Лабораторная работа № 2 по курсу дискретного анализа: сбалансированные деревья

Выполнил студент группы 08-208 МАИ *Былькова Кристина.*

Условие

Общая постановка задачи: 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. — 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.

Вариант структуры данных: AVL-дерево

Метод решения

Моя программа обрабатывает строки входного файла до его окончания, при этом я разделяю ввод команды, ключа и значения. В зависимости от команды обрабатываю следующие входные данные. Например, если команда = "+ далее я считываю ключ и значение, проверяю, есть ли такое слово в моём словаре. Если нет - добавляю в дерево, иначе вывожу сообщение в соответствие с требованиями. Прежде всего, AVL-дерево — это сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Из-за этого при всех манипуляциях с деревом, необходимо проверять это свойство. И в случае его невыполнения производить балансировку, путем поворотов. Для этого у меня написаны такие функции, как:

- int BFactor(TNode *tree) – возвращает разницу между высотам поддеревьев
- TNode *RightRotate(TNode *tree) – поворот правого поддерева
- TNode *LeftRotate(TNode *tree) – поворот левого поддерева
- TNode *Balance(TNode *tree) – балансировка дерева

Поиск в AVL-дереве основан на поиске в обычном двоичном дереве поиска: спускаемся по дереву и сравниваем значение искомого узла с текущим. В зависимости от сравнения, идем по левому или правому поддереву.

Вставка в AVL-дерево основана на том, что мы спускаемся по дереву, на каждом шаге сравнивая значение нового узла с текущими. Затем, доходим до конца какого-либо поддерева и делаем новый узел правым или левым его потомком в зависимости от значения. При этом не забываем про балансировку.

Для удаления узла из AVL-дерева сначала в дереве ищем узел, который нужно удалить. Если такого узла нет, ничего не делается. Если узел находится, то надо пройти по правому поддереву удаляемого узла и найти в нем узел с самым маленьким значением — min. После этого удаляемый узел нужно заменить на узел min, и структура дерева перестроится. Если правого поддерева у удаляемого узла нет, вместо min на место узла подставляется его левый потомок. Если левого потомка тоже нет, значит, удаляемый узел — лист, то есть потомки у него отсутствуют в принципе. Тогда его можно просто удалить и ничего не подставлять на его место. После всех этих манипуляций, также не забываем про балансировку.

Описание программы

Были написаны два класса и структура:

- class TString – собственная реализация строки
- class TAVLTree – реализация AVL-дерева

- struct TNode – структура вершины дерева, предназначенная в том числе для хранения пары ключ-значение.

Для выполнения задания я использовала:

- TAVLTree<TString, unsigned long long> tree – словарь в виде AVL-дерева, где хранятся пары ключ-значения.

Дневник отладки

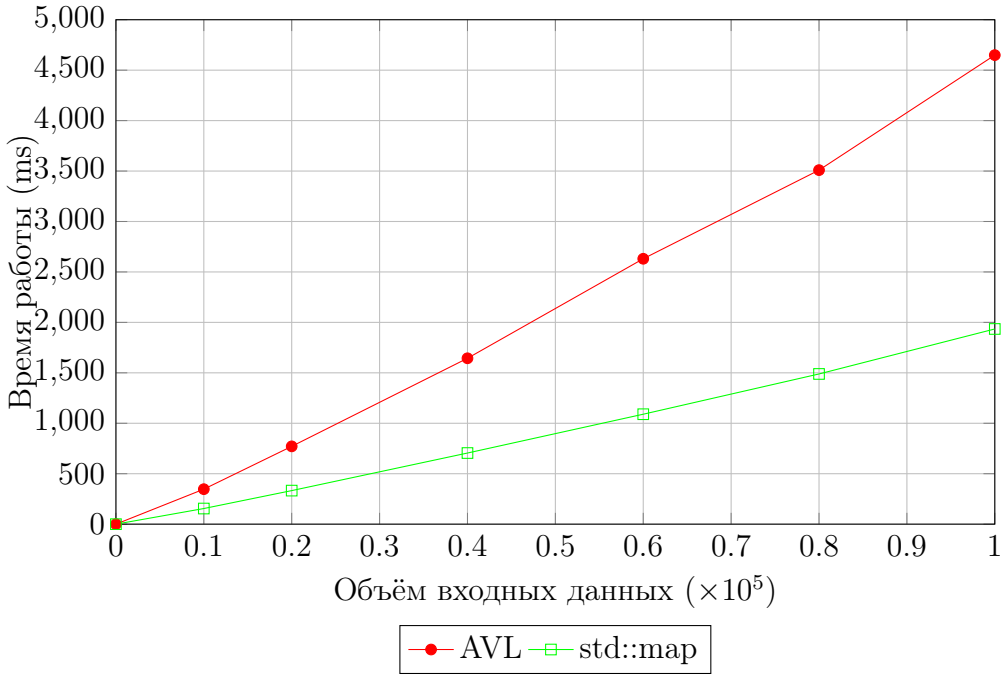
Основной проблемой было то, что я невнимательно прочитала задание и неверно считывала входные данные, что приводило к зацикливанию программы, а, следовательно, к превышению лимита времени. Также небольшую сложность вызвало написание оператора для ввода в собственном классе TString.

Тест производительности

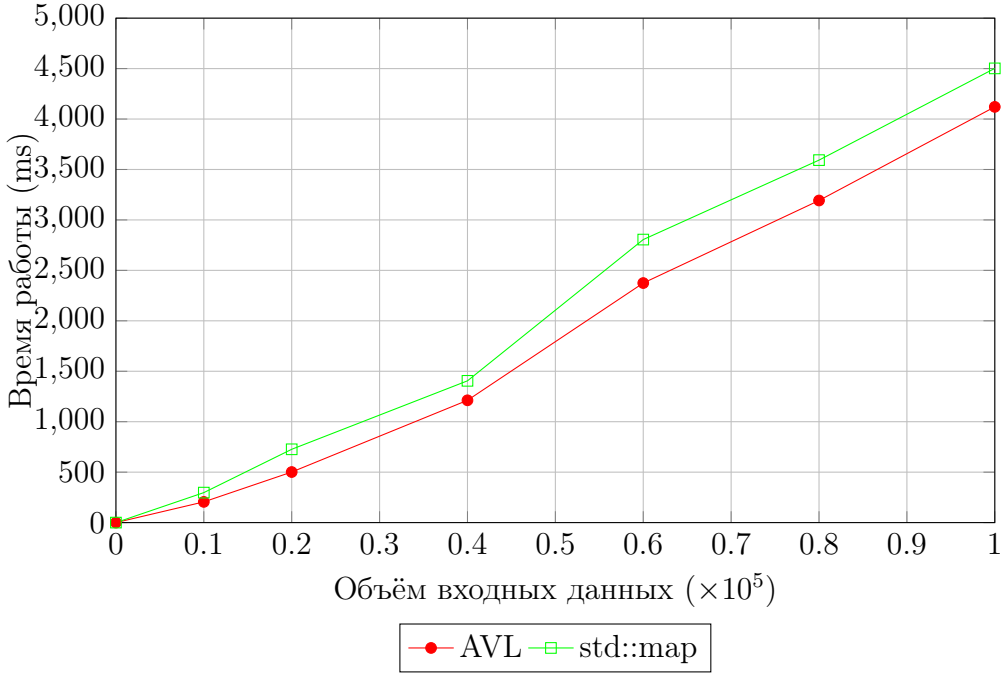
Для проверки производительности моего словаря в виде AVL-дерева я использовала сравнение со стандартным контейнером std::map. Это имеет смысл так как в его основе лежит красно-черное дерево, которое тоже является сбалансированным. Сравнение производилось на входных данных больших размеров, не превышающих 10^5.

Исходя из графика, представленного ниже, можно увидеть, что вставка (аналогично с удалением) элементов в std::map работает быстрее, чем в AVL-дереве, а с поиском ситуация противоположна. Это происходит в силу различий в алгоритмах балансировки: в красно-черных деревьях она происходит гораздо реже, чем в AVL-дереве. Как следствие вставка (удаление) происходит быстрее. Однако высота красно-черного дерева больше высоты соответствующего AVL-дерева. Отсюда вытекает большее время поиска. Не логарифмическая зависимость на графиках, по моему мнению, получается из-за довольно частых выделений/освобождений памяти на кучи, а это, как известно, не дешевая в плане времени операция. К тому же графики имеют очень похожую форму, что говорит об асимптотически одинаковой скорости роста времени выполнения операций.

Вставка



Поиск



Выводы

В результате данной лабораторной работы была написана и отлажена программа на языке C++, реализующая AVL-дерево, на основе которого разработана программа-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 $2^{64} - 1$.

Сложность моей реализации словаря на основе AVL-дерева близка к $O(nlog(n))$, где n – высота дерева, что объективно хуже, что $O(log(n))$. Мне кажется, это связано с тем, что я довольно часто выделяю/освобождаю память на куче, что снижает эффективность программы. Также я храню строки, у которых тоже происходит частое выделение/освобождение памяти.

В общем случае, AVL-деревья очень важны и применяются довольно часто. Благодаря сбалансированности и борьбе с вырождением дерева информация в нем хранится более эффективно. Поэтому доступ к данным оказывается быстрее и найти их становится легче.