# **Лабораторная работа №2** по курсу дискретного анализа: строковые алгоритмы.

**Выполнил**: студент группы М8О-307Б-23
Дубровина Софья Андреевна

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 $2^{64}-1$. 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».

Метод решения:

Для решения задачи использована структура данных — **декартово дерево (Treap)**, представляющее собой комбинацию бинарного дерева поиска (BST) и кучи. Treap обеспечивает эффективное выполнение операций поиска, вставки и удаления элементов благодаря сочетанию упорядоченности по ключам (как в BST) и балансировки с помощью случайных приоритетов (как в куче). В данной работе словарь сопоставляет регистронезависимые строки длиной до 256 символов с 64-битными целыми числами (от 0 до $2^{64}-1$).

Программа обрабатывает команды из входного потока до его окончания. Поддерживаются следующие типы команд:

- **+ word value** — добавление слова с заданным значением; выводит «OK» при успехе или «Exist», если слово уже есть.

- **- word** — удаление слова; выводит «OK» при успехе или «NoSuchWord», если слово отсутствует.

- **word** — поиск слова; выводит «OK: value» при нахождении или «NoSuchWord» в противном случае.

Основные особенности алгоритма:

- **Регистронезависимость.** Все символы слов приводятся к нижнему регистру, что позволяет обрабатывать, например, «Word» и «word» как одинаковые ключи.

- **Операции Treap.** Используются функции Split (разделение дерева), Merge (объединение деревьев), Insert (вставка узла), Remove (удаление узла) и Find (поиск узла). Эти операции обеспечивают корректное и сбалансированное управление структурой.

- **Случайные приоритеты.** Каждому узлу присваивается случайный приоритет, генерируемый собственным xorshift-алгоритмом. Это в среднем гарантирует балансировку дерева и логарифмическую сложность операций.

Архитектура программы представляет собой однопоточное приложение на C++, реализованное в одном файле. Входные команды читаются из стандартного ввода, обрабатываются с использованием Treap, а результаты выводятся в консоль. Ожидаемая сложность операций вставки, удаления и поиска составляет **O(log n)** в среднем, где n — количество узлов в дереве.

## Описание программы

Программа реализована в одном файле main.cpp.

**Основные типы данных**

- **Класс TNode** (узел дерева):
    - Key: динамически выделяемая строка (массив символов длиной до 256 + '\0').
    - Value: 64-битное целое число (unsigned long long), связанное с ключом.
    - Prior: случайный приоритет (unsigned int), используемый для балансировки.
    - Left, Right: указатели на левое и правое поддерево.
- **Класс TTreap**: реализует операции с декартовым деревом и хранит корень дерева.

**Основные функции**

- CompareStrings(const char a, const char b):** выполняет регистронезависимое сравнение двух строк, приводя символы к нижнему регистру. Возвращает отрицательное, нулевое или положительное значение в зависимости от лексикографического порядка.

- Split(Node root, const char key, Node*& left, Node*& right):** разделяет дерево на два поддерева: левое содержит ключи, меньшие key, правое — ключи, большие или равные. Используется при вставке и удалении.

- Merge(Node left, Node right):** объединяет два поддерева, выбирая корень с большим приоритетом, сохраняя свойства дерева поиска и кучи.

- Insert(char key, unsigned long long value):* вставляет новый узел. Если ключ уже существует, возвращается сообщение «Exist».

- Remove(const char key):* удаляет узел с данным ключом, объединяя поддеревья через Merge. Если ключ не найден, возвращает «NoSuchWord».

- Find(const char key):* ищет узел с данным ключом. При успехе возвращает значение, иначе — сообщение «NoSuchWord».

- ReadWord, ReadUnsignedLongLong, SkipHSpaces, SkipToLineEnd: вспомогательные функции для разбора ввода, обеспечивающие корректное чтение слов и чисел, игнорирование регистра, пробелов и лишних символов.

- main(): основная функция, организующая цикл чтения команд из стандартного ввода, вызов соответствующих методов TTreap и вывод результата в нужном формате.

**Логика работы программы**

1. **Инициализация**
    - Создаётся пустое декартово дерево TTreap.
    - Подготавливается буфер для ключа (до 256 символов).
2. **Чтение входа**
    - Программа читает поток построчно до EOF (обычно через перенаправление ./dict < input.txt > output.txt).
    - В начале каждой итерации пропускаются пустые строки и служебные пробелы/табы.
3. **Определение типа команды**
    - Если первый символ строки — '+': ожидаются **слово** и **число**.
    - Если '-': ожидается **слово**.
    - Если буква: считается командой **поиска** (одно **слово**).
    - Иной первый символ — игнорируется (считывается и переход к следующей итерации).
4. **Парсинг аргументов**
    - **Слова** читаются функцией ReadWord: посимвольно, только латинские буквы, в процессе приводятся к **нижнему регистру** (регистронезависимость).
    - **Числа** читаются ReadUnsignedLongLong (диапазон $0..2^{64}-1$).
    - Между токенами пропускаются **горизонтальные** пробелы/табы (SkipHSpaces), чтобы не «перепрыгивать» на

следующую строку.

- Остаток текущей строки всегда отбрасывается (SkipToLineEnd), чтобы игнорировать мусор в конце (например, комментарии/лишние токены).

5. **Обработка команд**

   - + word value:

     - Проверяется наличие ключа через Find.

     - Если уже есть — печатается Exist.

     - Иначе формируется узел с новым случайным приоритетом и выполняется вставка Insert (через Split/Merge). Вывод: OK.

   - - word:

     - Пытается удалить узел Remove. При успехе OK, иначе NoSuchWord.

   - word (поиск):

     - Find возвращает значение — печатается OK: <value>, иначе NoSuchWord.

6. **Treap-операции (суммарно)**

   - Split(root, key) делит дерево на < key и ≥ key.

   - Merge(left, right) сливает два корректных поддерева, поддерживая свойство кучи по приоритету.

- Insert рекурсивно спускается как в BST; если приоритет нового узла выше текущего — делает Split и подставляет новый узел над поддеревьями.

- Remove при совпадении ключа заменяет узел на Merge(left, right).

7. **Вывод результатов**

   - Для каждой корректной команды печатается ровно одна строка результата: OK, Exist, NoSuchWord, либо OK: <число>.

   - Некорректные или неполные команды (например, + word без числа) тихо **игнорируются** (ничего не выводят), при этом следующая строка не теряется.

8. **Завершение и память**

   - При выходе из цикла рекурсивно освобождаются все узлы дерева (без утечек).

   - Процесс завершается с кодом 0.

## Дневник отладки

Первая версия программы не прошла 5-й тест из-за ошибки "Wrong Answer". Симптом: строка + bad без числа приводила к тому, что следующая команда пропускалась. В результате итоговый вывод не совпадал с ожидаемым.

**Причина**: функция SkipSpaces() использовалась внутри ReadWord и ReadUnsignedLongLong. Она пропускала не только пробелы и табы, но и символ перевода строки. Из-за этого при неполной команде + bad парсер «перескакивал» на следующую строку и ломал разбор.

**Исправление 1**: введены функции IsHSpace и SkipHSpaces, которые обрабатывают только горизонтальные пробелы (пробел и таб). Теперь ReadWord и ReadUnsignedLongLong используют именно их, а пропуск перевода строки делается только на верхнем уровне в SkipSpaces или SkipToLineEnd. Это устранило проблему «съедания» команд.

**Исправление 2**: в Insert добавлена явная проверка на существование ключа через Find перед вызовом рекурсивной вставки. Это гарантирует, что в дереве не появятся дубликаты при использовании операции Split.

**Исправление 3**: упрощена и исправлена логика работы со входом в main: лишние пустые строки игнорируются, «мусор» в конце команд обрезается через SkipToLineEnd.

**Результат**: после внесённых изменений программа корректно обрабатывает все диагностические тесты: базовые операции, регистронезависимость, удаление несуществующих слов, ключи длиной 256 символов, граничные значения 0 и $2^{64}-1$, а также некорректные строки ввода. Ошибка с пропуском команд устранена, дубликаты ключей не появляются. Программа прошла все тесты системы.

## Тест производительности

Производительность измерялась для последовательной обработки команд из файла. Для каждого размера входа генерировался набор операций (вставка/поиск/удаление), после чего замерялось «стеночное» время выполнения программы.

Команда запуска:

time ./dict < input.txt > output.txt

Генерация входных данных осуществлялась утилитой gen_dict по формату:

./gen_dict <количество_команд> input.txt

Чтобы исключить влияние локали и округлений при парсинге времени, использовался формат с точкой (LC_ALL=C), а вывод перенаправлялся в /dev/null для минимизации дисковых задержек.

**Конфигурация измерений.** Один прогон на каждый размер входа; реальное время real переведено в миллисекунды.

**Результаты:**

| Количество строк | Время работы (мс) |
|---|---|
| 10 000 | 22 |
| 50 000 | 245 |
| 100 000 | 3693 |

**Анализ:**

Рост времени соответствует оценке **O(n log n)**: при увеличении числа операций в 10 раз время растёт примерно в 8–15 раз, что согласуется с логарифмическим множителем. Такая динамика ожидаема для декартова дерева (Treap), где средняя сложность операций вставки, удаления и поиска — **O(log n)** благодаря случайной балансировке приоритетами.

## Недочёты

1. **Тихий игнор некорректных команд.** Строки вида `+ a` (без числа) просто пропускаются без сообщения — это удобно для тестов, но не информативно для пользователя.

2. **Ограничение алфавита.** Ключи состоят только из латинских букв; другие символы внутри слова не поддержаны.

3. **Жёсткий предел длины.** Ключи обрезаются до 256 символов (лишнее отбрасывается без предупреждения).

## Выводы

- В ходе работы была реализована структура данных **декартово дерево (Treap)**, используемая для построения словаря. Программа поддерживает три основные операции: добавление, удаление и поиск слова, при этом ключи обрабатываются регистронезависимо, а значения хранятся в виде 64-битных целых чисел.
- Программа корректно проходит тесты, в том числе на граничные случаи (ключи длиной 256 символов, максимальные значения $2^{64}-1$, повторные вставки и удаление отсутствующих элементов). Экспериментально подтверждена средняя асимптотика **O(log n)** для каждой операции.
- Таким образом, поставленная задача выполнена: реализован эффективный словарь без использования стандартных контейнеров STL, программа устойчива к некорректному вводу и демонстрирует предсказуемую производительность на больших данных.
- обработка больших массивов данных в реальном времени, где важно линейное время работы;
- подготовка данных для дальнейшей обработки в других структурах (хэш-таблицы, базы данных).

Сложность программирования относительно невысока, но потребовала:

- корректного управления динамической памятью;
- аккуратного вывода индексов с ведущими нулями;
- обеспечения стабильности сортировки.

Таким образом, в работе была реализована сортировка подсчётом для пар «почтовый индекс — число». Алгоритм продемонстрировал линейную временную сложность O(n + k), стабильность сортировки и укладывание в ограничения по времени и памяти. Тестирование подтвердило, что при увеличении объёма данных время работы растёт пропорционально числу элементов, что соответствует теоретическому анализу. Сортировка подсчётом оказалась удобной и эффективной для обработки данных с целочисленными ключами фиксированного диапазона и может

применяться, например, для сортировки идентификаторов, кодов или индексов.