**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе № 1**

**по дисциплине «Параллельные алгоритмы и системы»**

**Тема: Хэш-таблица с открытой адресацией**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9307 |  | Казачук М.О. |
| Преподаватель |  | Пазников А.А. |

Оглавление

[Цель работы 3](#_Toc131486570)

[Основные положения 3](#_Toc131486571)

[Измерения 5](#_Toc131486572)

[1. Без оптимизаций 6](#_Toc131486573)

[2. Использование битовых операций 6](#_Toc131486574)

[3. Использование C-style строк + локальность 6](#_Toc131486575)

[4. Использование векторизации и развертка цикла в хеш-функции 7](#_Toc131486576)

[5. Предсказание переходов 8](#_Toc131486577)

[6. Использования оптимизаторов компилятора (флаги -O2 -O3) 8](#_Toc131486578)

[7. Нелинейное размещение 9](#_Toc131486579)

[Выводы 10](#_Toc131486580)

[Последняя версия кода 10](#_Toc131486581)

# Цель работы

Разработать хэш-таблицу с открытой адресацией и оптимизировать операции вставки и поиска.

# Основные положения

Было решено реализовать хеш-таблицу для пары строка-строка, так как шаблонные классы и структуры будут работать медленнее и хуже обрабатываются компилятором.

В качестве ячеек таблицы используется вспомогательная структура Node:

class Node {

    friend class HT;

    str key;

    str value;

    Flag flag;

public:

    Node() : flag(Flag::FREE) {}

    Node(str key, str value) : key(key), value(value), flag(Flag::FREE) {}

};

Ключи и значения являются объектами стандартной библиотеки строк.

Флаги нужны для указания состояния ячейки:

enum class Flag : uint8\_t {

    FREE = 0,

    OCCUPIED,

    DELETED

};

Сама таблица задается значениями максимального размера, коэффициента заполняемости и множителя при увеличении.

class HT {

    size\_t size, maxSize, capacity;

    float loadFactor, multiplier;

    Node\* data;

}

Массив узлов создается динамически.

Функция вычисления хеша была выбрана:

inline size\_t getHash(str s) {

    size\_t h = 0;

    for (char\* c = s.data(); \*c; c++) {

        h = (h << 5) - h + (size\_t) \*c;

    }

    return h;

}

Так как хеш-таблица с открытой адресацией, при коллизии ищем свободную ячейку следующей за занятой.

void put(str key, str value) {

        size\_t hash = getHash(key) % capacity;

        size\_t hashStep = 1;

        while(data[hash].flag == Flag::OCCUPIED) {

            if (strEquals(data[hash].key, key)) {

                return;

            }

            hash = (hash + hashStep) % capacity;

        }

        data[hash].key = key;

        data[hash].value = value;

        data[hash].flag = Flag::OCCUPIED;

        size++;

        if (size > maxSize) {

            increaseData();

        }

    }

Таблица расширяется при заполнении больше определенного коэффициента. Сам коэффициент был выбран в 0.75.

Для увеличения размера необходимо выделить новый массив и перенести в него старые элементы.

void increaseData() {

        Node\* oldData = data;

        size\_t oldCapacity = capacity;

        data = initNodes(capacity \* multiplier);

        capacity = capacity \* multiplier;

        size = 0;

        maxSize = capacity \* loadFactor;

        Node curNode;

        for (size\_t i = 0; i < oldCapacity; i++) {

            curNode = oldData[i];

            if (curNode.flag == Flag::OCCUPIED) {

                put(curNode.key, curNode.value);

            }

        }

        delete []oldData;

    }

Получение и удаления выполняются аналогично поиском подряд идущих элементов, пока не найдем элемент с таким же ключом.

bool get(str key, str &outValue) {

        size\_t hash = getHash(key) % capacity;

        bool fl = true, found = false;

        Node \*curNode;

        while(fl) {

            curNode = data+hash;

            if (curNode->flag == Flag::FREE) {

                fl = false;

            }

            if (curNode->flag == Flag::OCCUPIED) {

                if (strEquals(curNode->key, key)) {

                    outValue = curNode->value;

                    found = true;

                    fl = false;

                }

            }

            hash = (hash + 1) % capacity;

        }

        return found;

    }

# Измерения

Идея проверки заключается в том, что данные из файла 1 добавляются в таблицу 1, после чего в файле 2 ищутся слова, которые содержаться в таблице 1 и считается общее количество общих слов для проверки корректности выполнения.

Таким образом, вычисляется время для добавления N элементов в таблицу и M получений значений из нее. Где N и M – размеры файлов 1 и 2. Было выбрано N = M = 100 000.

Цикл выполняется по 100 раз и вычисляется среднее время выполнения.

# Без оптимизаций

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.031252 | 1 | 1 |

# Использование битовых операций

Больше всего времени уходит на вычисление хеша. Очевидным вариантов его ускорить – заменить взятие остатка на битовое И.

Таким образом, добавляется поле capacity\_minus\_one, с которым складывается вычисленное значение хеш-функции:

size\_t hash = getHash(key) & cap\_minus\_one;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.032252 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.188 | 1.03 |

Видно, что это больше повлияло на добавления элементов в таблицу, так как при том же количестве операций, необходимо еще и пересчитывать хеши при расширении. В реальной работе операции получения значения используются куда чаще, что покажет даже больший выигрыш в сравнении с первым вариантом.

# Использование C-style строк + локальность

Использование строк С++ несет с собой значительные накладные расходы. Заменим строки на массив символов и вставим по необходимости более оптимизированные функции работы с ними из Си.

class Node {

    friend class HT;

    char[] key;

    char[] value;

    Flag flag;

}

Также введем максимальный размер строк в 32 символа, что позволит расположить данные непосредственно в самой ячейке близко друг к другу.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.032252 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.19 | 1.03 |
| 3. | 0.0149098 | 0.0144078 | 3.22 | 2.23 |

Также значительно сократилось количество кэш промахов.

# Использование векторизации и развертка цикла в хеш-функции

Так как у нас фиксированный размер строки, то можно было бы без особых потерь использовать векторизацию для вычисления хеш-функции.

inline size\_t getHash(str data) {

    size\_t crc = 0;

    \_\_m256i block = \_mm256\_loadu\_si256((\_\_m256i\*)data);

    crc = \_mm\_crc32\_u64(crc, \_mm256\_extract\_epi64(block, 0));

    crc = \_mm\_crc32\_u64(crc, \_mm256\_extract\_epi64(block, 1));

    crc = \_mm\_crc32\_u64(crc, \_mm256\_extract\_epi64(block, 2));

    crc = \_mm\_crc32\_u64(crc, \_mm256\_extract\_epi64(block, 3));

    return crc;

}

Также для сравнения строк их тоже можно применить

inline int memcmp32\_avx2(const char\* s1, const char\* s2) {

    \_\_m256i a = \_mm256\_loadu\_si256((\_\_m256i\*)(s1));

    \_\_m256i b = \_mm256\_loadu\_si256((\_\_m256i\*)(s2));

    \_\_m256i cmp = \_mm256\_cmpeq\_epi8(a, b);

    int mask = \_mm256\_movemask\_epi8(cmp);

    if (mask != 0xffffffff) {

        return 1;

    }

    return 0;

}

К сожалению, результаты оказались даже хуже предыдущих:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.0322520 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.19 | 1.03 |
| 3. | 0.0149098 | 0.0144078 | 3.22 | 2.23 |
| 4 | 0.0793697 | 0.0355465 | 0.6 | 0.9 |

Скорее всего, это где-то возникла ошибка при их использовании, но еще подобные результаты могут быть оправданы тем, что некоторые оптимизации компилятор выполняет по умолчанию, а сравнение строк, например, в большинстве случаев останавливается на сравнении первого символа, что выходит даже быстрее, чем векторизация.

# Предсказание переходов

При перезаписи массива выполняется достаточно большое число проверок каждый раз. Также при сравнении тоже можно предположить, какая ветка более вероятна.

 void increaseData() {

...

        for (size\_t i = 0; i < oldCapacity; i++) {

            curNode = oldData+i;

            if (curNode->flag == Flag::OCCUPIED) [[unlikely]] {

                put(curNode->key, curNode->value);

            }

        }

...

Здесь результат не такой очевидный, но на большем числе запусков можно отметить небольшой прирост.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.0322520 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.19 | 1.03 |
| 3. | 0.0149098 | 0.0144078 | 3.22 | 2.23 |
| 4. | 0.0793697 | 0.0355465 | 0.6 | 0.9 |
| 5. | 0.0145463 | 0.0143450 | 3.3 | 2.24 |

# Использования оптимизаторов компилятора (флаги -O2 -O3)

В MinGW встроены некоторые оптимизаторы, которые можно подключить с помощью вышеприведенных флагов. Флаг -О3 включает О2 и некоторые дополнительные, которые не всегда ведут к увеличению производительности.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.0322520 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.19 | 1.03 |
| 3. | 0.0149098 | 0.0144078 | 3.22 | 2.23 |
| 4. | 0.0793697 | 0.0355465 | 0.6 | 0.9 |
| 5. | 0.0145463 | 0.0143450 | 3.3 | 2.24 |
| 6.1 | 0.0107172 | 0.0095096 | 4.49 | 3.39 |
| 6.2 | 0.0107040 | 0.00941704 | 4.496 | 3.42 |

# Нелинейное размещение

Одним из способов модернизации хеш-таблиц является нелинейное размещение элементов при коллизии. Очевидно, что в худшем случае придется перебрать почти всю таблицу для поиска подходящего места, что часто неэффективно. Предлагается искать пустое место с увеличивающимся шагом на каждом элементе.

void put(str key, str value) {

        size\_t hash = getHash(key)& cap\_minus\_one;

        size\_t hashStep = 1;

        while(data[hash].flag == Flag::OCCUPIED) {

            if (strEquals(data[hash].key, key)) [[unlikely]]{

                return;

            }

            hash = (hash + hashStep++) & cap\_minus\_one;

        }

...

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.0322520 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.19 | 1.03 |
| 3. | 0.0149098 | 0.0144078 | 3.22 | 2.23 |
| 4. | 0.0793697 | 0.0355465 | 0.6 | 0.9 |
| 5. | 0.0145463 | 0.0143450 | 3.3 | 2.24 |
| 6.1 | 0.0107172 | 0.0095096 | 4.49 | 3.39 |
| 6.2 | 0.0107040 | 0.00941704 | 4.496 | 3.42 |
| 7. | 0.0106724 | 0.0094099 | 4.509 | 3.427 |

Стоит отметить, что результаты тоже увеличились едва заметно.

# Выводы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Время на инициализацию | Время на поиск | Ускорение (отн. 1) | |
| 1. | 0.0481268 | 0.0322520 | 1 | 1 |
| 2. | 0.0405035 | 0.0312857 | 1.19 | 1.03 |
| 3. | 0.0149098 | 0.0144078 | 3.22 | 2.23 |
| 4. | 0.0793697 | 0.0355465 | 0.6 | 0.9 |
| 5. | 0.0145463 | 0.0143450 | 3.3 | 2.24 |
| 6.1 | 0.0107172 | 0.0095096 | 4.49 | 3.39 |
| 6.2 | 0.0107040 | 0.00941704 | 4.496 | 3.42 |
| 7. | 0.0106724 | 0.0094099 | 4.509 | 3.427 |

1 – начальный алгоритм

2 – битовые операции вместо арифметических

3 – Использование более эффективных типов данных + улучшенная локальность данных за счет размещения в соседних участках памяти

4 – Векторизация

5 – Предсказание переходов

6 – Оптимизация с помощью флагов компилятора

7 – Нелинейное размещение элементов при коллизиях

В среднем удалось повысить эффективность в 4 раза.

# Последняя версия кода

ht7.h

#include <iostream>

#include <cstring>

static const float LOAD\_FACTOR = 0.75f;

static const float MULTIPLIER  = 2;

static const size\_t INIT\_SIZE  = 64;

static const int BUF\_SIZE  = 32;

using str = char[BUF\_SIZE];

enum class Flag : uint8\_t {

    FREE = 0,

    OCCUPIED,

    DELETED

};

class Node {

    friend class HT;

    str key;

    str value;

    Flag flag;

public:

    Node() : flag(Flag::FREE) {

    }

    Node(str key, str value) : flag(Flag::FREE) {

        memcpy(this->key, key, BUF\_SIZE);

        memcpy(this->value, value, BUF\_SIZE);

    }

};

inline Node\* initNodes(size\_t capacity) {

    return new Node[capacity];

}

inline bool strEquals(str s1, str s2) {

    return strcmp(s1, s2) == 0;

}

inline size\_t getHash(str s) {

    size\_t h = 0;

    for (char\* c = s; \*c; c++) {

        h = (h << 5) - h + (size\_t) \*(c);

    }

    return h ;

}

class HT {

    size\_t size, maxSize, capacity, cap\_minus\_one;

    float loadFactor, multiplier;

    Node\* data;

    void increaseData() {

        Node\* oldData = data;

        size\_t oldCapacity = capacity;

        data = initNodes(capacity \* multiplier);

        capacity = capacity \* multiplier;

        cap\_minus\_one = capacity - 1;

        size = 0;

        maxSize = capacity \* loadFactor;

        Node\* curNode;

        for (size\_t i = 0; i < oldCapacity; i++) {

            curNode = oldData+i;

            if (curNode->flag == Flag::OCCUPIED) [[unlikely]] {

                put(curNode->key, curNode->value);

            }

        }

        delete []oldData;

    }

public:

    HT(size\_t capacity=INIT\_SIZE, float multiplier=MULTIPLIER, float loadFactor=LOAD\_FACTOR) {

        this->multiplier = multiplier;

        this->loadFactor = loadFactor;

        this->capacity = capacity;

        cap\_minus\_one = capacity - 1;

        maxSize = capacity \* loadFactor;

        data = initNodes(capacity);

    }

    ~HT() {

        delete [] data;

    }

    void put(str key, str value) {

        size\_t hash = getHash(key)& cap\_minus\_one;

        size\_t hashStep = 1;

        while(data[hash].flag == Flag::OCCUPIED) {

            if (strEquals(data[hash].key, key)) [[unlikely]]{

                return;

            }

            hash = (hash + hashStep++) & cap\_minus\_one;

        }

        memcpy(data[hash].key, key, BUF\_SIZE);

        memcpy(data[hash].value, value, BUF\_SIZE);

        data[hash].flag = Flag::OCCUPIED;

        size++;

        if (size > maxSize) {

            increaseData();

        }

    }

    bool get(str key, str &outValue) {

        size\_t hash = getHash(key) & cap\_minus\_one;

        size\_t hashStep = 1;

        bool fl = true, found = false;

        Node \*curNode;

        while(fl) {

            curNode = data+hash;

            if (curNode->flag == Flag::FREE) {

                fl = false;

            }

            if (curNode->flag == Flag::OCCUPIED) {

                if (strEquals(curNode->key, key)) {

                    memcpy(outValue, curNode->value, BUF\_SIZE);

                    found = true;

                    fl = false;

                }

            }

            hash = (hash + hashStep++) & cap\_minus\_one;

        }

        return found;

    }

    bool remove(str key) {

        size\_t hash = getHash(key) & cap\_minus\_one;

        size\_t hashStep = 1;

        bool fl = true, found = false;

        Node \*curNode;

        while(fl) {

            curNode = data+hash;

            if (curNode->flag == Flag::FREE) {

                fl = false;

            }

            if (curNode->flag == Flag::OCCUPIED) {

                if (strEquals(curNode->key, key)) {

                    curNode->flag == Flag::DELETED;

                    found = true;

                    fl = false;

                    size--;

                }

            }

            hash = (hash + hashStep++) & cap\_minus\_one;

        }

        return found;

    }

    size\_t getSize() {

        return size;

    }

};

main.cpp

#include <iostream>

#include <memory>

#include <stdexcept>

#include "bench.h"

#include "ht7.h"

#include "text.h"

#define REPEATS 100

int main() {

    float time;

    bench\_start();

    CharLines lines1("data1.txt", 100000);

    time = bench\_measure();

    std::cout << "Load data1: " << lines1.size() << " items for " << time << " s" << std::endl;

    bench\_start();

    CharLines lines2("data2.txt", 100000);

    time = bench\_measure();

    std::cout << "Load data2: " << lines2.size() << " items for " << time << " s" << std::endl;

    HT ht1;

    bench\_start();

    for (auto i : lines1) {

        ht1.put(i, "");

    }

    time = bench\_measure();

    std::cout << "Init HT1: " << time << " s" << std::endl;

    float t = 0;

    int count;

    for (int r = 0; r < REPEATS; r++) {

        HT ht2;

        count = 0;

        str s;

        bench\_start();

        for (auto i : lines2) {

            if (ht1.get(i, s)) {

                count++;

            }

        }

        time = bench\_measure();

        t += time;

    }

    std::cout << "\nCycles: " << REPEATS << " for " << t << " s" << std::endl;

    std::cout << "Avg time: " << t / REPEATS << " s" << std::endl;

    std::cout << "Num of ops: \n\tput = repeats = " << count << std::endl;

    std::cout << "\tget = data2.size = " << lines2.size() << std::endl;

    return 0;

}