Лабораторная работа № 2 по курсу дискретного анализа: сбалансированные деревья

Выполнил студент группы М80-208Б-22 МАИ *Цирулев Николай*.

Условие

- Необходимо создать программную библиотеку, реализующую указанную структуру данных, на основе которой разработать программу-словарь. В словаре каждому ключу, представляющему из себя регистронезависимую последовательность букв английского алфавита длиной не более 256 символов, поставлен в соответствие некоторый номер, от 0 до $2^{64} - 1$. Разным словам может быть поставлен в соответствие один и тот же номер.

  Программа должна обрабатывать строки входного файла до его окончания. Каждая строка может иметь следующий формат:

  + word 34 — добавить слово «word» с номером 34 в словарь.

  - word — удалить слово «word» из словаря.

  word — найти в словаре слово «word».

  ! Save /path/to/file — сохранить словарь в бинарном компактном представлении на диск в файл, указанный парамером команды.

  ! Load /path/to/file — загрузить словарь из файла.

- Вариант задания: AVL-дерево.

Метод решения

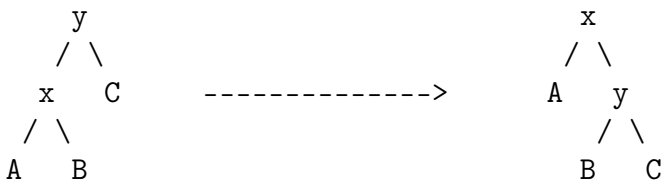
Для реализации словаря был разработан класс его элементов (пар "ключ-значение"), где ключ реализован в виде статического массива символов. Для хранения и управления данными был реализован шаблонный класс `TAVLTree`, представляющий собой AVL-дерево.

AVL-дерево представляет собой бинарное дерево поиска со сбалансированной высотой: для каждого узла x высота левого и правого поддеревьев x отличается не более чем на 1.

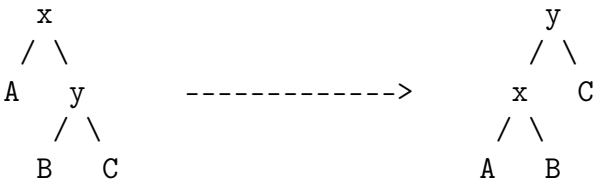
Вставка и удаление элементов были реализованы с использованием функции `balance()`. Данная функция предназначена для балансировки AVL-дерева с использованием левого и правого поворотов.

Балансировка высоты дерева достигается следующими двумя операциями:

- Левый поворот дерева:

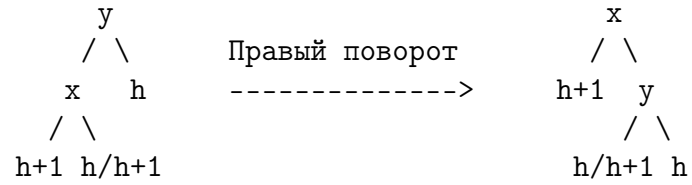


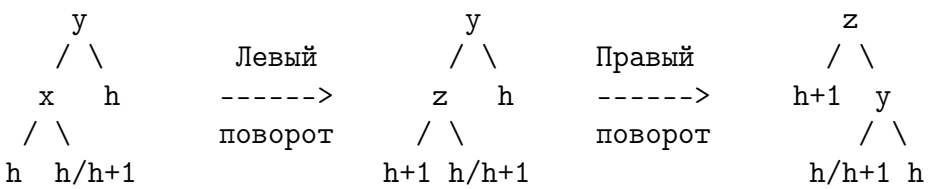
- Правый поворот дерева:

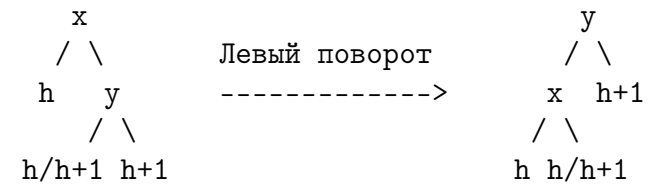


где x и y — внутренние узлы, а A, B и C — AVL-деревья.

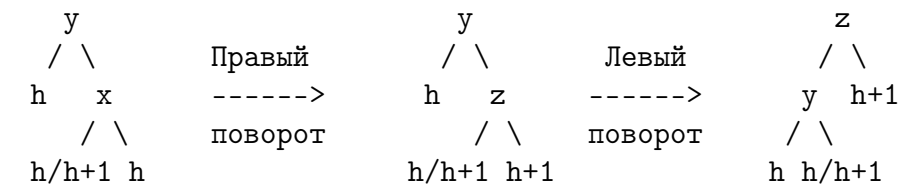
Так как после каждого удаления и каждой вставки мы балансируем дерево, то разница высот между левым и правым поддеревом корневого узла перед каждой балансировкой <= 2. Тогда требуется рассмотреть всего 4 случая:

1. 

2. 

3. 

4.



где x, y, z — внутренние узлы, а h и h+1 - AVL-деревья с высотами h и h+1 соответственно.

Описание программы

Исходный код программы содержится в файле main.cpp. В программе реализованы классы `AVLTreeData` и `TAVLTree`, которые обеспечивают работу словаря с использованием AVL-дерева. В классе `AVLTreeData` реализованы необходимые методы, конструкторы, деструкторы, а также перегружены операторы сравнения и индексации для корректной работы с данными:

```
#include <iostream>
#include <cstdint>
#include <algorithm>
#include <cstddef>
#include <fstream>
#include <stdexcept>
#include <utility>
#include <cstring>

#define STRING_SIZE 257

class AVLTreeData {
public:
    char key[STRING_SIZE];
    uint64_t value;
    AVLTreeData() {
        std::fill_n(key, STRING_SIZE, '\0');
        value = 0;
    }
    ~AVLTreeData() {
    }
    bool operator<(const AVLTreeData& other) const {
        return strcmp(key, other.key) < 0;
    }
    bool operator==(const AVLTreeData& other) const {
        return strcmp(key, other.key) == 0;
```

```
}
bool operator!=(const AVLTreeData& other) const {
    return strcmp(key, other.key) != 0;
}
bool operator>(const AVLTreeData& other) const {
    return strcmp(key, other.key) > 0;
}

char& operator[](size_t idx) {
    if (idx >= STRING_SIZE) {
        throw std::out_of_range("Index out of range");
    }
    return key[idx];
}

const char& operator[](size_t idx) const {
    if (idx >= STRING_SIZE) {
        throw std::out_of_range("Index out of range");
    }
    return key[idx];
}
AVLTreeData(const AVLTreeData& other) {
    std::copy(other.key, other.key + STRING_SIZE, key);
    value = other.value;
}

AVLTreeData& operator=(const AVLTreeData& other) {
    if (this != &other) {
        std::copy(other.key, other.key + STRING_SIZE, key);
        value = other.value;
    }
    return *this;
}

friend std::ostream& operator<<(std::ostream& os, const AVLTreeData& da
    for (size_t i = 0; i < STRING_SIZE; ++i) {
        os << data[i];
    }
    os << ", " << data.value;
    return os;
}
friend std::istream& operator>>(std::istream& is, AVLTreeData& data) {
```

```
        is >> data.key;
        for (size_t i = 0; i < STRING_SIZE; i++) {
            if (data.key[i] != '\0') {
                data.key[i] = std::tolower(data.key[i]);
            }
        }
        return is;
    }
};


template <typename T>
class TAVLTree {
public:
    class TNode {
    public:
        T data;
        int height;
        TNode* left;
        TNode* right;
        TNode(T d) {
            height = 1;
            data = d;
            left = nullptr;
            right = nullptr;
        }
        void destroy() {
            if (this != nullptr) {
                left->destroy();
                right->destroy();
                delete this;
            }
        }
    };
    TNode* root = nullptr;
    int n;
    TAVLTree() : root(nullptr), n(0) { }

    void insert(T x) { root = insertUtil(root, x); }

    void remove(T x) {
        if (root != nullptr) {
```

```
            root = removeUtil(root, x);
        }
    }

    TNode* search(const T x) { return searchUtil(root, x); }

    void printTree() { printTreeUtil(root, 0); }

    void clear() {
        if (root != nullptr) {
            destroy(root->left);
            destroy(root->right);
            delete root;
        }
        root = nullptr;
    }
    void destroy(TNode*& node) {
        if (node != nullptr) {
            destroy(node->left);
            destroy(node->right);
            delete node;
        }
    }

    void destroy() {
        destroy(root);
    }

    void LoadFromFile(std::ifstream& file) {
        clear();
        size_t fileSize;
        if (file.read(reinterpret_cast<char *>(&fileSize), sizeof(size_t)))
            for (size_t i = 0; i < fileSize; ++i) {
                T data;
                file.read(data.key, STRING_SIZE);
                file.read(reinterpret_cast<char*>(&data.value), sizeof(data
                insert(data);
            }
        }
    }

    void saveToFile(std::ofstream& file) {
```

```
        saveToFileUtil(file, root);
    }
    size_t getSize() {
        return getSizeUtil(root);
    }

private:
    size_t getSizeUtil(const TNode* node) {
        if (node != nullptr) {
            return 1 + getSizeUtil(node->left) + getSizeUtil(node->right);
        }
        return 0;
    }
    void saveToFileUtil(std::ofstream& file, TNode* node) {
        if (node == nullptr) {
            return;
        }
        file.write(node->data.key, STRING_SIZE);
        file.write(reinterpret_cast<const char*>(&node->data.value), sizeof

        saveToFileUtil(file, node->left);
        saveToFileUtil(file, node->right);
    }

    void printTreeUtil(TNode* head, int space) {
        if (head == nullptr)
            return;
        space += 10;
        printTreeUtil(head->right, space);
        std::cout << '\n';
        for (int i = 10; i < space; i++)
            std::cout << "␣";
        std::cout << head->data << "\n";
        printTreeUtil(head->left, space);
    }

    TNode* searchUtil(TNode* head, T x) {
        if (head == nullptr)
            return nullptr;
        T k = head->data;
        if (k == x) {
            return head;
```

```
    } else if (k > x) {
        return searchUtil(head->left, x);
    } if (k < x) {
        return searchUtil(head->right, x);
    }
    return nullptr;
}

int height(TNode* head) {
    if (head == nullptr)
        return 0;
    return head->height;
}

TNode* rightRotation(TNode* head) {
    TNode* newhead = head->left;
    head->left = newhead->right;
    newhead->right = head;
    head->height = 1 + std::max(height(head->left), height(head->right
    newhead->height = 1 + std::max(height(newhead->left), height(newhea
    return newhead;
}

TNode* leftRotation(TNode* head) {
    TNode* new_head = head->right;
    head->right = new_head->left;
    new_head->left = head;
    head->height = 1 + std::max(height(head->left), height(head->right
    new_head->height = 1 + std::max(height(new_head->left), height(new_
    return new_head;
}

TNode* balance(TNode* head) {
    int bal = height(head->left) - height(head->right);
    if (bal > 1) {
        if (height(head->left) >= height(head->right)) {
            return rightRotation(head);
        } else {
            head->left = leftRotation(head->left);
            return rightRotation(head);
        }
    } else if (bal < -1) {
```

```
        if (height(head->right) >= height(head->left)) {
            return leftRotation(head);
        } else {
            head->right = rightRotation(head->right);
            return leftRotation(head);
        }
    }
    return head;
}

TNode* insertUtil(TNode* head, T x) {
    if (head == nullptr) {
        n += 1;
        TNode* temp = new TNode(x);
        if (temp == nullptr) {
            throw std::bad_alloc();
        }
        return temp;
    }
    if (x < head->data)
        head->left = insertUtil(head->left, x);
    else if (x > head->data)
        head->right = insertUtil(head->right, x);
    head->height = 1 + std::max(height(head->left), height(head->right)
    return balance(head);
}
TNode* removeUtil(TNode* head, T x) {
    if (head == nullptr)
        return nullptr;
    if (x < head->data) {
        head->left = removeUtil(head->left, x);
    } else if (x > head->data) {
        head->right = removeUtil(head->right, x);
    } else {
        TNode* r = head->right;
        if (head->right == nullptr) {
            TNode* l = head->left;
            delete (head);
            head = l;
        } else if (head->left == nullptr) {
            delete (head);
            head = r;
```

```
            } else {
                while (r->left != nullptr)
                    r = r->left;
                head->data = r->data;
                head->right = removeUtil(head->right, r->data);
            }
        }
        if (head == nullptr)
            return head;
        head->height = 1 + std::max(height(head->left), height(head->right
        return balance(head);
    }
};



int main() {
    TAVLTree<AVLTreeData> tree;
    AVLTreeData command;
    while (std::cin >> command) {
        TAVLTree<AVLTreeData>::TNode* node;
        if (command[0] == '+') {
            AVLTreeData data;
            std::cin >> data >> data.value;
            node = tree.search(data);
            if (node == nullptr) {
                tree.insert(data);
                std::cout << "OK" << '\n';
            } else {
                std::cout << "Exist" << '\n';
            }
        } else if (command[0] == '-') {
            AVLTreeData data;
            std::cin >> data;
            node = tree.search(data);
            if (node == nullptr) {
                std::cout << "NoSuchWord" << '\n';
            } else {
                tree.remove(data);
                std::cout << "OK" << '\n';
            }
```

```
        } else if (command[0] == '!') {
            AVLTreeData action;
            std::cin >> action;
            AVLTreeData path;
            std::cin >> path;
            if (action[0] == 's') {
                std::ofstream file(path.key, std::ios::binary | std::ios::t
                size_t size = tree.getSize();
                file.write(reinterpret_cast<char*>(&size), sizeof(size_t));
                if (size > 0) {
                    tree.saveToFile(file);
                }
                std::cout << "OK\n";
                file.close();

            } else {
                std::ifstream file(path.key, std::ios::binary);
                tree.LoadFromFile(file);
                std::cout << "OK\n";
                file.close();

            }

        } else {
            node = tree.search(command);
            if (node == nullptr) {
                std::cout << "NoSuchWord" << '\n';
            } else {
                std::cout << "OK: " << node->data.value << '\n';
            }
        }
    }
    tree.destroy();
}
```

Дневник отладки

При тестировании программы была обнаружена ошибка - была неверно реализована перегрузка операторов сравнения строк для класса `AVLTreeData`, при сравнении строк разных длин результат был непредсказуем, поэтому было решено использовать встроенную в язык функцию `strcmp()`.

Тест производительности

Померим скорость работы кода лабораторной (будем записывать n-е количество элементов а потом удалять) и сравним с встроенной в C++ структурой `std::map`. Как видим, на больших значениях от 10^4 до 10^6 наш алгоритм идет наравне с библиотечным, это связано с тем, что в реализации `std::map` используется красно-черное дерево со временем операций вставки и удаления $O(log(n))$, где n - количество элементов в дереве. Такую же сложность имеют аналогичные операции в AVL-дереве.

Количество пар "ключ-значение"	TAVLTree, мс	std::map, мс
10000	139125	246289
100000	1193582	1654637
1000000	12535842	16596308

Ниже приведена программа benchmark.cpp, использовавшаяся для определения времени работы функций:

```
#include <iostream>
#include <random>
#include <string>
#include <algorithm>
#include <chrono>
#include <vector>
#include <map>

#include "main.cpp"

template <typename KeyT, typename ValueT>
struct test_pair {
    ValueT value;
    KeyT key;
    bool operator<(const test_pair& other) const {
        return key < other.key;
    }
    bool operator>(const test_pair& other) const {
        return key > other.key;
    }
    bool operator==(const test_pair& other) const {
        return key == other.key;
    }
    bool operator!=(const test_pair& other) const {
        return key != other.key;
    }
};
```

```
int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<uint8_t> key_dist;
    std::uniform_int_distribution<uint8_t> value_dist;
    std::uniform_int_distribution<> char_dist(0, 25);

    for (size_t k = 10000; k <= 10000000; k *= 10) {
        std::cout << k << " & ";
        std::map<uint8_t, uint8_t> map;
        TAVLTree<test_pair<uint8_t, uint8_t>> tree;
        test_pair<uint8_t, uint8_t> data;
        std::vector<test_pair<uint8_t, uint8_t>> vec;

        for (size_t i = 0; i < k; ++i) {
            data.value = key_dist(gen);
            data.key = value_dist(gen);
            vec.push_back(data);
        }

        auto start1 = std::chrono::high_resolution_clock::now();
        for (size_t i = 0; i < k; ++i) {
            tree.insert(vec[i]);

        }
        for (size_t i = 0; i < k; ++i) {
            tree.remove(vec[i]);
        }
        auto finish1 = std::chrono::high_resolution_clock::now();
        auto duration1 = std::chrono::duration_cast<std::chrono::microsecon
        std::cout << duration1.count() << " & ";

        auto start2 = std::chrono::high_resolution_clock::now();
        for (size_t i = 0; i < k; ++i) {
            map[vec[i].key] = vec[i].value;

        }
        for (size_t i = 0; i < k; ++i) {
            map.erase(vec[i].key);
        }
        auto finish2 = std::chrono::high_resolution_clock::now();
```

```
        auto duration2 = std::chrono::duration_cast<std::chrono::microsecon
        std::cout << duration2.count() << " \\" << "\\ ";

        std::cout << '\n';
        vec.clear();
        tree.destroy();
    }
}
```

Выводы

В ходе выполнения данной работы были изучены методы построения сбалансированных деревьев, в частности AVL-дерева. При написании дерева наибольшую сложность имела задача написания логики вставки и удаления с балансировкой. Судя по проведенным тестам можно сказать, что AVL-дерево - достаточно быстрая и эффективная структура для хранения ключей и быстрого доступа к данным по ключу.