Hash Table

quarta-feira, 16 de outubro de 2024 13:49

→ Used for fast data retrieval
↳ Efficient in lookups / insertions / deletions O(1)
↳ Depends on the collisions

Hash table ≈ hash map → maps keys to values

↳ Achieve fast lookup by using hash functions to compute an index / hash for a key
↳ Constant time complexity O(1)

2. Key Concepts

- **Hash Function**: A function that takes an input (key) and returns an integer (hash code). This hash code is typically converted into an array index where the value is stored.
- **Collision**: Two different keys may generate the same hash code, leading to a conflict. Hash tables must handle collisions to maintain efficiency.
- **Buckets**: The array used by the hash table to store key-value pairs. Each bucket can hold multiple items in case of collisions.

B1 V1 V2
B2 V1
B3 V1 V2
B4 V1 V2

3. Collision Handling Techniques

①

- **Separate Chaining**: Each bucket is a linked list (or another data structure). When collisions occur, multiple elements are stored in the same bucket by chaining them together.
- **Open Addressing**: Instead of using linked lists, open addressing probes (searches) for the next available slot when a collision occurs.
  - **Linear Probing**: If a collision occurs, the algorithm checks the next slot (linearly) until an empty slot is found.
  - **Quadratic Probing**: Instead of linearly searching for the next slot, this method uses a quadratic function to find the next available position.
  - **Double Hashing**: Uses a second hash function to determine the step size for probing.

→ Values are not stored in a sorted order
→ ① Implemented with an array of linked lists
↳ Map the key to a index in the array using hash function
↳ Store the value in the linked list at the position

- Map the key to a index in the array using hash function
  - Then, add the value to the linked list at the position
- Use good hash functions to ensure that the keys are well distributed and reduce the number of collisions
  - We'll have collisions!!!
- Large amount of data
- thread safe
- Not sorted
- Implemented as unordered_map<>
  - Cache information for fast retrieval

Load Factor (LF)

Ratio → $LF = \frac{\text{N° of elements in the hash table}}{\text{N° of available buckets}}$

High LF → More likely to have collisions
- When cross a treshold → Resize the hash table

6. Time Complexity

- **Insert**: O(1) on average, O(n) in the worst case (when many collisions occur).
- **Lookup**: O(1) on average, O(n) in the worst case.
- **Delete**: O(1) on average, O(n) in the worst case.

5. Hash Table Operations in C++

In C++, hash tables can be implemented using `std::unordered_map`, which provides the basic functionality of a hash table with separate chaining for collision resolution.

Example:

cpp

Copy code

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> hashTable;

    // Insert key-value pairs
    hashTable["apple"] = 1;
    hashTable["banana"] = 2;
    hashTable["orange"] = 3;

    // Access elements
    std::cout << "apple: " << hashTable["apple"] << std::endl;

    // Check if a key exists
    if (hashTable.find("banana") != hashTable.end()) {
        std::cout << "banana exists" << std::endl;
    }

    // Delete a key
    hashTable.erase("orange");

    return 0;
}
```

→ Way to access the value using the key

key

→ Super important before any access point

↳ Just need to pass the key

→ Difference between map() ≈ unordered_map()

Map → Keys in incremental order → search / Insert → O(log N)

↳ Uses a BST → Self Balanced Binary search

↳ Used when order is important

unordered_map() → Implemented using hash table (no order)

↳ search / Insertion O(1)

↳ Used for when data that order is not important

→ Used to store data that ordering is not important

→ Implement not using unordered_map

→ Class definition ⟶ Insert, Remove, Get

- How to define a hash functions?
  - ↳ Produce a fixed size output used to index a data structure
  - ↳ Deterministic → same input same output
  - ↳ Efficiently ⟶ fast to calculate
  - ↳ Uniformly ⟶ Distribute the values evenly

↳ Define this hash functions as a private method

↳ Don't want to expose it as public