Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Лабораторная работа № 3

«ОБРАБОТКА ДАННЫХ НА ОСНОВЕ МЕТОДА

ЗАКРЫТОГО ХЕШИРОВАНИЯ»

Выполнил: студент гр. 281073

Буйновский М.В.

Проверил: Потоцкий Д.С.

Минск 2023

**Цель работы:** изучить способы ускорения доступа к файлам.

**Задание:** На основе динамических списков необходимо реализовать словарь и хранить его во внешней памяти. Основные операции, выполняемые над данными словаря: поиск, вставка и удаление. В сочетании со списками для построения и обработки словарей использовать хешированные файлы, позволяющее ускорить выполнение операций с данными словаря.

**Код программы:**

**file\_hash\_map.h**

#pragma once

#include <cstddef>

#include <filesystem>

#include <fstream>

#include <iomanip>

#include <iostream>

#include <stdexcept>

#include <string>

#include <vector>

class Color

{

public:

inline static const std::string RED = "\033[31m";

inline static const std::string GREEN = "\033[32m";

inline static const std::string YELLOW = "\033[33m";

inline static const std::string CYAN = "\033[36m";

inline static const std::string RESET = "\033[0m";

};

template <typename T = int>

class HashMap

{

public:

const std::string path = "map";

typedef std::string key\_t;

private:

std::size\_t capacity;

std::string hash(key\_t key)

{

return this->path + "/key-" + std::to\_string(key.size() % this->capacity);

}

void showMap(HashMap::key\_t key, T label)

{

std::cout << " map[" << Color::CYAN << key << Color::RESET << "] - " << Color::YELLOW << label << Color::RESET

<< std::endl;

}

T convertToScalar(const std::string &str)

{

std::istringstream iss(str);

T value;

iss >> value;

return value;

}

public:

HashMap(std::size\_t capacity) : capacity(capacity)

{

std::filesystem::create\_directory(this->path);

}

void show()

{

std::cout << "Hash table: " << std::endl;

for (const auto &entry : std::filesystem::directory\_iterator(this->path))

{

if (entry.is\_regular\_file())

{

std::ifstream file(entry.path());

if (file.good())

{

std::string line;

std::cout << "Reading file: " << entry.path() << std::endl;

while (std::getline(file, line))

{

std::istringstream iss(line);

std::string key, value;

if (std::getline(iss, key, '='))

{

if (std::getline(iss, value))

{

this->showMap(key, this->convertToScalar(value));

}

else

{

std::cout << Color::RED << "Error parsing value in line: " << line << Color::RESET

<< std::endl;

}

}

else

{

std::cout << Color::RED << "Error parsing key-value pair in line: " << line << Color::RESET

<< std::endl;

}

}

std::cout << std::endl;

}

else

{

std::cout << Color::RED << "Error opening file: " << entry.path() << Color::RESET << std::endl;

}

}

}

}

void insert(const std::string &key, const T &value)

{

std::cout << "Inserting '" << Color::YELLOW << value << Color::RESET << "' with key '" << Color::CYAN << key

<< Color::RESET << "':" << std::endl;

std::string hash = this->hash(key);

std::ofstream file(hash, std::ios::app);

if (!file.good())

{

std::cerr << "Error opening file: " << hash << std::endl;

return;

}

file << key << "=" << value << std::endl;

file.close();

}

T find(const std::string &key)

{

std::string hash = this->hash(key);

std::ifstream file(hash);

if (!file.good())

{

std::cout << Color::RED << "Value not found!" << Color::RESET << std::endl;

return T();

}

std::string line;

while (std::getline(file, line))

{

std::istringstream iss(line);

std::string \_key, value;

if (std::getline(iss, \_key, '='))

{

if (std::getline(iss, value))

{

std::cout << Color::GREEN << "Found: " << Color::YELLOW << value << Color::RESET << std::endl;

return this->convertToScalar(value);

}

else

{

std::cout << Color::RED << "Error parsing value in line: " << line << Color::RESET << std::endl;

}

}

else

{

std::cout << Color::RED << "Error parsing key-value pair in line: " << line << Color::RESET

<< std::endl;

}

}

file.close();

std::cout << Color::RED << "Value not found!" << Color::RESET << std::endl;

return T();

}

void remove(const std::string &key)

{

std::string hash = this->hash(key);

std::string tempFile = hash + ".tmp";

std::ifstream inputFile(hash);

std::ofstream outputFile(tempFile);

if (!inputFile)

{

std::cout << Color::RED << "Value not found!" << Color::RESET << std::endl;

return;

}

std::string line;

while (std::getline(inputFile, line))

{

std::istringstream iss(line);

std::string \_key, value;

if (std::getline(iss, \_key, '='))

{

if (key != \_key)

{

outputFile << line << std::endl;

}

}

else

{

std::cout << Color::RED << "Error parsing key-value pair in line: " << line << Color::RESET

<< std::endl;

}

}

inputFile.close();

outputFile.close();

std::remove(hash.c\_str());

std::rename(tempFile.c\_str(), hash.c\_str());

}

};

**main.cpp**

#include "file\_hash\_map.hpp"

#include <chrono>

#include <iomanip>

#include <iostream>

#include <random>

#include <stdlib.h>

int main()

{

HashMap<int> hashMap(10);

while (true)

{

std::cout << "Enter action (" << Color::CYAN << "show" << Color::RESET << ", " << Color::CYAN << "find"

<< Color::RESET << ", " << Color::CYAN << "add" << Color::RESET << ", " << Color::CYAN << "delete"

<< Color::RESET << "): " << Color::YELLOW;

std::string action;

std::cin >> action;

std::cout << Color::RESET;

if (action == "show")

{

hashMap.show();

}

else if (action == "find")

{

std::cout << "Enter key to find: ";

std::string key;

std::cin >> key;

hashMap.find(key);

}

else if (action == "add")

{

std::cout << "Enter key to add: ";

std::string key;

std::cin >> key;

std::cout << "Enter value to add: ";

int value;

std::cin >> value;

hashMap.insert(key, value);

}

else if (action == "delete")

{

std::cout << "Enter key to delete: ";

std::string key;

std::cin >> key;

hashMap.remove(key);

}

else

{

std::cout << "Invalid action!" << std::endl;

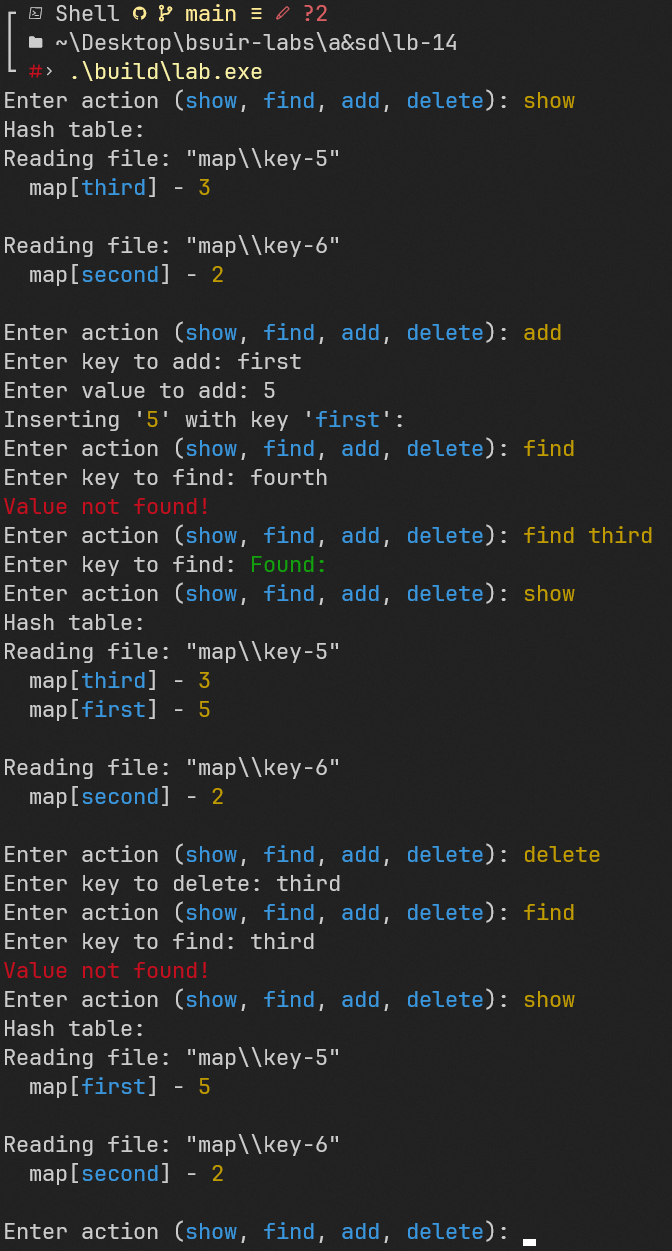
}

}

return EXIT\_SUCCESS;

}

**Скриншот работы программы, демонстрация результатов:**



**Контрольные вопросы:**

**1. В чем заключаются особенности операций с данными, хранящимися во внешней памяти?**

Операции с данными, хранящимися во внешней памяти (например, на жестком диске, сетевом хранилище или облачном хранилище), имеют несколько особенностей по сравнению с операциями над данными, находящимися в оперативной памяти (RAM). Вот некоторые из этих особенностей:

1. Медленный доступ к данным: Внешние носители данных, такие как жесткие диски или удаленные серверы, обычно имеют гораздо более высокую задержку доступа по сравнению с оперативной памятью. Это означает, что время, необходимое для чтения или записи данных из внешней памяти, значительно больше, чем из оперативной памяти.

2. Ограниченная пропускная способность: Внешние устройства имеют ограниченную пропускную способность, что ограничивает скорость передачи данных между внешней памятью и процессором. Это может привести к узким местам в системе при работе с большими объемами данных.

3. Персистентность: Данные, хранящиеся во внешней памяти, обычно сохраняются после выключения компьютера или программы. Это позволяет сохранять данные между сеансами работы и обеспечивает их долгосрочное хранение.

4. Система управления файлами: Для доступа к данным во внешней памяти часто используются операционные системы и системы управления файлами. Это включает в себя работу с файловыми системами, папками, разрешениями на доступ и другими аспектами управления данными.

5. Буферизация: Для улучшения производительности операции с данными могут включать буферизацию. Данные часто читаются или записываются блоками, что позволяет уменьшить количество операций ввода-вывода с внешней памятью.

6. Работа с данными в "партиях": При работе с внешней памятью может потребоваться использовать различные методы оптимизации, такие как сортировка, фильтрация и слияние данных в "партиях" (batch processing) для улучшения эффективности.

7. Асинхронные операции: При работе с удаленными хранилищами или сетевыми ресурсами могут использоваться асинхронные операции для уменьшения задержек в ожидании данных.

Операции с данными во внешней памяти требуют более внимательного управления ресурсами и оптимизации производительности, чем операции с данными в оперативной памяти, из-за различий в скорости доступа и характере хранения данных.

**2. Что общего и различного у хешированных файлов и алгоритма открытого хеширования, предназначенного для работы с данными в оперативной памяти?**

Хешированные файлы и алгоритмы открытого хеширования имеют общие концепции хеширования, но они применяются к разным контекстам и имеют некоторые ключевые различия. Вот общее и различное между ними:

Общие черты:

1. Хеш-функции: Оба метода используют хеш-функции для преобразования данных в фиксированный размер хеш-значения. Это позволяет быстро и эффективно находить данные или проверять целостность данных.

2. Быстрый доступ: Ключевым преимуществом хеш-функций является быстрый доступ к данным. Они позволяют выполнять операции поиска и проверки данных во многие разы быстрее, чем последовательный поиск.

3. Определенный размер: Как и в случае с хеш-функциями, и хешированными файлами, результат хеширования всегда имеет фиксированный размер. Это важно для обеспечения согласованности и быстрого доступа.

Различия:

1. Контекст использования: Хешированные файлы обычно используются для организации данных на диске или во внешней памяти. Они позволяют быстро находить и обновлять данные по ключу. Алгоритмы открытого хеширования предназначены для работы с данными в оперативной памяти, обеспечивая быстрый доступ к данным внутри программы.

2. Хранение данных: Хешированные файлы используют хеш-таблицы для хранения данных на диске. Они могут быть более сложными, чем алгоритмы открытого хеширования, потому что они должны управлять файловой структурой. Алгоритмы открытого хеширования, как правило, используют массивы или списки в оперативной памяти для хранения данных.

3. Обработка коллизий: Хешированные файлы обычно реализуют методы для обработки коллизий, такие как цепочки или открытое линейное адресование, так как данные хранятся на диске и коллизии могут возникать. В алгоритмах открытого хеширования, которые предназначены для оперативной памяти, чаще используется более простое решение для обработки коллизий, такое как перебор с шагом.

4. Сохранение на диске: Хешированные файлы обеспечивают сохранение данных на диске между сеансами работы программы. Алгоритмы открытого хеширования не предполагают сохранение данных на диске и предназначены для работы только в оперативной памяти.

В целом, хешированные файлы и алгоритмы открытого хеширования обладают схожими концепциями, но их применение и реализация различаются в зависимости от контекста использования и целей.

**3. Почему удается ускорить обработку данных при использовании индексированных файлов?**

Использование индексированных файлов позволяет ускорить обработку данных по нескольким причинам:

1. Ускоренный поиск: Индексы представляют собой структуры данных, которые хранят информацию о местоположении данных в файле. Поиск данных в индексированных файлах может быть значительно быстрее, чем в неиндексированных файлах, так как индексы предоставляют информацию о том, где искомые данные находятся в файле. Вместо последовательного просмотра всего файла, при использовании индексов можно прыгать к нужному местоположению данных, что сокращает время доступа.

2. Оптимизированный доступ: Индексы могут быть организованы таким образом, чтобы ускорить доступ к данным. Например, B-деревья и хеш-таблицы используются для организации индексов в базах данных. Они обеспечивают балансировку дерева или равномерное распределение данных, что позволяет быстро находить нужную информацию.

3. Меньшая сложность алгоритмов: Поиск данных в индексированных файлах может осуществляться с использованием более эффективных алгоритмов, чем последовательный поиск. Это уменьшает временную сложность операций поиска и обработки данных.

4. Улучшенная производительность: Благодаря быстрому доступу к данным и оптимизированным алгоритмам обработки, обработка данных в индексированных файлах может выполняться с высокой производительностью, что особенно важно для систем, обрабатывающих большие объемы данных.

5. Возможность параллельной обработки: Индексы могут упростить параллельную обработку данных, так как они позволяют быстро и без конфликтов находить и доступ к различным частям данных. Это особенно полезно в многозадачных и многопоточных средах.

6. Эффективное обновление данных: Хорошо разработанные индексы позволяют эффективно обновлять данные без необходимости пересортировки всего файла.

Индексированные файлы подходят для сценариев, где необходим быстрый и эффективный доступ к данным, особенно в базах данных, поисковых системах и других приложениях, где обработка больших объемов информации играет важную роль.

**Вывод:** в результате выполнения лабораторной работы изучили способ ускорения доступа к файлам.