# Лабораторная работа №2 по курсу Дискретного Анализа: Сбалансированные деревья

Выполнил студент группы М8О-207Б-21 МАИ *Кажекин Денис*.

## 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. 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.

Различия вариантов заключаются только в используемых структурах данных:

**B-дерево.**

## Метод решения / Описание программы

Для выполнения поставленной задачи было необходимо нулевым шагом определить структуру данных, которая будет представлять узел нашего дерева. Так как нам необходимо реализовать словарь, в котором каждому ключу сопоставляется свое значение, то для начала определим структуру пары (Pair), а затем и структуру самого узла дерева (Node), которая будет содержать массив структур пар, массив указателей на сыновей (в B-tree у нас количество ключей и сыновей зависит от параметра “T”), булевскую переменную (leaf), которая будет говорить, является ли узел листом, а также переменную количества ключей в узле. Помимо самой структуры удобно будет определить конструктор и деструктор (для предотвращения утечек памяти). После определения основных структур необходимо реализовать функции по работе с нашим деревом: поиск, вставку, удаление, сериализацию и десериализацию. Операции вставки и удаления сопровождаются несколькими вспомогательными функциями, так как структура у них достаточно сложная. Поиск работает очень примитивно (обобщение поиска для бинарного дерева поиска), сериализация осуществляется рекурсивным обходом дерева в глубину и записью в файл основной информации об узле, а именно (ключи/значения), потому что десериализовать дерево мы сможем, просто считывая наши ключи/значения и вызывая функцию вставки. Таким образом, заново построится сериализованное дерево. Хочу отметить, что я использовал параметр дерева по Кнуту (Knuth’s order)! То есть минимальное и максимальное количество элементов в узле дерева определяется следующим образом: если у нас параметр равен T, то минимальное количество элементов во внутреннем и листовом узле будет: ceil(T/2) – 1, а максимальное: T – 1. Для корня минимальная граница, очевидно, 1. Также хочется сказать, что я использовал функцию бинпоиска индекса массива для определения места куда необходимо вставить элемент, так как на практике берутся достаточно большие значения “T”.  P.S. При использовании утилиты GPROF для анализа времени работы программы было замечено, что программа 50% времени при исполнении проводит в функции бинпоиска. Это говорит о том, что не зря была сделана эта доработка.

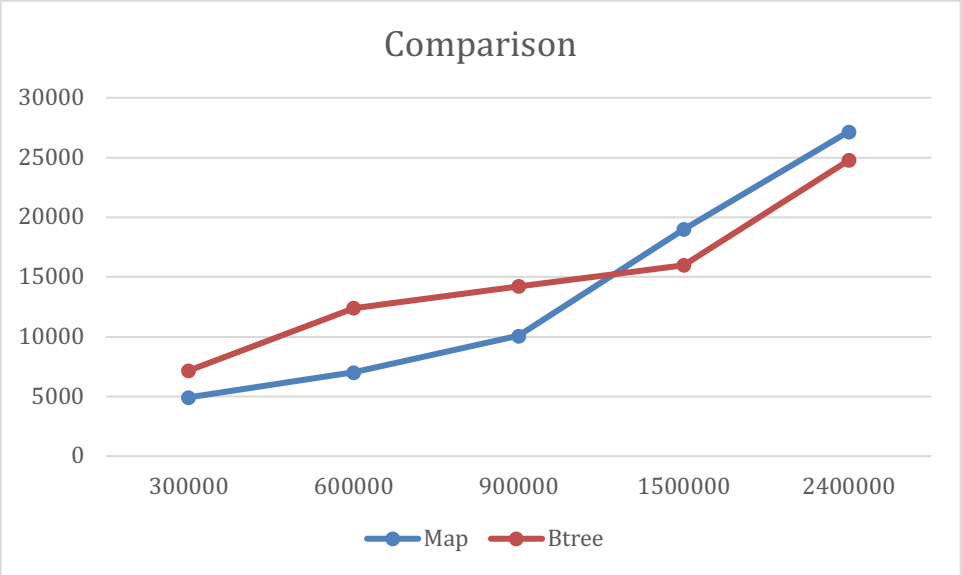
## Дневник отладки

1) Ошибка RE на пятом тесте была устранена с помощью корректировки параметра “T” B-дерева.

2) Ошибка WA на пятом тесте была устранена после корректировки вывода в консоль в функции “FindNode” (программа работала корректно, я неверно написал вывод)

## Тест производительности

Тесты производительности проведем в сравнении со встроенной структурой данных словарь - <Map>. На оси X изображены размеры тестов, а по оси Y – время, затраченное на выполнение в миллисекундах. Тесты состояли из одинакового количества операций добавления, поиска и удаления. То есть график ниже отображает общее сравнение производительности B-tree с Map.




Таким образом, можно увидеть, что на данных, размер которых превышает 1 млн «B-tree» показывает лучшую производительность, в то время как на данных меньше 1 млн «Map» оказывается быстрее.

## Недочёты

Не выявлены

## Выводы

В ходе выполнения лабораторной работы я изучил и реализовал структуру «B-дерево», сравнил производительность со встроенным <Map>.

B-дерево очень часто используется для проектирования баз данных, потому что оно позволяет минимизировать количество обращений к жесткому диску, так как считывание с жесткого диска происходит в разы дольше чем работа с оперативной памятью. Эта минимизация достигается путем хранения сразу нескольких ключей в одном узле (в зависимости от параметра, параметр зачастую принимается очень большим). Тогда мы буквально за 1-2 перехода к сыновьям всегда будем получать наш искомый элемент, который хотим найти.

Операции вставки, поиска, удаления выполняются за O(log(n))