Лабораторная работа №2
по дисциплине
«Методы программирования»

Поиск и сравнение алгоритмов

Работу выполнил

Саркисянц Юрий Григорьевич

Студент СКБ 221

Работу проверил

Сластников Сергей Александрович

Алфавитный указатель классов

Классы

Классы с их кратким описанием.

Классы

Структура BSTNode

Узел бинарного дерева поиска (BST).

Открытые члены

- **BSTNode** (int val)
  Конструктор узла.

Открытые атрибуты

- int **key**
  Значение узла.

- **BSTNode * left**
  Указатель на левое поддерево (узлы с меньшими значениями).

- **BSTNode * right**
  Указатель на правое поддерево (узлы с большими значениями).

Подробное описание

Узел бинарного дерева поиска (BST).

Каждый узел содержит значение и указатели на левого и правого потомков.

Объявления и описания членов структуры находятся в файле:

- main_dox.cpp

Класс HashTable

Класс хеш-таблицы с открытой адресацией (линейное пробирование).

Открытые члены

- **HashTable** (int sz)
  Создает хеш-таблицу заданного размера.

- bool **insert** (int key)
  Вставляет ключ в хеш-таблицу с линейным пробированием.

- bool **search** (int key)
  Ищет ключ в хеш-таблице.

Подробное описание

Класс хеш-таблицы с открытой адресацией (линейное пробирование).

Хеш-таблица хранит целочисленные ключи. Для вычисления индекса используется простая функция hash(key) = key % size. При коллизии (ячейка занята другим ключом) осуществляется линейный переход к следующей ячейке (idx+1) по кругу.

Конструктор(ы)

HashTable::HashTable (int sz) `[inline]`

Создает хеш-таблицу заданного размера.

Аргументы

sz	Размер таблицы.

Методы

bool HashTable::insert (int key) `[inline]`

Вставляет ключ в хеш-таблицу с линейным пробированием.

Вычисляет индекс как key % size. Если ячейка свободна, вставляет ключ. При коллизии (ячейка занята и ключ не совпадает) последовательно проверяет следующую ячейку (idx+1)size, и так далее. Если возвращается к началу (таблица полна), вставка прерывается.

Пример использования:

```
HashTable ht(10);
ht.insert(5);
ht.insert(15); // коллизия с 5, вставит в следующую свободную ячейку
```

Аргументы

key	Ключ для вставки.

Возвращает

true, если вставка успешна, или false, если таблица полна или ключ уже есть.

bool HashTable::search (int key) `[inline]`

Ищет ключ в хеш-таблице.

Начинает с индекса key % size и проверяет ячейки, пока не найдет искомый ключ или не наткнется на пустую. При коллизии идет на следующую ячейку (линейное пробирование). Это простой метод, но в худшем случае при заполненной таблице требуется проверить многие ячейки (линейный поиск) — что медленно.

Пример использования:

```
HashTable ht(10);
ht.insert(3);
bool found = ht.search(13); // если 13 было вставлено, вернет true
```

Аргументы

key	Искомый ключ.

Возвращает

true, если ключ найден, иначе false.

Объявления и описания членов класса находятся в файле:

- main_dox.cpp

Класс RBT

Класс красно-черного дерева (**RBT**) с операциями вставки и поиска.

Открытые члены

- **RBT** ()
  Конструктор: инициализирует пустое дерево.

- void **insertRBT** (int value)
  Вставляет значение в красно-черное дерево.

- **RBTNode * searchRBT** (int value)
  Ищет узел по значению в красно-черном дереве.

Подробное описание

Класс красно-черного дерева (**RBT**) с операциями вставки и поиска.

Методы

void RBT::insertRBT (int value) `[inline]`

Вставляет значение в красно-черное дерево.

Сначала узел вставляется как в обычное BST: помещается в лист по правилу "меньше — налево, больше — направо", и красится в красный цвет. После этого вызывается процедура балансировки insertFix():

- Если у родителя нового узла тоже красный, устраняется двойной красный за счет поворотов и перекраски: рассматриваются случаи с «дядей» (красный или черный).
- Если дядя красный, оба (родитель и дядя) становятся черными, а дед — красным.
- Иначе выполняются вращения (leftRotate или rightRotate) и перекраска, чтобы восстановить свойства **RBT**. В результате гарантируется, что дерево сбалансировано и корень всегда черный.

Пример использования:

```
RBT tree;
tree.insertRBT(20);
tree.insertRBT(15);
tree.insertRBT(25);
RBTNode* node = tree.searchRBT(15);
```

Аргументы

value	Значение для вставки.

RBTNode * RBT::searchRBT (int value) `[inline]`

Ищет узел по значению в красно-черном дереве.

Осуществляется обычный поиск по BST: начиная с корня сравниваем искомое значение со значением текущего узла и идем налево или направо.

Пример использования:

```
RBT tree;
tree.insertRBT(8);
tree.insertRBT(4);
RBTNode* found = tree.searchRBT(4);
if (found) {
    std::cout << "Found RBT node with key " << found->key;
}
```

Аргументы

value	Искомое значение.

Возвращает

Указатель на узел с этим значением или nullptr, если не найден.

Объявления и описания членов класса находятся в файле:

- main_dox.cpp

Структура RBTNode

Узел красно-черного дерева (**RBT**).

Открытые члены

- **RBTNode** (int val)
  Конструктор.

Открытые атрибуты

- int **key**
  Значение узла.

- Color **color**
  Цвет узла (RED или BLACK).

- **RBTNode * left**
  Левый потомок.

- **RBTNode * right**
  Правый потомок.

- **RBTNode * parent**
  Родительский узел.

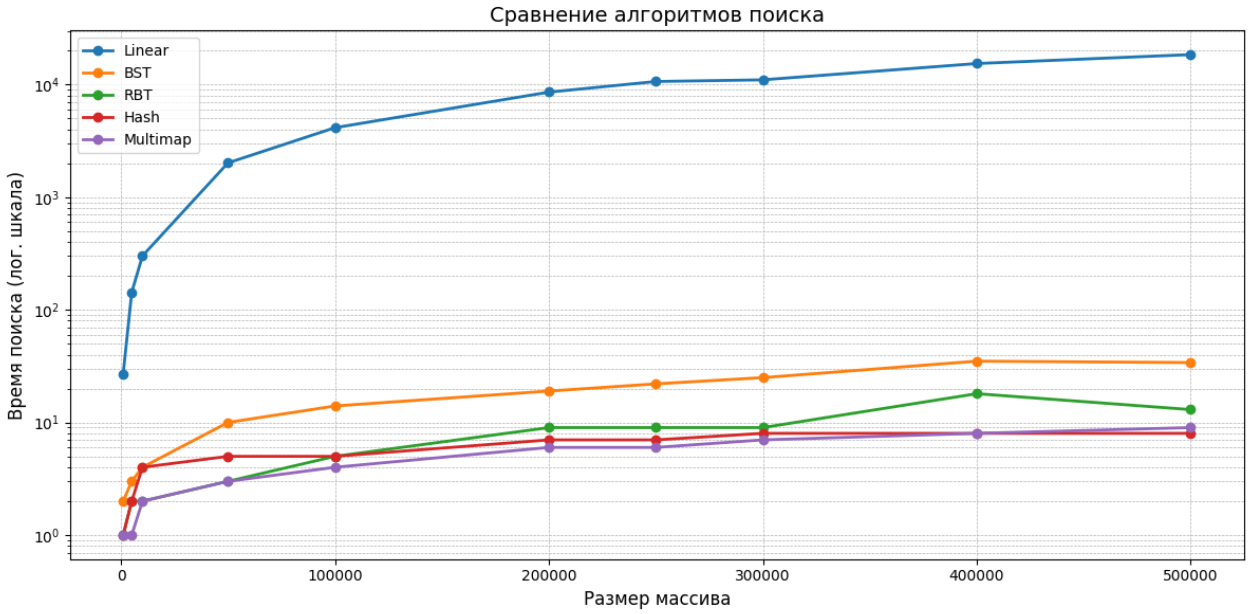
Подробное описание

Узел красно-черного дерева (**RBT**).

Каждому узлу кроме ключа сопоставлен цвет (красный или черный), а также указатели на родителя и детей. Красно-черное дерево поддерживает балансировку после вставок.

Объявления и описания членов структуры находятся в файле:

- main_dox.cp



-

Исходник main.cpp

```
#include <iostream>
#include <vector>
#include <map>
#include <unordered_map>
#include <chrono>
#include <fstream>
#include <random>
#include <algorithm>
#include <sstream>

// Структура данных
struct Data {
    std::string key;
    int value;
};

// === Линейный поиск ===
std::vector<Data> linearSearch(const std::vector<Data>& data, const
std::string& key) {
    std::vector<Data> results;
    for (const auto& item : data) {
        if (item.key == key) results.push_back(item);
    }
    return results;
}

// === BST ===
struct BSTNode {
    Data data;
    BSTNode* left;
    BSTNode* right;
    BSTNode() : left(nullptr), right(nullptr) {}
};
```

```
void insertBST(BSTNode*& root, const Data& value) {
    if (!root) {
        root = new BSTNode;
        root->data = value;
    }
    else if (value.key < root->data.key) {
        insertBST(root->left, value);
    }
    else {
        insertBST(root->right, value);
    }
}

void searchBST(BSTNode* root, const std::string& key,
std::vector<Data>& results) {
    if (!root) return;
    if (root->data.key == key) results.push_back(root->data);
    if (key <= root->data.key) searchBST(root->left, key, results);
    if (key >= root->data.key) searchBST(root->right, key, results);
}

// === Красно-черное дерево ===
enum Color { RED, BLACK };

struct RBNode {
    std::string key;
    std::vector<Data> values;
    Color color;
    RBNode* parent;
    RBNode* left;
    RBNode* right;

    RBNode(const std::string& k, const Data& d)
        : key(k), color(RED), parent(nullptr), left(nullptr),
right(nullptr) {
        values.push_back(d);
    }
};

RBNode* rbRoot = nullptr;

void leftRotate(RBNode*& root, RBNode* x) {
    if (!x || !x->right) return;
    RBNode* y = x->right;
    x->right = y->left;
    if (y->left) y->left->parent = x;
    y->parent = x->parent;
    if (!x->parent) root = y;
    else if (x == x->parent->left) x->parent->left = y;
    else x->parent->right = y;
    y->left = x;
    x->parent = y;
}

void rightRotate(RBNode*& root, RBNode* x) {
    if (!x || !x->left) return;
    RBNode* y = x->left;
    x->left = y->right;
    if (y->right) y->right->parent = x;
    y->parent = x->parent;
    if (!x->parent) root = y;
```

```
    else if (x == x->parent->right) x->parent->right = y;
    else x->parent->left = y;
    y->right = x;
    x->parent = y;
}

void fixInsert(RBNode*& root, RBNode* node) {
    while (node != root && node->parent && node->parent->color ==
RED) {
        RBNode* parent = node->parent;
        RBNode* grand = parent->parent;
        if (!grand) break;

        if (parent == grand->left) {
            RBNode* uncle = grand->right;
            if (uncle && uncle->color == RED) {
                parent->color = BLACK;
                uncle->color = BLACK;
                grand->color = RED;
                node = grand;
            }
            else {
                if (node == parent->right) {
                    node = parent;
                    leftRotate(root, node);
                }
                parent->color = BLACK;
                grand->color = RED;
                rightRotate(root, grand);
            }
        }
        else {
            RBNode* uncle = grand->left;
            if (uncle && uncle->color == RED) {
                parent->color = BLACK;
                uncle->color = BLACK;
                grand->color = RED;
                node = grand;
            }
            else {
                if (node == parent->left) {
                    node = parent;
                    rightRotate(root, node);
                }
                parent->color = BLACK;
                grand->color = RED;
                leftRotate(root, grand);
            }
        }
    }
    if (root) root->color = BLACK;
}

void insertRBT(RBNode*& root, const Data& d) {
    RBNode* node = root;
    RBNode* parent = nullptr;
    while (node) {
        parent = node;
        if (d.key == node->key) {
            node->values.push_back(d);
            return;
```

```
        }
        else if (d.key < node->key) node = node->left;
        else node = node->right;
    }
    RBNode* newNode = new RBNode(d.key, d);
    newNode->parent = parent;
    if (!parent) root = newNode;
    else if (d.key < parent->key) parent->left = newNode;
    else parent->right = newNode;
    fixInsert(root, newNode);
}

std::vector<Data> searchRBT(RBNode* root, const std::string& key) {
    RBNode* node = root;
    while (node) {
        if (key == node->key) return node->values;
        else if (key < node->key) node = node->left;
        else node = node->right;
    }
    return {};
}

// === Хеш-таблица с открытой адресацией ===
struct HashEntry {
    bool occupied = false;
    Data data;
};

struct HashTable {
    std::vector<HashEntry> table;
    size_t size;
    int collisions = 0;

    HashTable(size_t s) : size(s), table(s) {}

    // Простая хеш-функция
    size_t hash(const std::string& key) {
        unsigned long hash = 5381;
        for (char c : key) {
            hash = ((hash << 5) + hash) + c;
        }
        return hash % size;
    }

    void insert(const Data& d) {
        size_t idx = hash(d.key);
        size_t original_idx = idx;
        int steps = 0;

        while (table[idx].occupied) {
            if (table[idx].data.key == d.key) break; // не нужно
дублировать одинаковые ключи
            idx = (idx + 1) % size;
            steps++;
            if (idx == original_idx) return; // таблица заполнена
        }

        if (steps > 0) collisions++;
        table[idx].data = d;
        table[idx].occupied = true;
    }
```

```
    std::vector<Data> search(const std::string& key) {
        size_t idx = hash(key);
        size_t original_idx = idx;

        while (table[idx].occupied) {
            if (table[idx].data.key == key) return {table[idx].data};
            idx = (idx + 1) % size;
            if (idx == original_idx) break;
        }
        return {};
    }
};

// === Загрузка данных ===
std::vector<Data> loadDataset(const std::string& filename) {
    std::ifstream file(filename);
    std::vector<Data> result;
    std::string line;
    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;
        std::vector<std::string> tokens;
        while (std::getline(ss, token, ',')) tokens.push_back(token);
        if (tokens.size() >= 5) {
            result.push_back({tokens[4], rand()});
        }
    }
    return result;
}

// === Измерение времени ===
template<typename Func>
long long measureTime(Func f) {
    auto start = std::chrono::high_resolution_clock::now();
    f();
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::microseconds>(end
- start).count();
}

int main() {
    std::ofstream out("results.csv");
    out << "Size,Linear,BST,RBT,Hash,Multimap,Collisions\n";

    std::vector<int> sizes = {1000, 5000, 10000, 50000, 100000,
200000, 250000, 300000, 400000, 500000};
    int repeats = 10;

    for (int size : sizes) {
        std::string filename = "apartments_" + std::to_string(size) +
".txt";
        auto data = loadDataset(filename);

        if (data.empty()) {
            std::cerr << "[ERROR] Dataset " << filename << " is empty
or unreadable.\n";
            continue;
        }

        long long totalLinear = 0, totalBST = 0, totalRBT = 0,
```

```
totalHash = 0, totalMM = 0;
        int totalCollisions = 0;

        for (int rep = 0; rep < repeats; ++rep) {
            std::string targetKey = data[rand() % data.size()].key;

            totalLinear += measureTime([&]() {
                linearSearch(data, targetKey);
            });

            BSTNode* root = nullptr;
            for (const auto& d : data) insertBST(root, d);
            totalBST += measureTime([&]() {
                std::vector<Data> res;
                searchBST(root, targetKey, res);
            });

            rbRoot = nullptr;
            for (const auto& d : data) insertRBT(rbRoot, d);
            totalRBT += measureTime([&]() {
                searchRBT(rbRoot, targetKey);
            });

            size_t tableSize = size * 2;
            HashTable ht(tableSize);
            for (const auto& d : data) ht.insert(d);
            totalCollisions += ht.collisions;
            totalHash += measureTime([&]() {
                ht.search(targetKey);
            });

            std::multimap<std::string, Data> mm;
            for (const auto& d : data) mm.insert({d.key, d});
            totalMM += measureTime([&]() {
                auto range = mm.equal_range(targetKey);
                for (auto it = range.first; it != range.second; ++it)
{};
            });
        }

        out << size << "," << (totalLinear / repeats) << "," <<
(totalBST / repeats) << ","
            << (totalRBT / repeats) << "," << (totalHash / repeats)
<< ","
            << (totalMM / repeats) << "," << (totalCollisions /
repeats) << "\n";

        std::cout << "Size: " << size << " done.\n";
    }

    return 0;
}
```

Исходник gen.py

```
import pandas as pd
import matplotlib.pyplot as plt

# Загрузка данных
df = pd.read_csv("results.csv")  # замените на имя вашего файла
```

```
# Оформление графика
plt.figure(figsize=(12, 6))

# Стили и цвет
styles = {
    'Linear':     {'label': 'Linear',     'color': 'tab:blue',
'marker': 'o'},
    'BST':        {'label': 'BST',        'color': 'tab:orange',
'marker': 'o'},
    'RBT':        {'label': 'RBT',        'color': 'tab:green',
'marker': 'o'},
    'Hash':       {'label': 'Hash',       'color': 'tab:red',
'marker': 'o'},
    'Multimap':   {'label': 'Multimap',   'color': 'tab:purple',
'marker': 'o'},
}

# Построение линий
for column, style in styles.items():
    plt.plot(df['Size'], df[column],
             label=style['label'],
             color=style['color'],
             marker=style['marker'],
             linewidth=2)

# Настройки осей
plt.yscale('log')
plt.xlabel("Размер массива", fontsize=12)
plt.ylabel("Время поиска (лог. шкала)", fontsize=12)
plt.title("Сравнение алгоритмов поиска", fontsize=14)

# Сетка и легенда
plt.grid(True, which="both", linestyle='--', linewidth=0.5)
plt.legend()
plt.tight_layout()

# Сохранение
plt.savefig("comparison_cleaned.png")
plt.show()
```