

Московский авиационный институт  
(национальный исследовательский университет)

Факультет информационных технологий и прикладной  
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Дискретный анализ»

Студент: И. П. Попов  
Преподаватель: А. А. Кухтичев  
Группа: М8О-206Б  
Дата:  
Оценка:  
Подпись:

Москва, 2022

## Лабораторная работа №1

**Задача:** 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 264 - 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. Программа должна вывести строку «OK», если операция прошла успешно, «Exist», если слово уже находится в словаре.

- word — удалить слово «word» из словаря. Программа должна вывести «OK», если слово существовало и было удалено, «NoSuchWord», если слово в словаре не было найдено.

word — найти в словаре слово «word». Программа должна вывести «OK: 34», если слово было найдено; число, которое следует за «OK:» — номер, присвоенный слову при добавлении. В случае, если слово в словаре не было обнаружено, нужно вывести строку «NoSuchWord».

! Save /path/to/file — сохранить словарь в бинарном компактном представлении на диск в файл, указанный параметром команды. В случае успеха, программа должна вывести «OK», в случае неудачи выполнения операции, программа должна вывести описание ошибки (см. ниже).

! Load /path/to/file — загрузить словарь из файла. Предполагается, что файл был ранее подготовлен при помощи команды Save. В случае успеха, программа должна вывести строку «OK», а загруженный словарь должен заменить текущий (с которым происходит работа); в случае неуспеха, должна быть выведена диагностика, а рабочий словарь должен остаться без изменений. Кроме системных ошибок, программа должна корректно обрабатывать случаи несовпадения формата указанного файла и представления данных словаря во внешнем файле.

Для всех операций, в случае возникновения системной ошибки (нехватка памяти, отсутствие прав записи и т.п.), программа должна вывести строку, начинающуюся с `enquoteERROR:` и описывающую на английском языке возникшую ошибку.

AVL-дерево.

# 1 Описание

Интернет-ресурс [4] дает следующее описание AVL-деревьям:

АВЛ-дерево – структура данных, изобретенная в 1968 году двумя советскими математиками: Евгением Михайловичем Ландисом и Георгием Максимовичем Адельсон-Вельским. Прежде чем дать конструктивное определение АВЛ-дереву, сделаем это для сбалансированного двоичного дерева поиска.

Сбалансированным называется такое двоичное дерево поиска, в котором высота каждого из поддеревьев, имеющих общий корень, отличается не более чем на некоторую константу  $k$ , и при этом выполняются условия характерные для двоичного дерева поиска.

АВЛ-дерево – сбалансированное двоичное дерево поиска с  $k=1$ . Для его узлов определен коэффициент сбалансированности (balance factor). Balance factor – это разность высот правого и левого поддеревьев, принимающая одно значение из множества  $-1, 0, 1$ . Ниже изображен пример АВЛ-дерева, каждому узлу которого поставлен в соответствие его реальный коэффициент сбалансированности.

Сбалансированное дерево эффективно в обработке, что следует из следующих рассуждений. Максимальное количество шагов, которое может потребоваться для обнаружения нужного узла, равно количеству уровней самого бинарного дерева поиска. А так как поддерева сбалансированного дерева, «растущие» из произвольного корня, практически симметричны, то и его листья расположены на сравнительно невысоком уровне, т. е. высота дерева сводится к оптимальному минимуму. Поэтому критерий баланса положительно сказывается на общей производительности. Но в процессе обработки АВЛ-дерева, балансировка может нарушиться, тогда потребуется осуществить операцию балансировки. Помимо нее, над АВЛ-деревом определены операции вставки и удаления элемента. Именно выполнение последних может привести к дисбалансу дерева.

Доказано, что высота АВЛ-дерева, имеющего  $N$  узлов, примерно равна  $\log_2 N$ . Имея в виду это, а также то, что время выполнения операций добавления и удаления напрямую зависит от операции поиска, получим временную сложность трех операций для худшего и среднего случая –  $O(\log N)$ .

## 2 Исходный код

Программа разделена на три файла:

1. *main.cpp*
2. *avl.h*
3. *detailavl.h*

В файле *main.cpp* описан только интерфейс программы, основная логика же описана в двух других файлах.

В файле *avl.cpp* описана структура узла моего дерева:

```
1 struct TAvlNode {  
2     K key;  
3     V value;  
4     unsigned long long height;  
5     TAvlNode *left;  
6     TAvlNode *right;  
7     TAvlNode() : key(), value(), height{1}, left{nullptr},  
8                 right{nullptr} {};  
9     TAvlNode(K k, V v) : key{k}, value{v}, height{1},  
10                    left{nullptr}, right{nullptr} {};  
11 };
```

Реализованный класс AVL-дерева(TAvl) с основными методами: *Add*, *Delete*, *Find*. Но наиболее интересная часть реализации AVL-дерева скрывается внутри них, а именно балансировка после вставки и удаления элементов.

Относительно AVL-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев  $= 2$ , изменяет связи предок-потомок в поддереве данной вершины так, что разница становится  $\leq 1$ , иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

Реализована данная механика путем совершением деревом 4 видов различных поворотов: [2]:

1. Малое левое вращение AVL LR. Данное вращение используется тогда, когда высота b-поддерева — высота L  $= 2$  и высота C  $\leq$  высота R.
2. Большое левое вращение AVL BR. Данное вращение используется тогда, когда высота b-поддерева — высота L  $= 2$  и высота c-поддерева  $>$  высота R.

3. Малое правое вращение AVL LL. Данное вращение используется тогда, когда высота b-поддерева — высота R = 2 и высота C ≤ высота L.
4. Большое правое вращение AVL BL. Данное вращение используется тогда, когда высота b-поддерева — высота R = 2 и высота c-поддерева > высота L.

В каждом случае достаточно просто доказать то, что операция приводит к нужному результату и что полная высота уменьшается не более чем на 1 и не может увеличиться. Также можно заметить, что большое вращение это комбинация правого и левого малого вращения. Из-за условия балансированности высота дерева  $O(\log(N))$ , где N- количество вершин, поэтому добавление элемента требует  $O(\log(N))$  операций.

```

1 ull Height(const TNode *node) {
2     return node != nullptr ? node->height : 0;
3 }
4
5 int Balance(const TNode *node) {
6     return Height(node->left) - Height(node->right);
7 }
8
9 void Reheight(TNode *node) {
10     node->height = std::max(Height(node->left), Height(node->right)) + 1;
11 }
12
13 TNode *RotateLeft(TNode *a) {
14     TNode *b = a->right;
15     a->right = b->left;
16     b->left = a;
17     Reheight(a);
18     Reheight(b);
19     return b;
20 }
21
22 TNode *RotateRight(TNode *a) {
23     TNode *b = a->left;
24     a->left = b->right;
25     b->right = a;
26     Reheight(a);
27     Reheight(b);
28     return b;
29 }
30
31 TNode *RLrotate(TNode *a) {
32     a->right = RotateRight(a->right);
33     return RotateLeft(a);
34 }
35

```

```

36 TNode *LRrotate(TNode *a) {
37     a->left = RotateLeft(a->left);
38     return RotateRight(a);
39 }
40
41 TNode *Rebalance(TNode *node) {
42     if (node == nullptr) {
43         return nullptr;
44     }
45     Reheight(node);
46     int balanceRes = Balance(node);
47     if (balanceRes == -2) {
48         if (Balance(node->right) == 1) {
49             return RLrotate(node);
50         }
51         return RotateLeft(node);
52     }
53     else if (balanceRes == 2) {
54         if (Balance(node->left) == -1) {
55             return LRrotate(node);
56         }
57         return RotateRight(node);
58     }
59     return node;
60 }

```

В файле *detailavl.h* описан алгоритм сохранения и загрузки из бинарного файла объектов моего класса *TDetailAvl*, который является наследником ранее описанного класса *TAvl*. Именно с этим классом взаимодействует пользователь.

### 3 Консоль

tmp:

```

+ a 1
+ A 2
+ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
18446744073709551615
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
A
-A
a

```

console output:

```
root@Lunidep:~/DA/da_lab2/src# ./da_lab2 <tmp
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```



## 4 Тест производительности

Тест представлял из себя сравнение реализованного мной класса AVL-дерева с `std::map`, который реализован на красно-черном дереве.

Задачей было построить деревья из  $n$  различных элементов, сделать поиск  $n$  различных значений в этих деревьях, удалить  $n$  различных элементов (элементы сразу же добавлялись обратно, но отслеживалось только время удаления). В результате работы *benchmark.cpp* видны следующие результаты:

```
root@Lunidep:~/DA/da_lab2# ./benchmark
-----numder_of_nodes = 1000 -----
Insert:
std::map ms=75
avl ms=28

Find:
std::map ms=1
avl ms=0

Delete:
std::map ms=5
avl ms=3

-----numder_of_nodes = 10000 -----
Insert:
std::map ms=518
avl ms=3060

Find:
std::map ms=202
avl ms=45

Delete:
std::map ms=165
avl ms=201

-----numder_of_nodes = 100000 -----
Insert:
std::map ms=5940
avl ms=90917
```

Find:

std::map ms=2535

avl ms=822

Delete:

std::map ms=6610

avl ms=17866

-----number\_of\_nodes = 1000000 -----

Insert:

std::map ms=96845

avl ms=996143

Find:

std::map ms=43218

avl ms=11257

Delete:

std::map ms=118482

avl ms=821968

Из примера видно, что AVL-дерево проявляет себя в разы эффективнее при поиске элементов, однако уступает в эффективности вставки и удаления (в связи с необходимостью постоянно перестраиваться).

## 5 Выводы

Выполнив первую лабораторную работу по курсу «Дискретный анализ», мною были изучены различные виды деревьев поиска и реализован один из них - AVL-дерево.

Использование этих структур данных позволят делать программы более эффективными, что играет тем большую роль, чем большие данные подвергаются обработке.

AVL-дерево показывает отличные результаты при решении задач поиска элемента, однако и остальные операции (добавления и удаления) происходят довольно быстро, поскольку напрямую зависит от операции поиска, временную сложность –  $O(\log N)$ .

## Список литературы

- [1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание*. — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))
- [2] *АВЛ-дерево* — *Википедия*.  
URL: <https://ru.wikipedia.org/wiki/АВЛ-дерево> (дата обращения: 14.04.2022).
- [3] Список использованных источников оформлять нужно по ГОСТ Р 7.05-2008
- [4] *АВЛ-дерево* — *Kvodo*.  
URL: <https://kvodo.ru/avl-tree.html> (дата обращения: 14.04.2022).