



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №4

Тема: Сбалансированные деревья поиска (СДП) и их применение для поиска
данных в файле

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент

Хан А.А.

группа

ИКБО-04-20

Москва 2021

Оглавление

Тема	3
Цель.....	3
Персональный вариант и задания.....	3
Отчет по заданию 1	4
Постановка задачи.....	4
Подход к решению	4
Отчет по заданию 2	9
Постановка задачи.....	9
Подход к решению	9
Отчет по заданию 3	11
Постановка задачи.....	11
Код приложения	12
Выводы	18
Список информационных источников.....	18

Тема

Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле.

Цель

- получить навыки в разработки и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями);
- получить навыки в применении файловых потоков прямого доступа к данным файла;
- получить навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

Персональный вариант и задания

Вариант этого практического тот же, что и в практическом задании 2 – хеш-таблицы.

Вариант №12

№	Сбалансированное дерево поиска (СДП)	Структура элемента множества (ключ – подчеркнутое поле) остальные поля представляют данные элемента
12	АВЛ	Регистрация малого предприятия: номер лицензии, название, учредитель

Отчет по заданию 1

Постановка задачи

Задание 1.

Разработайте приложение, которое использует БДП для организации прямого доступа к записям файла, структура записи которого приведена в варианте (отобразить задачу варианта)

1. Разработать класс «Бинарное дерево поиска». Тип информационной части узла ключ и ссылка на запись в файле.

Методы: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева.

2. Разработать класс управления файлом. Включить методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск записи в файле с использованием БДП; остальные методы по вашему усмотрению.

Дано.

Файл двоичный с записями фиксированной длины.

Результат.

Приложение, выполняющее операции (перечислить для файла и поисковой структурой)

Подход к решению

Класс «Бинарное дерево поиска».

Поля класса:

- *int num* – номер лицензии.
- *int shift* – сдвиг от начала бинарного файла до места, где расположены данные (название и учредитель), соответствующие номеру лицензии.
- *int h* - высота, на которой находится вершина дерева.
- *node * left* – указатель на левого потомка.
- *node * right* – указатель на правого потомка.

```

class node {
public:
    int num;
    int shift;
    int h;
    node * left;
    node * right;

    node(int _num = 0, int _shift = 0) {
        num = _num;
        shift = _shift;
        h = 1;
        left = NULL;
        right = NULL;
    }
};

```

Методы класса:

- *static int get_h(node *)* – возвращает высоту вершины.
- *static void set_h(node *)* – устанавливает значение высоты вершины, используя высоты потомков.
- *static node * update(node *)* – проверяет баланс вершины и, при необходимости, балансирует, запуская функции поворотов.
- *static node * small_left_rotate(node *)* – выполняет малый левый поворот.
- *static node * small_right_rotate(node *)* – выполняет малый правый поворот.
- *static node * big_left_rotate(node *)* – выполняет большой левый поворот.
- *static node * big_right_rotate(node *)* – выполняет большой правый поворот.
- *static node * find_elem(node *,int)* – поиск вершины в дереве по ключу.
- *static int find_left(node *)* – поиск минимального значения ключа в поддереве.
- *static node * delete_left(node *)* – удаление вершины с минимальным ключом в поддереве.
- *static node * add_elem(node *,int,int)* – добавление вершины в дерево по ключу и сдвигу от начала бинарного файла до соответствующей записи.
- *static node * delete_elem(node *,int)* – удаление вершины из дерева по ключу.
- *static void print_tree(node *)* – вывод дерева на печать.
- *static bool is_balanced(node *)* – проверка всего дерева на сбалансированность.

Класс записей для двоичного файла.

Поля класса:

- *int num* – номер лицензии.
- *char name[SZ]* – строка фиксированного размера, содержащая название.
- *char founder[SZ]* – строка фиксированного размера, содержащая учредителя.

```
class block {  
public:  
    int num;  
    char name[SZ];  
    char founder[SZ];  
};
```

Операции по управлению бинарным файлом:

- *node * tree_from_file()* – из текстового файла *input.txt* извлекаем данные в класс *block*, добавляем их в созданный бинарный файл *input.bin* и в дерево с корнем *root* добавляем новую вершину, соответствующую извлеченным данным.

Алгоритмы на псевдокоде.

Вставка элемента в БДП:

```
Function добавить вершину  
Pass in: v, x, shift  
  
IF v == NULL  
    THEN  
        создать новую вершину дерева new_node = node(x, shift)  
        вернуть указатель на new_node  
ELSE IF v->num == x  
    вернуть указатель на v  
  
ELSE IF v->num > x  
    добавить вершину в левого потомка v  
ELSE  
    добавить вершину в правого потомка v  
  
обновить высоту v  
сбалансировать v  
вернуть указатель на v
```

Поиск записи по ключу в БДП:

```
Function найти вершину
Pass in: v, x

IF v == NULL
    THEN
        вернуть NULL
ELSE IF v->num == x
    вернуть указатель на v

ELSE IF v->num > x
    найти вершину в левом потомке v
ELSE
    найти вершину в правом потомке v
```

Удаление элемента БДП:

```
Function удалить вершину
Pass in: v, x

IF v == NULL
    THEN
        вернуть NULL
ELSE IF v->num == x
    IF v лист
        THEN
            удалить v
            вернуть NULL
    ELSE IF нет правого потомка v
        подвесить левого потомка L на место v
        вернуть L
    ELSE IF нет левого потомка v
        подвесить правого потомка R на место v
        вернуть R
    ELSE
        найти вершину P с минимальным значением в поддереве правого потомка v
        скопировать данные P в v
        удалить P
        вернуть v

ELSE IF v->num > x
    удалить вершину в левом потомке v
ELSE
    удалить вершину в правом потомке v

обновить высоту v
сбалансировать v
вернуть v
```

Тестовый файл:

1 n1 f1 4 n4 f4 2 n2 f2 100 n100 f100 7 n7 f7 9 n9 f9 3 n3 f3

Результат тестирования:

```
TREE:
1
 2
3
 4
 7
9
 100

ENTER:
<key_value> (to find) OR -1 (to stop)

INPUT:4
KEY 4: found
VALUE: n4 f4

TREE:
1
 2
3
 4
 7
9
 100
```

Скриншот №1

```
ENTER:
<key_value> (to find) OR -1 (to stop)

INPUT:100
KEY 100: found
VALUE: n100 f100

TREE:
1
 2
3
 4
 7
9
 100

ENTER:
<key_value> (to find) OR -1 (to stop)

INPUT:6
KEY 6: NOT found
TREE:
1
 2
3
 4
 7
9
 100
```

Скриншот №2

```
ENTER:
<key_value> (to find) OR -1 (to stop)

INPUT:0
KEY 0: NOT found
TREE:
1
 2
3
 4
 7
9
 100

ENTER:
<key_value> (to find) OR -1 (to stop)

INPUT:9
KEY 9: found
VALUE: n9 f9

TREE:
1
 2
3
 4
 7
9
 100

ENTER:
<key_value> (to find) OR -1 (to stop)

INPUT:-1
```

Скриншот №3

Отчет по заданию 2

Постановка задачи

Задание 2

Разработать приложение, которое использует сбалансированное дерево поиска, предложенное в варианте, для доступа к записям файла.

Подход к решению

Функция test, генерирующая массивы add и del (из чисел от 1 до n в произвольном порядке) для последовательного добавления ключей в дерево, а затем удаления.

```
bool test(int n) {
    node * root = NULL;

    std::vector<int> a(n);
    for (int i = 0; i < n; ++i) {
        a[i] = i + 1;
    }

    std::vector<int> add(n), del(n);
    add = a;
    del = a;

    srand(time(0));

    std::random_shuffle(add.begin(), add.end());
    std::random_shuffle(del.begin(), del.end());

    for (int i = 0; i < n; ++i) {
        root = node::add_elem(root, add[i], 0);
        bool cur = node::is_balanced(root);
        if (!cur) {
            return 0;
        }
    }

    for (int i = 0; i < n; ++i) {
        root = node::delete_elem(root, del[i]);
        bool cur = node::is_balanced(root);
        if (!cur) {
            return 0;
        }
    }
    return 1;
}
```

Функция `count_rotate(int test_cnt)`, возвращающая среднее значение отношения количества поворотов к количеству добавленных ключей (на `test_cnt` тестах).

```
double count_rotate(int test_cnt) {  
    double ans_cnt = 0;  
  
    srand(time(0));  
  
    for (int i = 0; i < test_cnt; ++i) {  
        int n = rand() % 500;  
        rotate_cnt = 0;  
        test(n);  
        ans_cnt += (double)rotate_cnt / (double)n;  
    }  
  
    return ans_cnt / (double)test_cnt;  
}
```

Результаты на `test cnt = 1, 10, 100, 100`:

```
test number: 1 rotates mean: 0.692308  
test number: 10 rotates mean: 0.665778  
test number: 100 rotates mean: 0.690418  
test number: 1000 rotates mean: 0.691789
```

Отчет по заданию 3

Постановка задачи

Задание 3

Выполнить анализ алгоритма поиска записи с заданным ключом при применении структур данных:

- хеш-таблица;
- бинарное дерево поиска;
- СДП. Требования по выполнению задания

Таблица 1. Таблица результатов

Вид поисковой структуры	Количество элементов, загруженных в структуру в момент выполнения поиска	Объем памяти для структуры	Количество выполненных сравнений, время на поиск ключа в структуре
СДП	1	20	1
	100	2000	3
	1000	20000	4
	10000	200000	7
	100000	2000000	11
БДП	1	16	1
	100	1600	4
	1000	16000	5
	10000	160000	10
	100000	1600000	17

Таким образом, балансировка дерева является важной оптимизацией при достаточно больших n ($n > 10^5$).

Код приложения

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>
#include <ctime>

const int SZ = 500;
int rotate_cnt = 0;

class block {
public:
    int num;
    char name[SZ];
    char founder[SZ];
};

class node {
public:
    int num;
    int shift;
    int h;
    node * left;
    node * right;

    node(int _num = 0, int _shift = 0) {
        num = _num;
        shift = _shift;
        h = 1;
        left = NULL;
        right = NULL;
    }

    static int get_h(node *);
    static void set_h(node *);
    static node * update(node *);
    static node * small_left_rotate(node *);
    static node * small_right_rotate(node *);
    static node * big_left_rotate(node *);
    static node * big_right_rotate(node *);
    static node * find_elem(node *, int);
    static int find_left(node *);
    static node * delete_left(node *);
    static node * add_elem(node *, int, int);
    static node * delete_elem(node *, int);
    static void print_tree(node *);
    static bool is_balanced(node *);
};

int node::get_h(node * v) {
    if (v == NULL) {
        return 0;
    }
    return v->h;
}

void node::set_h(node * v) {
    if (v == NULL) {
        return;
    }
    v->h = std::max(get_h(v->left), get_h(v->right)) + 1;
}
```

```

}

node * node::small_left_rotate(node * a) {
    ++rotate_cnt;

    node * b = a->right;

    a->right = b->left;
    b->left = a;
    set_h(a);
    set_h(b);
    return b;
}

node * node::small_right_rotate(node * a) {
    ++rotate_cnt;

    node * b = a->left;

    a->left = b->right;
    b->right = a;
    set_h(a);
    set_h(b);
    return b;
}

node * node::big_left_rotate(node * a) {
    a->right = small_right_rotate(a->right);
    a = small_left_rotate(a);
    return a;
}

node * node::big_right_rotate(node * a) {
    a->left = small_left_rotate(a->left);
    a = small_right_rotate(a);
    return a;
}

node * node::find_elem(node * v, int x) {
    if (v == NULL) {
        return NULL;
    }
    if (v->num == x) {
        return v;
    }
    if (v->num > x) {
        return find_elem(v->left, x);
    } else {
        return find_elem(v->right, x);
    }
}

node * node::update(node * a) {
    if (a == NULL) {
        return a;
    }
    int delta_a = get_h(a->left) - get_h(a->right);
    if (std::abs(delta_a) <= 1) {
        return a;
    }

    if (delta_a == -2) {
        node * b = a->right;
        int delta_b = get_h(b->left) - get_h(b->right);
    }
}

```

```

        if (delta_b == 0 || delta_b == -1) {
            return small_left_rotate(a);
        } else {
            return big_left_rotate(a);
        }
    } else {
        node * b = a->left;
        int delta_b = get_h(b->left) - get_h(b->right);

        if (delta_b == 0 || delta_b == 1) {
            return small_right_rotate(a);
        } else {
            return big_right_rotate(a);
        }
    }
}

int node::find_left(node * v) {
    if (v->left == NULL) {
        return v->num;
    } else {
        return find_left(v->left);
    }
}

node * node::delete_left(node * v) {
    if (v->left == NULL) {
        node * q = v->right;
        delete v;
        return q;
    } else {
        v->left = delete_left(v->left);

        set_h(v->left);
        set_h(v);
        v = update(v);
        return v;
    }
}

node * node::add_elem(node * v, int x, int shift) {
    if (v == NULL) {
        node * new_node = new node(x, shift);
        return new_node;
    }
    if (v->num == x) {
        return v;
    }

    if (v->num > x) {
        v->left = add_elem(v->left, x, shift);
    } else {
        v->right = add_elem(v->right, x, shift);
    }

    set_h(v);
    v = update(v);

    return v;
}

node * node::delete_elem(node * v, int x) {

```

```

    if (v == NULL) {
        return NULL;
    }

    if (v->num == x) {
        if (v->left == NULL && v->right == NULL) {
            delete v;
            return NULL;
        } else if (v->left == NULL) {
            node * new_v = v->right;
            delete v;
            return new_v;
        } else if (v->right == NULL) {
            node * new_v = v->left;
            delete v;
            return new_v;
        } else {
            v->num = find_left(v->right);
            v->right = delete_left(v->right);

            set_h(v);
            v = update(v);
            return v;
        }
    }

    if (v->num > x) {
        v->left = delete_elem(v->left, x);
    } else {
        v->right = delete_elem(v->right, x);
    }
    set_h(v);
    v = update(v);

    return v;
}

void node::print_tree(node * v) {
    if (v == NULL) {
        return;
    }
    print_tree(v->left);

    for (int i = 0; i < v->h; ++i) {
        std::cout << " ";
    }
    std::cout << v->num << "\n";

    print_tree(v->right);
}

bool node::is_balanced(node * v) {
    if (v == NULL) {
        return 1;
    }
    if (is_balanced(v->left) && is_balanced(v->right)) {
        return std::abs(get_h(v->left) - get_h(v->right)) <= 1;
    } else {
        return 0;
    }
}

```

```

bool test(int n) {
    node * root = NULL;

    std::vector<int> a(n);
    for (int i = 0; i < n; ++i) {
        a[i] = i + 1;
    }

    std::vector<int> add(n), del(n);
    add = a;
    del = a;

    srand(time(0));

    std::random_shuffle(add.begin(), add.end());
    std::random_shuffle(del.begin(), del.end());

    for (int i = 0; i < n; ++i) {
        root = node::add_elem(root, add[i], 0);
        bool cur = node::is_balanced(root);
        if (!cur) {
            return 0;
        }
    }

    for (int i = 0; i < n; ++i) {
        root = node::delete_elem(root, del[i]);
        bool cur = node::is_balanced(root);
        if (!cur) {
            return 0;
        }
    }
    return 1;
}

double count_rotate(int test_cnt) {
    double ans_cnt = 0;

    srand(time(0));

    for (int i = 0; i < test_cnt; ++i) {
        int n = rand() % 500;
        rotate_cnt = 0;
        test(n);
        ans_cnt += (double)rotate_cnt / (double)n;
    }

    return ans_cnt / (double)test_cnt;
}

node * tree_from_file() {
    std::ifstream f;
    f.open("input.txt");

    FILE * f0 = fopen("input.bin", "wb");

    if (!f) {
        std::cout << "input file error";
        return NULL;
    }
    if (f0 == NULL) {
        std::cout << "binary file error";
        return NULL;
    }
}

```



```

node * root = NULL;
int shift = 0;
block cur;

while (f >> cur.num >> cur.name >> cur.founder) {

    root = node::add_elem(root, cur.num, shift);

    fwrite(&cur, sizeof(block), 1, f0);
    ++shift;
}

fclose(f0);
f.close();
return root;
}

int main() {
    node * root = tree_from_file();

    FILE * f1 = fopen("input.bin", "rb");
    std::ifstream f;
    f.open("input.txt");

    std::cout << "_____ \n";
    std::cout << "TREE:\n";
    node::print_tree(root);
    std::cout << "\n";

    while (1) {
        std::cout << "_____ \n";
        std::cout << "ENTER:\n<key_value> (to find) OR -1 (to stop)\n\n";
        std::cout << "INPUT:";
        int x;
        std::cin >> x;

        std::cout << "\n";

        if (x == -1) {
            break;
        }

        node * tmp = node::find_elem(root, x);
        block ans;

        if (tmp == NULL) {
            std::cout << "KEY " << x << ": NOT found\n";
        } else {
            std::cout << "KEY " << x << ": found\n";
            fseek(f1, tmp->shift * sizeof(block), SEEK_SET);
            fread(&ans, sizeof(block), 1, f1);
            std::cout << "\nVALUE: ";
            std::cout << ans.name << " ";
            std::cout << ans.founder << "\n\n";
        }

        std::cout << "TREE:\n";
        node::print_tree(root);
    }
    fclose(f1);
    f.close();

    return 0;
}

```

Выводы

В результате проделанной работы, я получила: навыки в разработке и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (АВЛ – деревьями), навыки в применении файловых потоков прямого доступа к данным файла, навыки в применении сбалансированного дерева поиска для прямого доступа к записям файла.

Список информационных источников

1. Лекционный материал по структурам и алгоритмам обработки данных Сартакова М. В. (дата обращения 17.10.2021)
2. Дополнительный материал к практическим работам по структурам и алгоритмам обработки данных Сорокина А. В. (дата обращения 17.10.2021)