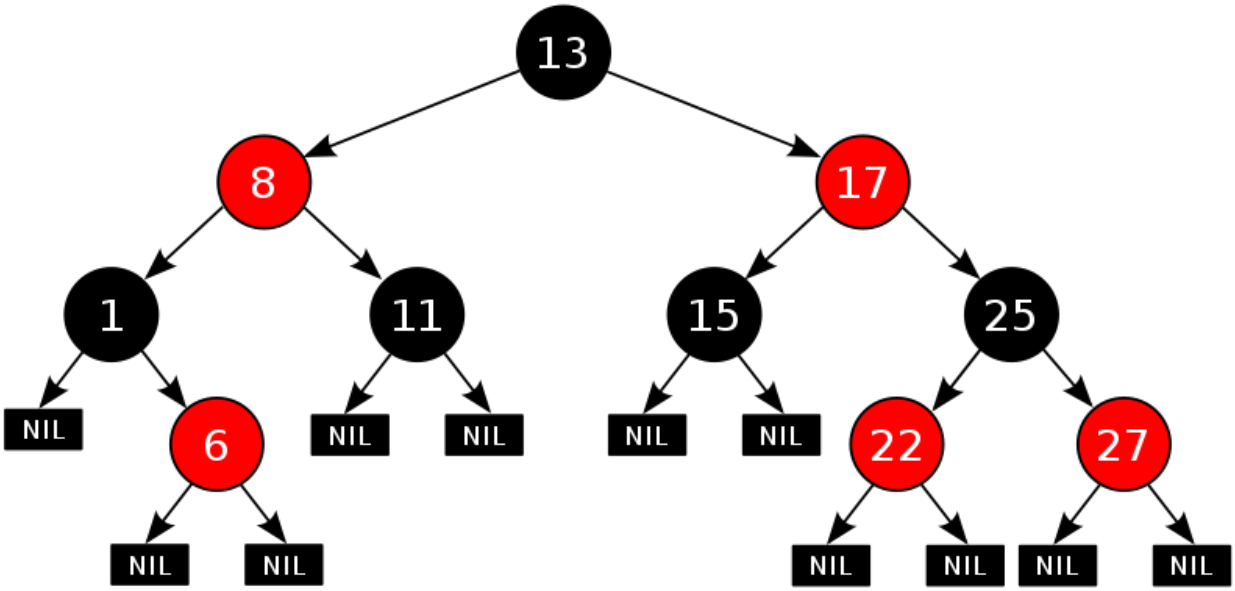

Рисунок 1 – Граф красно-черного дерева

Реализация алгоритма

Алгоритм был реализован на основе бинарного дерева поиска. Так же существует алгоритм на основе B-дерева. Были реализованы вышеперечисленный функционал двоичного дерева поиска, а также вывод дерева с помощью обхода в ширину и обхода в глубину, замер времени при заполнение разным количеством элементов.

Алгоритм реализован на языке C++, полный код представлен в Приложение 1.

Анализ алгоритма

Сложность алгоритма

Так как красно-черное дерево является сбалансированным, то сложность всех операций над узлами одинаковая, так же она одинаковая при худшем и лучшем случае расположения элементов и имеет нотацию O(log N). Занимаемое в памяти место под хранение данных – O(N).

Численный анализ алгоритма

Посчитаем время заполнения нашего дерева различным количеством данных. Численные результаты приведены в таблице 1.

Таблица 1.

Число элементов, шт.	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Время выполнения, мс.	0	0	0	1	576	2673	6320	10325	16234

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

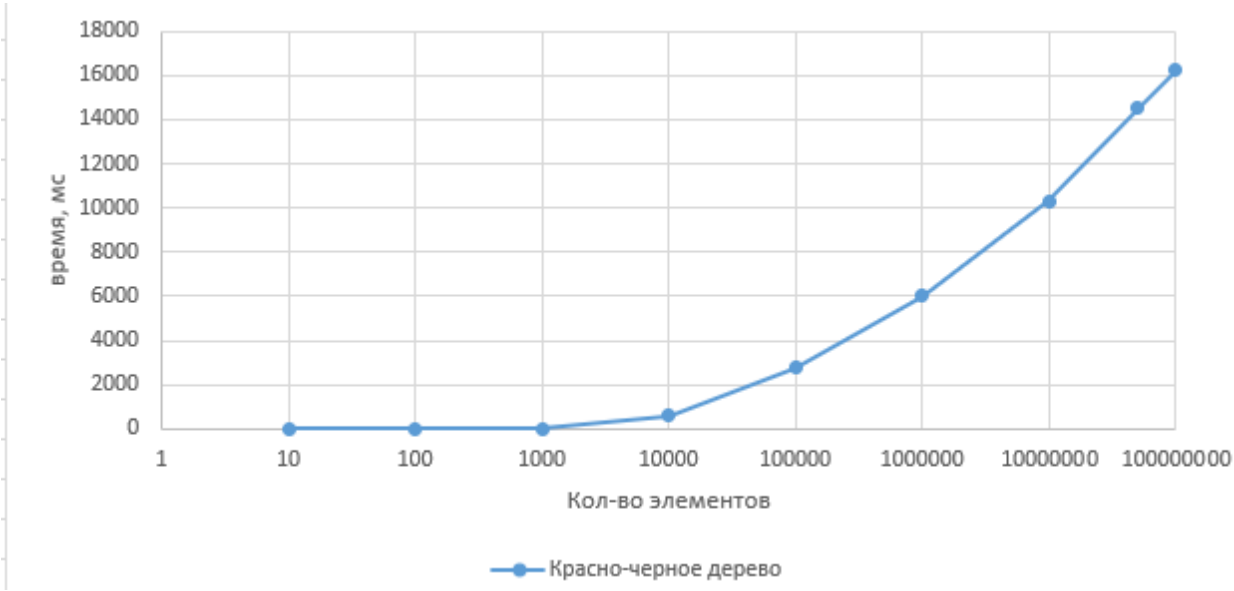

Рисунок 2 – Численный анализ.

Область применения

Красно-чёрные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. В частности, ассоциативные контейнеры библиотеки STL(map, set, multiset, multimap) основаны на красно-чёрных деревьях. TreeMap в Java тоже реализован на основе красно-чёрных деревьев.

Заключение

В ходе выполнения курсовой работы было рассмотрено красно-черно дерево, выполнена реализация алгоритма на языке C++, проведен анализ сложности алгоритма и его численный анализ. Так же была рассмотрена область применения данного двоичного дерева поиска.

Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ — 2-е изд. — М.: Издательский дом «Вильямс», 2011. — С. 336-364.

Приложение 1

```cpp
#include <iostream>
#include "RBTree.h"
#include <ctime>
using namespace std;

int main() {
    int N = 1000*1000*1 ;   //максимум элементов
    RBTree rbTree1;

    //считаем время вставки от 1 до N элементов с шагом в 1 порядок  Красно-черное дерево
    for (int j = 1; j <= N; j*=10)
    {
        cout << j << " elements" << endl;
        int start_time = clock();
        for (int i = 0; i < j ; i++)
        {
            rbTree1.insertValue(rand());
        }
        int end_time = clock(); // конечное время
        cout  << end_time - start_time << "ms" << "\n-------" << endl;

    }

    return 0;
}
```

```cpp
#ifndef RED_BLACK_TREE_RBTREE_H
#define RED_BLACK_TREE_RBTREE_H

enum Color { RED, BLACK, DOUBLE_BLACK };

struct Node
{
    int data;
    int color;
    Node* left;
    Node* right;
    Node* parent;

    Node(int data);
};

class RBTree
{
protected:
    void rotateLeft(Node*&);
    void rotateRight(Node*&);
    void fixInsertRBTree(Node*&);
    void fixDeleteRBTree(Node*&);
    int getColor(Node*&);
    void setColor(Node*&, int);
    Node* minValueNode(Node*&);
    Node* maxValueNode(Node*&);
    Node* insertBST(Node*&, Node*&);
    Node* deleteBST(Node*&, int);
    int getBlackHeight(Node*);
public:
    Node* root;
    RBTree();
    void insertValue(int);
    void deleteValue(int);
    void printDepth(Node* root);
    void printBreadth();

};


#endif //RED_BLACK_TREE_RBTREE_H
```

```cpp
#include <iostream>
#include <queue>
#include "RBTree.h"
using namespace std;

Node::Node(int data) {
    this->data = data;
    color = RED;
    left = right = parent = nullptr;
}

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

int RBTree::getColor(Node*& node) {
    if (node == nullptr)
        return BLACK;

    return node->color;
}

void RBTree::setColor(Node*& node, int color) {
    if (node == nullptr)
        return;

    node->color = color;
}

Node* RBTree::insertBST(Node*& root, Node*& ptr) {
    if (root == nullptr)
        return ptr;

    if (ptr->data < root->data) {
        root->left = insertBST(root->left, ptr);
        root->left->parent = root;
    }
    else if (ptr->data > root->data) {
        root->right = insertBST(root->right, ptr);
        root->right->parent = root;
    }

    return root;
}

void RBTree::insertValue(int n) {
    Node* node = new Node(n);
    root = insertBST(root, node);
    fixInsertRBTree(node);
}

void RBTree::rotateLeft(Node*& ptr) {
    Node* right_child = ptr->right;
    ptr->right = right_child->left;

    if (ptr->right != nullptr)
        ptr->right->parent = ptr;

    right_child->parent = ptr->parent;

    if (ptr->parent == nullptr)
        root = right_child;
    else if (ptr == ptr->parent->left)
        ptr->parent->left = right_child;
    else
        ptr->parent->right = right_child;

    right_child->left = ptr;
    ptr->parent = right_child;
}

void RBTree::rotateRight(Node*& ptr) {
    Node* left_child = ptr->left;
    ptr->left = left_child->right;

    if (ptr->left != nullptr)
        ptr->left->parent = ptr;

    left_child->parent = ptr->parent;

    if (ptr->parent == nullptr)
        root = left_child;
```

```
        root = left_child;
    else if (ptr == ptr->parent->left)
        ptr->parent->left = left_child;
    else
        ptr->parent->right = left_child;

    left_child->right = ptr;
    ptr->parent = left_child;
}

void RBTree::fixInsertRBTree(Node*& ptr) {
    Node* parent = nullptr;
    Node* grandparent = nullptr;
    while (ptr != root && getColor(ptr) == RED && getColor(ptr->parent) == RED) {
        parent = ptr->parent;
        grandparent = parent->parent;
        if (parent == grandparent->left) {
            Node* uncle = grandparent->right;
            if (getColor(uncle) == RED) {
                setColor(uncle, BLACK);
                setColor(parent, BLACK);
                setColor(grandparent, RED);
                ptr = grandparent;
            }
            else {
                if (ptr == parent->right) {
                    rotateLeft(parent);
                    ptr = parent;
                    parent = ptr->parent;
                }
                rotateRight(grandparent);
                swap(parent->color, grandparent->color);
                ptr = parent;
            }
        }
        else {
            Node* uncle = grandparent->left;
            if (getColor(uncle) == RED) {
                setColor(uncle, BLACK);
                setColor(parent, BLACK);
                setColor(grandparent, RED);
                ptr = grandparent;
            }
            else {
                if (ptr == parent->left) {
                    rotateRight(parent);
                    ptr = parent;
                    parent = ptr->parent;
                }
                rotateLeft(grandparent);
                swap(parent->color, grandparent->color);
                ptr = parent;
            }
        }
    }
    setColor(root, BLACK);
}

void RBTree::fixDeleteRBTree(Node*& node) {
    if (node == nullptr)
        return;

    if (node == root) {
        root = nullptr;
        return;
    }

    if (getColor(node) == RED || getColor(node->left) == RED || getColor(node->right) == RED) {
        Node* child = node->left != nullptr ? node->left : node->right;

        if (node == node->parent->left) {
            node->parent->left = child;
            if (child != nullptr)
                child->parent = node->parent;
            setColor(child, BLACK);
            delete (node);
        }
        else {
            node->parent->right = child;
            if (child != nullptr)
                child->parent = node->parent;
            setColor(child, BLACK);
            delete (node);
        }
    }
```

```
    }
    else {
        Node* sibling = nullptr;
        Node* parent = nullptr;
        Node* ptr = node;
        setColor(ptr, DOUBLE_BLACK);
        while (ptr != root && getColor(ptr) == DOUBLE_BLACK) {
            parent = ptr->parent;
            if (ptr == parent->left) {
                sibling = parent->right;
                if (getColor(sibling) == RED) {
                    setColor(sibling, BLACK);
                    setColor(parent, RED);
                    rotateLeft(parent);
                }
                else {
                    if (getColor(sibling->left) == BLACK && getColor(sibling->right) == BLACK) {
                        setColor(sibling, RED);
                        if (getColor(parent) == RED)
                            setColor(parent, BLACK);
                        else
                            setColor(parent, DOUBLE_BLACK);
                        ptr = parent;
                    }
                    else {
                        if (getColor(sibling->right) == BLACK) {
                            setColor(sibling->left, BLACK);
                            setColor(sibling, RED);
                            rotateRight(sibling);
                            sibling = parent->right;
                        }
                        setColor(sibling, parent->color);
                        setColor(parent, BLACK);
                        setColor(sibling->right, BLACK);
                        rotateLeft(parent);
                        break;
                    }
                }
            }
            else {
                sibling = parent->left;
                if (getColor(sibling) == RED) {
                    setColor(sibling, BLACK);
                    setColor(parent, RED);
                    rotateRight(parent);
                }
                else {
                    if (getColor(sibling->left) == BLACK && getColor(sibling->right) == BLACK) {
                        setColor(sibling, RED);
                        if (getColor(parent) == RED)
                            setColor(parent, BLACK);
                        else
                            setColor(parent, DOUBLE_BLACK);
                        ptr = parent;
                    }
                    else {
                        if (getColor(sibling->left) == BLACK) {
                            setColor(sibling->right, BLACK);
                            setColor(sibling, RED);
                            rotateLeft(sibling);
                            sibling = parent->left;
                        }
                        setColor(sibling, parent->color);
                        setColor(parent, BLACK);
                        setColor(sibling->left, BLACK);
                        rotateRight(parent);
                        break;
                    }
                }
            }
        }
        if (node == node->parent->left)
            node->parent->left = nullptr;
        else
            node->parent->right = nullptr;
        delete(node);
        setColor(root, BLACK);
    }
}

Node* RBTree::deleteBST(Node*& root, int data) {
    if (root == nullptr)
        return root;

    if (data < root->data)
```

```
250     if (data < root->data)
251         return deleteBST(root->left, data);
252 
253     if (data > root->data)
254         return deleteBST(root->right, data);
255 
256     if (root->left == nullptr || root->right == nullptr)
257         return root;
258 
259     Node* temp = minValueNode(root->right);
260     root->data = temp->data;
261     return deleteBST(root->right, temp->data);
262 }
263 
264 void RBTree::deleteValue(int data) {
265     Node* node = deleteBST(root, data);
266     fixDeleteRBTree(node);
267 }
268 
269 
270 
271 void RBTree::printDepth(Node* root) {
272     if (root == nullptr)
273         return;
274 
275     cout << root->data << " " << root->color << endl;
276     printDepth(root->left);
277     printDepth(root->right);
278 
279 }
280 
281 
282 void RBTree::printBreadth()       //вывод дерева в консоль в ширину
283 {
284     queue <Node*> elements;
285     elements.push(root);
286 
287     while (!elements.empty())
288     {
289         Node* buffer = elements.front();
290         cout << buffer->data << endl;
291         elements.pop();
292 
293         if (buffer->left != nullptr)
294         {
295             elements.push(buffer->left);
296         }
297         if (buffer->right != nullptr)
298         {
299             elements.push(buffer->right);
300         }
301     }
302 }
303 
304 Node* RBTree::minValueNode(Node*& node) {
305 
306     Node* ptr = node;
307 
308     while (ptr->left != nullptr)
309         ptr = ptr->left;
310 
311     return ptr;
312 }
313 
314 Node* RBTree::maxValueNode(Node*& node) {
315     Node* ptr = node;
316 
317     while (ptr->right != nullptr)
318         ptr = ptr->right;
319 
320     return ptr;
321 }
322 
323 int RBTree::getBlackHeight(Node* node) {
324     int blackheight = 0;
325     while (node != nullptr) {
326         if (getColor(node) == BLACK)
327             blackheight++;
328         node = node->left;
329     }
330     return blackheight;
331 }
332 
```