Лабораторная работа № 2 по курсу дискретного анализа: сбалансированные деревья

Выполнила студентка группы 08-208 МАИ *Шевлякова София.*

Условие

Реализовать декартово дерево с возможностью поиска, добавления и удаления элементов. 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».

Метод решения

Декартово дерево (Treap) — это структура данных, объединяющая в себе бинарное дерево поиска и кучу. Более строго, это бинарное дерево, в узлах которого хранятся пары (x,y), где x — это ключ, а y — это приоритет. Основными операциями декартова дерева является merge и split.
Операция split позволяет сделать следующее: разрезать исходное дерево T по ключу k. Возвращать она будет такую пару деревьев $\langle$T1,T2$\rangle$, что в дереве T1 ключи меньше k, а в дереве T2 все остальные: split(T,k)$\rightarrow\langle$T1,T2$\rangle$.
Рассмотрим вторую операцию с декартовыми деревьями — merge. С помощью этой операции можно слить два декартовых дерева в одно. Причём, все ключи в первом(левом) дереве должны быть меньше, чем ключи во втором(правом). В результате получается дерево, в котором есть все ключи из первого и второго деревьев: merge(T1,T2)$\rightarrow$T
Используя эти 2 функции, мы можем реализовать основные операции с деревом:

- Операция поиска. Используем операцию split два раза: сначала по нашему ключу x, а потом правое дерево по ключу x+1. Так мы получим три дерева, в первом все элементы строго меньше x, в третьем строго больше x, а второе дерево может быть или пустым, или содержать единственный элемент x. Для поиска можно просто проверить, что второе дерево не пустое, и вывести его значение, в противном случае будем выводить "NoSuchWord". После этого применяем операцию merge два раза, чтобы вернуться к исходному дереву. Все остальные операции построены аналогично.

- Операция вставки. Проверяем, что второе дерево пустое, значит такого элемента пока не существует, а значит, мы можем создать в этом дереве ноду с полученным от пользователя ключом и значением. В противном случае выводим "Exist".

- Операция удаления. Проверяем, что второе дерево не пустое, значит, мы можем удалить этот элемент, сначала удаляем ноду, а потом указателю на ноду присваиваем значение null. Если дерево оказалось пустым, то выводим "NoSuchWord".

Описание программы

В данной программе содержится один файл main.cpp, в котором реализованы операции поиска, вставки и удаления.
Структура дерева содержит 2 поля:

- node_ptr root — ссылка на корень дерева

- char tmp[KEY_LEN] — временный ключ, который идет сразу за ключом x в лексикографическом порядке, благодаря ему мы будем делить исходное дерево на три других

Структура ноды дерева состоит из полей:

- node *left — указатель на левого ребенка

- node *right — указатель на правого ребенка

- char *key — строка, хранящее значение ключа

- uint64_t value — значение

- int64_t y — рандомное значение приоритета, которое используется для балансировки дерева

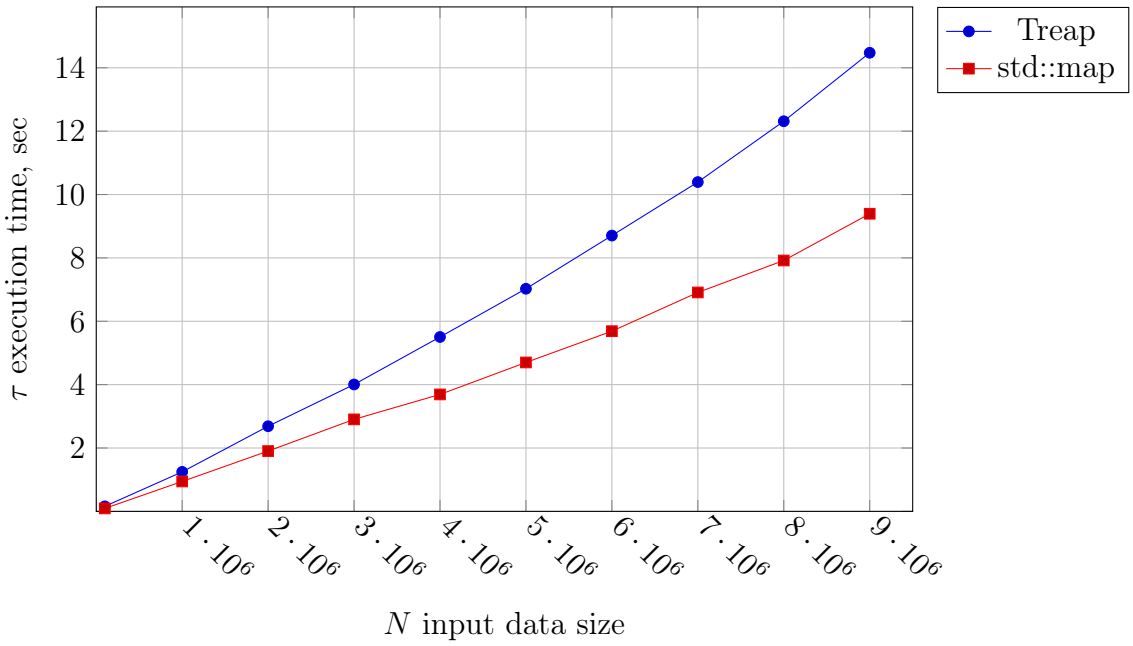
Для упрощения программы реализованы дополнительные функции: void cut(node_ptr t, node_ptr leftTree, node_ptr middle, node_ptr rightTree, const char* _key) - делает два раза split, сначала для исходного дерева root по ключу _key, разделяя его на leftTree и временное дерево r, а потом второй раз split дерева r по ключу tmp, который на

единицу больше ключа _key, получая middle и rightTree. Дерево middle или является пустым, или содержит единственную ноду с ключом _key. Функция join(node_ptr leftTree, node_ptr middle, node_ptr rightTree) выполняет обратную функцию: делает два раза merge, сначала middle и rightTree, получая дерево r, а потом merge для leftTree и r, возвращаюсь к исходному дереву.

Дневник отладки

№	Описание ошибки	Способ устранения
1	WA1, ключи, отличающиеся только регистром считались разными, что противоречило условию задачи.	Для исправления это ошибки я написала функцию StringToLower, которая переводила ключ в нижний регистр.

Тест производительности



Оценка сложности операций вставки, поиска и удаления: O(h), где h — высота дерева, так как дерево является сбалансированным, то сложность операций можно представить как O(logn), где n — количество элементов. Значит, сложность всей программы оценивается как O(n·logn). Для сравнения используется стандартная библиотека std::map, которая реализована на основе красно-черного дерева.

Выводы

В данной лабораторной работе было предложено изучить некоторые виды алгоритмов сбалансированных деревьев. Мной был реализован алгоритм декартово дерево. Операции вставки, поиска и удаления выполняются за временную сложность O(logn), где n — количество элементов. Также мной были изучены дополнительные операции merge и split, которые помогают реализовать операции поиска, вставки и удаления.
Я считаю, что эта лабораторная работа оказалась достаточно полезной. Ведь сбалансированные деревья применяется, когда необходимо осуществлять быстрый поиск элементов, чередующих со вставками новых элементов и удалениями существующих.