# 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: М. В. Леухин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-20

Дата:
Оценка:
Подпись:


**Москва, 2022**

# Лабораторная работа №2

**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 парамером 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, отсутсвие прав записи и т.п.), программа должна вывести строку, начинающуюся с «ERROR:» и описывающую на английском языке возникшую ошибку.

Различия вариантов заключаются только в используемых структурах данных: `AVL-дерево.`

# 1 Описание

Требуется написать реализацию AVL-дерева. Данная структура является сбалансированным бинарным деревом поиска, то есть оно обладает следующими свойствами:

1. У каждого элемента есть атрибуты $left$, $right$ и $p$, которые указывают на левый и правый дочерние узлы и родительский узел соответственно (могут иметь значение NIL) [1].

2. Если $x$ — узел бинарного дерева поиска, а $y$ — узел в левом поддереве, то $y.key < x.key$. Если же $y$ — узел в правом поддереве, то $y.key > x.key$ [1].

3. Структура сбалансирована: для каждой его вершины высота её двух поддеревьев различается не более чем на 1 [2].

Первые два свойства характеризуют бинарное дерево поиска. В общем случае для бинарного дерева поиска сложность поиска, вставки и удаления равна $O(h)$, причём $h$ принимает значения от $\log n$ до $n$. Но благодаря вышеописанному 3 свойству высота AVL-дерева всегда $\log n$ и сложность поиска, вставки и удаления всегда $O(\log n)$. Это достигается тем, что вставка и удаление элементов меняют структуру дерева, но не нарушают сбалансированности. Для каждой вершины вводится дополнительный аттрибут $balance$, который равен разности высот левого и правого поддерева. Если при вставке или удалении случилась такая ситуация, что у вершины значение $balance$ стало равно 2 или -2, то производится ребалансировка путём поворота поддеревьев.

# 2 Исходный код

AVL-дерево является рекурсивной структурой — воспользуемся этим при его реализации. Непосредственно дерево представляется классом $TAVLTree$, в котором имеется поле $root$, представленное классом $TNode$. Этот класс реализовывает модель вершины дерева и имеет поля $balance$, $parent$, $left$ и $right$. Такое разделение на классы произведено с двумя целями:

1. В определённый момент времени дерево может быть пустым — значение $root$ равно $NIL$. Любое прямое обращение к нему приводило бы к ошибке.
2. Методы класса $TNode$ имеют дополнительные параметры, необходимые для их рекурсивного описания. Например, метод $Insert$, помимо самого значения $valueToInsert$ имеет дополнительный параметр — логическую переменную $flag$, по которой определяется, была ли произведена вставка или же такой элемент в дереве уже существует.

Использование дополнтиельной структуры $TAVLTree$ позволяет решить обе этих проблемы.

Перейдём к описанию поиска, вставки и удаления элементов.

- Поиск. Поиск в AVL-дереве производится точно так же, как и в BST. Запускаем поиск от корня и на каждом этапе смотрим, совпадает ли ключ вершины с ключом, по которому мы ищем элемент. Если да, то элемент найден — поиск закончен. Иначе сравниваем ключ вершины с ключом поиска. Если ключ вершины больше, то проверяем, есть ли у вершины левое поддерево: да — запускаем поиск в нём, нет — элемента в дереве нет. Аналогично для случая, когда ключ вершины меньше ключа поиска, но работаем уже с правым поддеревом.
- Вставка. На первом этапе производим поиск переданного ключа — такой элемент в дереве уже может существовать. В таком случае вставка заканчивается, ничего не изменив. Если же в процессе поиска мы не нашли элемента стаким ключом, то создаём новую вершину и связываем её с деревом (в процессе поиска мы уже нашли её место — она станет поддеревом вершины, на которой мы завершили поиск, не найдя у неё левого или правого поддерева). Конечным этапом будет ребалансировка: баланс родительской вершины созданного поддерева меняется в зависимости от того, каким поддеревом является созданный нами элемент. Если левым, то баланс родительской вершины увеличивается на 1, иначе уменьшается на то же значение. Так как дерево, куда производилась вставка, являлось сбалансированным, новый баланс родительской вершины может принять следующие значения:

1. 0 — ничего не делаем, так как высоты левого и правого поддеревьев выровнялись, а, следовательно, нигде выше баланс не нарушен.
2. 1 или -1 — в этом случае необходимо рекусивно предолжить ребалансировку для родительской вершины, так как для неё баланс мог нарушиться.
3. 2 или -2 — необходимо производить повороты для восставновления баланса. Вид поворота определяется значениями баланс нашей вершины и одного из её поддеревьев. Например, если баланс вершины равен 2, а её левого поддерева — 1, то нужно сделать один правый поворот относительно нашей вершины и её левого поддерева.

- Удаление. Первый этап удаления такой же, как и в BST: находим удаляемый элемент, и если он есть, то выбираем один из 3 вариантов в зависимости от количества поддеревьев у удаляемого элемента. Если их нет вообще — просто удаляем вершину и у родителя значение соответствующего поддерева делаем NIL. Если есть один — связываем поддерево удаляемого элемента с соответствующим поддеревом родительской вершины, после чего удаляем вершину. Если есть два — находим минимальный элемент в правом поддереве, удаляем его, а значение записываем в нашу такущую вершину. Следующий этап — ребалансировка, работающая по такому же принципу, как и при вставке.

| TNode | |
|---|---|
| void Print(int height) | Рекурсивная функция печати вершины дерева |
| void Insert(TAVLTree *tree, const TPair &valueToInsert, bool &flag) | Рекурсивная функция вставки элемента $valueToInsert$ |
| void Remove(TAVLTree *tree, const string &key, bool &flag) | Рекурсивная функция удаления элемента с ключом $key$ |
| unsigned long long Find(const string &key, bool &flag) | Рекурсивная функция поиска элемента с ключом $key$ |
| void Save(ofstream &file) | Рекурсивная фнукция записи вершины в $file$ |
| void Clear() | Рекурсивная функция очищения вершины |
| void Rebalance(TAVLTree *tree, const string &key) | Рекурсивная функция ребалансировки при вставке |
| void RemovalRebalance(TAVLTree *tree, const string &key) | Рекурсивная функция ребалансировки при удалении вершины |
| TPair FindMinimumAndRemove(TAVLTree *tree) | Функция поиска и удаления наименьшего элемента |
| void LeftRotate(TAVLTree *tree) | Функция левого поворота |

| | |
|---|---|
| void RightRotate(TAVLTree *tree) | Функция правого поворота |
| TAVLTree | |
| void SetRoot(TNode *root) | Функция установки значения корня дерева |
| void Print() | Функция печати дерева |
| string Insert(TPair &value) | Функция вставки элемента *value* в дерево |
| string Remove(const string &key) | Функция удаления элемента с ключом *key* |
| string Find(const string &key) | Функция поиска элемента с ключом *key* |
| void Save(ofstream &file) | Функция сохранения дерева в *file* |
| void Load(ifstream &file) | Функция загрузки дерева из *file* |
| void Clear() | Функция удаоения дерева |

```
1  class TNode {
2  public:
3
4      TNode(TPair value, TNode *parent);
5
6      void Print(int height);
7
8      void Insert(TAVLTree *tree, const TPair &valueToInsert, bool &flag);
9
10     void Remove(TAVLTree *tree, const string &key, bool &flag);
11
12     unsigned long long Find(const string &key, bool &flag);
13
14     void Save(ofstream &file);
15
16     void Clear();
17
18 private:
19     TPair value;
20
21     int balance = 0;
22     TNode *parent = nullptr;
23     TNode *left = nullptr;
24
25     TNode *right = nullptr;
26
27     void Rebalance(TAVLTree *tree, const string &key);
28
29     void RemovalRebalance(TAVLTree *tree, const string &key);
30
```

```
31      TPair FindMinimumAndRemove(TAVLTree *tree);
32
33      void LeftRotate(TAVLTree *tree);
34
35      void RightRotate(TAVLTree *tree);
36
37  };
38
39
40  class TAVLTree {
41  private:
42
43      TNode *root = nullptr;
44
45  public:
46
47      void SetRoot(TNode *root);
48
49      void Print();
50
51      string Insert(TPair &value);
52
53      string Remove(const string &key);
54
55      string Find(const string &key);
56
57      void Save(ofstream &file);
58
59      void Load(ifstream &file);
60
61      void Clear();
62
63  };
```

# 3 Консоль

```
matvey@matvey-Lenovo-IdeaPad-S340-15API:~/labs/3da/2lab$ g++ -std=c++17 result.cpp
-o lab2
matvey@matvey-Lenovo-IdeaPad-S340-15API:~/labs/3da/2lab$ cat tests/01.t
+ a 1
+ A 2
+ aa 18446744073709551615
aa
A
-A
matvey@matvey-Lenovo-IdeaPad-S340-15API:~/labs/3da/2lab$ ./lab2 <tests/01.t
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

# 4 Тест производительности

Тест производительности представляет из себя следующее: сравнение эффективности реализованного AVL-дерева и BST.

```
# 100000 lines
matvey@matvey-Lenovo-IdeaPad-S340-15API:~/labs/3da/2lab$ ./avl <tests/01.t
AVL tree time: 31828us
matvey@matvey-Lenovo-IdeaPad-S340-15API:~/labs/3da/2lab$ ./bst <tests/01.t
BST tree time: 36357us
```

В приведённом выше тесте команды генерировались случайным образом, но уже на них видно, что AVL-дерево работает быстрее. Более наглядно разницу между AVL и BST можно увидеть на таких тестах, в которых стуктура элементов в дереве целенаправлено должна быть распределена неравномерно (худший случацй — BST вырождается в линейный список). На таких данных AVL-дерево рабоатет за 4.153 секунды, а BST — дольше 15 секунд.

# 5 Выводы

Выполнив вторую лабораторную работу по курсу «Дискретный анализ», я познакомился с такой структурой данных, как AVL-дерево, и научился его реализовывать. Я узнал о том, чем может быть неудобно обычное бинарное дерево поиска и зачем нужны сбалансированные деревья, как именно производится балансирвока в AVL-дереве и что такое левый и правый поворот. Помимо AVL-дерева, мною были изучены и другие сбалансированные структуры: красно-чёрное дерево, B-дерево, PATRICIA.

# 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] *AVL-дерево — Википедия.*
URL: `https://ru.wikipedia.org/wiki/АВЛ-дерево` (дата обращения: 22.04.2022).