Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: Г. А. Ермеков
Преподаватель: С. А. Михайлова
Группа: М8О-201Б

Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №3

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 $2^{64}-1$. Разным словам может быть поставлен в соответствие один и тот же номер.

Структура данных: B-дерево.

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.

1 Описание

Основная идея алгоритма решения задачи состоит в использовании B-дерева для эффективного хранения и управления словарём, где ключами являются регистронезависимые слова, а значениями — соответствующие номера. B-дерево обеспечивает быстрый поиск, вставку и удаление элементов за счёт сбалансированной структуры и оптимизации операций ввода-вывода, что особенно важно при работе с большими объёмами данных. Для реализации необходимо разработать B-дерево с поддержкой операций добавления, удаления и поиска, а также методов Save и Load для сохранения и загрузки словаря в/из бинарного файла.

2 Исходный код

btree.cpp	
`string to_lower(const string &o)`	Преобразование входной строки к нижнему регистру.
`BNode(bool leaf_)`	Конструктор узла B-дерева, устанавливает флаг листа и нулевое число ключей.
`void BNode::insertNotNull(const string &k, uint64_t v)`	Вставка пары «ключ–значение» в узел (листьевой или внутренний) без проверки корня на переполнение.
`bool BNode::remove(const string &key)`	Удаление ключа из поддерева с последующей балансировкой (слияние, перераспределение).
`pair<bool,uint64_t> BNode::search(const string &k)`	Бинарный поиск ключа в узле и рекурсивный спуск в потомки при необходимости.
`class BTree`	Внешний интерфейс B-дерева: методы `add`, `remove`, `search`, `dump`, `load`.
`int main()`	Цикл чтения команд из `stdin`, разбор и выполнение операций над деревом.

```
const int T = 64;

class BNode {
public:
    bool leaf;
    int c;
    string keys[2 * T - 1];
    uint64_t values[2 * T - 1];
    BNode* children[2 * T];

    BNode(bool leaf_);
    ~BNode();

    pair<bool, uint64_t> search(const string &k);
    void insertNotNull(const string &k, uint64_t v);
    void split(int index);
    bool remove(const string &key);
};

class BTree {
public:
    BNode* root;
```

```
23
24      BTree();
25      ~BTree();
26
27      bool add(const string &word, uint64_t val);
28      bool remove(const string &v);
29      pair<bool, uint64_t> search(const string &word);
30      bool dump(const string &filename, string &errmsg);
31      bool load(const string &fname, string &errmsg);
32  };
33
34  int main();
```

3 Консоль

```
george@GEORGE-PC:/mnt/c/Users/george/Projects/MaiLabs/MAI_labs_Discran/src$
./main
+ a 1
+ A 2
+ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
18446744073709551615
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
A
-A
a
OK
Exist
OK
OK: 18446744073709551615
OK: 1
OK
NoSuchWord
```

Поиск утечек и профилирование

Поиск утечек через `valgrind` не выявил никаких проблем с аллоцированием памяти.

Листинг 1: Вывод valgrind

```
==18481== Memcheck, a memory error detector
==18481== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==18481== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==18481== Command: ./main
==18481==
==18481==
==18481== HEAP SUMMARY:
==18481== in use at exit: 122,880 bytes in 6 blocks
==18481== total heap usage: 313 allocs, 307 frees, 229,822 bytes allocated
==18481==
==18481== LEAK SUMMARY:
==18481== definitely lost: 0 bytes in 0 blocks
==18481== indirectly lost: 0 bytes in 0 blocks
==18481== possibly lost: 0 bytes in 0 blocks
==18481== still reachable: 122,880 bytes in 6 blocks
==18481== suppressed: 0 bytes in 0 blocks
==18481==
==18481== For lists of detected and suppressed errors, rerun with: -s
==18481== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Для анализа производительности использовался инструмент `gprof`. Программа была скомпилирована с флагом `-pg`, и протестирована на массивном наборе входных данных (500.000 команд вставок, удалений и поиска).

Листинг 2: Flat profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  % cumulative self self total
 time seconds seconds calls ns/call ns/call name
 71.43 0.05 0.05 257750 193.99 193.99 BNode::remove(...)
 14.29 0.06 0.01 228750 43.72 43.72 BNode::insertNotNull(...)
 14.29 0.07 0.01 main
  0.00 0.07 0.00 500000 0.00 0.00 to_lower(...)
  0.00 0.07 0.00 500000 0.00 0.00 frame_dummy
  0.00 0.07 0.00 8000 0.00 193.99 BNode::cull(int)
  0.00 0.07 0.00 2250 0.00 0.00 BNode::merge(int)
  0.00 0.07 0.00 2250 0.00 0.00 BNode::split(int)
  0.00 0.07 0.00 250 0.00 0.00 BNode::~BNode()
  0.00 0.07 0.00 1 0.00 0.00 BTree::~BTree()
```

`Flat profile` показал, что:

- Основное время исполнения уходит на метод `BNode::remove` (71.43% от общего времени), что указывает на доминирование операций удаления в тесте.
- Вставка (`BNode::insertNotNull`) занимает 14.29% времени и выполняется эффективно.
- Функция `main` также потребляет 14.29% времени на координацию операций.
- Вспомогательные операции (`to_lower`, `frame_dummy`, структурные операции B-дерева) практически не влияют на производительность (0.00% времени каждая).
- Алгоритмическая сложность операций соответствует заявленной для B-деревьев: эффективное время выполнения при большом количестве операций.

Также `Call Graph` показал, что:

- Удаление вызывает `BNode::cull` для поддержания структурных свойств B-дерева, что ожидаемо для данной реализации.
- Операции `BNode::merge` и `BNode::split` вызываются в равном количестве (2250 раз каждая), что свидетельствует о сбалансированной работе алгоритма реструктуризации.
- Методы очистки (`BNode:: BNode`) и завершения (`BTree:: BTree`) вызываются ограниченное количество раз и не влияют на общую производительность.

Таким образом, можно сделать вывод, что реализация B-дерева демонстрирует ожидаемое поведение и эффективность, а узкие места (удаление и поддержание структуры дерева) соответствуют характеру входных данных и специфике алгоритма B-дерева.

4 Тест производительности

Тест производительности представляет из себя следующее: в B-дерево и в std::map последовательно вставляются 1.000.000 пар «ключ–значение», затем выполняется 100.000 операций поиска случайных ключей и 100.000 операций удаления. Все измерения проводятся в одной программе, выводятся отдельно для каждой фазы.

```
Benchmarking B-Tree implementation:
george@GEORGE-PC:/home/george/Projects/MaiLabs/MAI_labs_Discran/src$ g++ btree.cpp
main.cpp -std=c++17 -O2 -o bench_btree
george@GEORGE-PC:/home/george/Projects/MaiLabs/MAI_labs_Discran/src$ ./bench_btree
B-Tree insertion time: 0.752341 sec
B-Tree search time:    0.048912 sec
B-Tree deletion time:  0.603127 sec

Benchmarking std::map:
george@GEORGE-PC:/home/george/Projects/MaiLabs/MAI_labs_Discran/src$ g++ main_map.cpp
-std=c++17 -O2 -o bench_map
george@GEORGE-PC:/home/george/Projects/MaiLabs/MAI_labs_Discran/src$ ./bench_map
std::map insertion time: 1.127589 sec
std::map search time:    0.081204 sec
std::map deletion time:  0.898432 sec
```

Как видно, B-дерево опережает ‘std::map‘ по времени вставки (0.75с против 1.13с) и удаления (0.60с против 0.90с), а также чуть быстрее при поиске (0.05с против 0.08с), что демонстрирует преимущество сбалансированной структуры при больших объёмах данных.

5 Выводы

Выполнив третью лабораторную работу по курсу «Дискретный анализ», я освоил ключевые аспекты профилирования программ на C++.

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] *B-дерево — Википедия.*
URL: `https://ru.wikipedia.org/wiki/B-дерево` (дата обращения: 20.05.2025).

[3] *Структура данных B-дерево — Хабр.*
URL: `https://habr.com/ru/companies/otus/articles/459216/` (дата обращения: 20.05.2025).