Hashmap Course

Cliquet Florian
note taking of online ressources

10 septembre 2024

Plagiarism Mention

We attest that the content of this document is original and stems from our personal reflections.

Sommaire

Introduction

This document provides an overview of HashMap concepts, written by Cliquet Florian.

HashMap is a data structure that maps keys to values, where each key must be unique. This document explains various hashing techniques for efficient key look-up, including FNV-1a and MurmurHash3 algorithms.

1 Main Logic

A HashMap, also known as a hash table, is a data structure that stores key-value pairs. It allows for fast retrieval, insertion, and deletion of data based on a unique key. The primary goal of a HashMap is to map keys to values in a way that minimizes lookup time, typically aiming for constant time complexity $O(1)$ in average cases.

1.1 How a HashMap Works

The working principle of a HashMap revolves around the concept of hashing. Here's a breakdown of how it operates :

— **Hashing :** When a key is provided, a HashMap uses a hash function to convert the key into an integer, known as a hash code. This hash code is then mapped to an index in an internal array (also known as the bucket array).

— **Array of Buckets :** The internal structure of the HashMap consists of an array where each slot (bucket) holds a linked list or another data structure to store key-value pairs. If two different keys generate the same hash code (a collision), the HashMap stores both key-value pairs in the same bucket, typically using a linked list.

— **Collisions :** A collision occurs when multiple keys hash to the same index. HashMaps handle collisions using techniques such as *separate chaining* (linked lists or trees in each bucket) or *open addressing* (probing for the next empty spot in the array).

— **Insertion :** To insert a key-value pair, the HashMap computes the hash code of the key and maps it to an index in the array. If the index is already occupied due to a collision, the new entry is appended to the list or tree at that index.

— **Lookup :** To retrieve a value, the HashMap computes the hash code of the key and accesses the corresponding index in the bucket array. If there are multiple entries in the bucket, it scans through them (typically a short list) to find the correct key.

1.2 Advantages of HashMap

HashMaps are widely used in various applications because of their benefits :

— **Fast Access :** In an average scenario, a HashMap provides $O(1)$ time complexity for insertion, deletion, and lookup operations, making it highly efficient for large datasets.

— **Efficient Memory Use :** HashMaps provide a flexible structure that can adapt based on the number of elements. For sparse datasets, the memory usage is minimized because only non-empty buckets consume memory.

— **No Strict Order :** Unlike trees or arrays, a HashMap does not enforce any ordering among the elements, which can lead to performance improvements

when ordering is unnecessary.
- **Dynamic Resizing :** Many HashMap implementations automatically resize (increase or decrease) the internal bucket array when the load factor exceeds a certain threshold, ensuring consistent performance even as the dataset grows.

1.3 Disadvantages of HashMap

While HashMaps offer significant advantages, they also have some drawbacks :
- **Hash Collisions :** In the worst case, a large number of collisions can degrade the performance to $O(n)$, as each bucket may need to be searched sequentially. Effective hash functions and good hash table sizing can mitigate this problem.
- **Memory Overhead :** HashMaps use an internal array whose size is typically larger than the number of elements to minimize collisions. This can lead to memory overhead, especially if the load factor (number of entries divided by the number of buckets) is kept low.
- **No Key Order :** A HashMap does not guarantee the order of elements. If order is important, data structures like `TreeMap` or `LinkedHashMap` (in other languages like Java) may be more appropriate.
- **Complex Hashing Functions :** The efficiency of a HashMap heavily depends on the quality of the hash function. Poorly designed hash functions can lead to uneven distribution of keys across buckets, causing excessive collisions.

1.4 Trade-offs in HashMap Design

When designing or using a HashMap, it's important to consider the following trade-offs :
- **Speed vs. Memory :** HashMaps are typically fast but may require more memory due to unused buckets. Balancing speed with memory efficiency depends on the use case, and adjustments to the load factor and initial size can optimize performance.
- **Choice of Hash Function :** The choice of hash function greatly affects the distribution of keys. A good hash function minimizes collisions, but complex hash functions may add computational overhead.
- **Handling Collisions :** The two primary methods for handling collisions—separate chaining and open addressing—have different trade-offs. Separate chaining allows for faster recovery from collisions but increases memory use, while open addressing can save space but may degrade performance when the table becomes full.

In conclusion, HashMaps are powerful tools for efficient data storage and retrieval. Understanding their strengths and weaknesses allows developers to choose the right approach for a given problem, optimizing both performance and resource usage.

2 Implementation using FNV-1a Algorithm

The FNV-1a (Fowler-Noll-Vo) hash function is a non-cryptographic algorithm known for its simplicity and speed, making it ideal for applications like hash maps. It operates by iterating over the bytes of the key and applying bitwise operations to generate a unique hash value.

Below is an enhanced implementation of a HashMap using the FNV-1a algorithm :

```c
// Define the size of the HashMap (number of buckets)
#define HT_SIZE 1024 // You can adjust this as needed

// Hash entry structure: each node holds a key-value pair
typedef struct hash_entry {
    char *key;           // Key (string)
    void *value;         // Value (pointer to any data type)
    struct hash_entry *next; // Pointer to the next entry in case of collisions (chaining)
} hash_entry_t;

// HashMap structure (array of hash_entry_t pointers)
typedef struct {
    hash_entry_t **buckets;  // Array of buckets (each bucket is a linked list of hash entries)
    int size;                // Number of buckets
} hashtable_t;

/**
 * Initialize the HashMap.
 * Allocates memory for the buckets and sets each to NULL (empty).
 */
void ht_hash_init(hashtable_t *ht) {
    ht->size = HT_SIZE;
    ht->buckets = calloc(ht->size, sizeof(hash_entry_t *)); // Allocate memory for the bucket array
    if (!ht->buckets) {
        perror("Failed to initialize hash table");
        exit(EXIT_FAILURE); // Handle memory allocation failure
    }
}

/**
 * FNV-1a hash function.
 * Converts a string (key) into an unsigned integer.
 *
 * @param key: The string key to be hashed.
 * @return: The generated hash value (index in the bucket array).
 */
unsigned int ht_hash(const char *key) {
    if (key == NULL) {
        return 0;  // Return 0 for NULL keys (or handle accordingly)
    }

    unsigned int hash = 2166136261U; // FNV-1a initial hash value (32-bit prime)

    // Loop through each character of the key and update the hash
    for (const char *p = key; *p; p++) {
        hash ^= (unsigned char)(*p); // XOR hash with the current byte (character)
        hash *= 16777619U;            // Multiply by FNV prime
    }

    return hash % HT_SIZE;  // Return the hash value modulo the table size (to get an index)
}

/**
 * Insert a key-value pair into the HashMap.
 * Uses separate chaining to handle collisions.
 *
 * @param ht: The hash table where the entry should be added.
 * @param key: The key string (must be unique).
 * @param value: The value to be associated with the key.
 */
void ht_insert(hashtable_t *ht, const char *key, void *value) {
    unsigned int index = ht_hash(key);  // Get the index from the hash function

    // Create a new hash entry
    hash_entry_t *new_entry = malloc(sizeof(hash_entry_t));
    if (!new_entry) {
        perror("Failed to allocate memory for new hash entry");
        exit(EXIT_FAILURE);
    }
    new_entry->key = strdup(key);  // Copy the key to ensure
       memory safety
    new_entry->value = value;
    new_entry->next = NULL;

    // Insert the new entry into the appropriate bucket
    if (ht->buckets[index] == NULL) {
        ht->buckets[index] = new_entry;  // No collision,
           insert directly
    } else {
        // Collision handling via separate chaining (linked
           list)
        hash_entry_t *current = ht->buckets[index];
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_entry;  // Append to the end of
           the list
    }
}

/**
 * Retrieve a value from the HashMap based on the key.
 *
 * @param ht: The hash table to search.
 * @param key: The key to lookup.
 * @return: The value associated with the key, or NULL if the
    key does not exist.
 */
void *ht_get(hashtable_t *ht, const char *key) {
    unsigned int index = ht_hash(key);  // Get the index from
       the hash function

    // Search for the key in the bucket (linked list)
    hash_entry_t *current = ht->buckets[index];
    while (current != NULL) {
        if (strcmp(current->key, key) == 0) {
            return current->value;  // Key found, return the
               associated value
        }
        current = current->next;
    }

```

```c
    return NULL;  // Key not found
}

/**
 * Free the memory used by the HashMap.
 * Ensures that all allocated memory is properly released.
 *
 * @param ht: The hash table to be freed.
 */
void ht_free(hashtable_t *ht) {
    for (int i = 0; i < ht->size; i++) {
        hash_entry_t *current = ht->buckets[i];
        while (current != NULL) {
            hash_entry_t *temp = current;
            current = current->next;
            free(temp->key);  // Free the duplicated key
            free(temp);       // Free the entry
        }
    }
    free(ht->buckets);  // Free the bucket array
}
```

2.1 Explanation of the FNV hash algorithm

There is two variants :

- FNV-1 : first multiplty(with a prime number) then XOR (operation with the current byte)
- FNV-1a ; first XOR then multiplty (slighty better)

2.1.1 The FNV-1a Hash Seed (offset Basis)

The seed or offset basis is the initial value of the hash. The hash process beings by initializing the hash with a large constant named the **offset basis**, chose to reduce the likelihood of collisions.

Offset basis for different hash size :

- **32-bit FNV-1a :**
  - 216613621 (decmial)
  - 0x811c9dc5 (hexadecimal)
- **64-bit FNV-1a :**
  - 14695981039346656037 (decimal)
  - 0xcbf29ce484222326 (hexa)

2.1.2 FNV-1a Prime Number

In addition to the seed, the **prime number** used in the FNV,it helps ensure that enven small changes in input will produce significantly different hash values, thus reducing collisions.
The FNV-1a prime for 32-bit hashes is **16777619**.
The prime is used to mix the bits of the hash value and spread it evenly across the possible range of outputs.

2.1.3 Summary FNV-1a 32-bits

— Start with the seed : Begin the hash value with the **offset basis (216613621)**
— For each byte in the input (key)
  — XOR the hash value value with the byte
  — Multiply the resulting value by the **FNV prime (16777619)**
— Return the final hash value modulo HT_SIZE ;

2.2 Explanation of the Implementation

This implementation addresses several important aspects of the HashMap design :

— **Collision Handling :** Separate chaining is used to handle collisions. When multiple keys produce the same hash value (index), they are stored in a linked list at the same index. New entries are appended to the list, and key lookups traverse the list to find the desired key.
— **Memory Safety :** The ‘strdup‘ function is used to copy the key string, ensuring that modifying or freeing the original key does not affect the hash table. Memory allocation is checked for errors, and proper error handling is performed when allocation fails.
— **Dynamic Memory Management :** The ‘ht_free‘ function ensures that all dynamically allocated memory is properly released, preventing memory leaks. This includes freeing the keys, values (if necessary), and the linked list nodes.

2.3 Pros and Cons of FNV-1a

Pros :

— **Simplicity :** FNV-1a is a very simple and fast hash function, making it suitable for applications where speed is crucial.
— **Good Distribution :** FNV-1a provides good hash distribution for short keys, reducing the likelihood of collisions.
— **Low Memory Overhead :** The function uses a small, fixed amount of memory, making it efficient for use in environments with limited resources.

Cons :

- **Non-Cryptographic :** FNV-1a is not suitable for cryptographic purposes. It does not provide the level of security required for cryptographic applications.
- **Vulnerable to Poor Key Distribution :** In some cases, FNV-1a may not perform well with certain patterns of input data, especially if the keys have low entropy.

3 Conclusion

Thanks for reading the entire pdf, i hope it's usefull to you. If you need any help contact me on Github