Hashing Techniques

This C++ file is a powerful and practical demonstration of **Hashing**, a fundamental technique for enabling extremely fast data storage and retrieval (averaging $O(1)$ time). The core idea of hashing is to use a **hash function** to map a key (like a phone number or an ID) to a specific index in an array, called a hash table. 🗺️

However, the main challenge in hashing is the **collision**—what happens when two different keys map to the same index? This file is a masterclass in solving this problem. It showcases and compares the two primary collision resolution strategies:

1. **Separate Chaining (Closed Addressing):** This method handles collisions by creating a linked list at the collision index. All keys that hash to the same spot are simply stored in that list.
2. **Open Addressing:** This strategy handles collisions by finding the next available empty slot in the hash table itself. The file demonstrates three popular "probing" methods for finding that next slot: **Linear Probing**, **Quadratic Probing**, and **Double Hashing**.

By providing clear, side-by-side implementations, this file serves as an excellent guide to understanding the trade-offs between these crucial hashing techniques.

**Chaining Method**

This method, also known as **Separate Chaining**, is an elegant way to resolve collisions. Instead of searching for another slot, it treats each index in the hash table as the head of a **linked list**. When a collision occurs, the new key is simply inserted into the linked list at that index.

* struct Node \*HT[10];

This line declares the hash table. It's an array of pointers to Node. Each of these 10 pointers will serve as the head of a linked list. Initially, they are all set to NULL.

* int hashFunction(int key){ return key % 10; }

This is the simple modulo hash function. It takes a key and computes the index it should map to. For example, a key of 22 results in 22 % 10 = 2, so it belongs at index 2.

* void insert(struct Node \*H[], int key){ int index = hashFunction(key); SortedInsert(&H[index], key); }

This is the core insertion logic. It calculates the index for the key. Then, it calls SortedInsert to add the key into the linked list at H[index]. For example, 12, 22, and 2 all hash to index 2, so they will all be placed in the linked list that starts at HT[2].

* temp = LinearSearch(HT[hashFunction(key)], key);

To search for a key, the algorithm first computes its hash index to find the correct linked list. Then, it performs a standard linear search on that specific list to find the element.

**Linear Probing Method**

This is a type of **Open Addressing** where the hash table itself stores all the data in a single array. When a collision occurs (the target slot is already occupied), the algorithm resolves it by **linearly probing** for the next available empty slot.

* int probe(int HT[], int key){ ... while(HT[(index + i) % SIZE] != 0){ i++; } ... }

This is the "probing" function. It starts at the initial hash index. If that slot is not empty (!= 0), it increments i and checks the next slot (index + 1), then the next (index + 2), and so on. The modulo % SIZE ensures that the search wraps around to the beginning of the array if it reaches the end.

* insert(HT, 26);

Let's trace an insertion. 12 hashes to index 2. 25 hashes to 5. 35 also hashes to 5 (collision!). The probe function checks index 6, finds it empty, and places 35 there. Now, 26 hashes to 6 (another collision!). The probe function checks index 7, finds it empty, and places 26 there. This tendency for collisions to form clusters is known as primary clustering.

* int search(int HT[], int key){ ... }

The search function must follow the exact same probing sequence. To find 35, it first checks its hash index (5). It finds 25 there, which is not a match. So, it probes to the next slot (6) and finds 35. If it had encountered an empty slot during this process, it would conclude that the key is not in the table.

**Quadratic Probing**

This is another Open Addressing technique designed to improve upon Linear Probing. To resolve a collision, it probes for the next slot using a quadratic step size ($i^2$) instead of a linear one ($i$). This helps to break up the primary clustering that affects linear probing.

* int probe(int HT[], int key){ ... while(HT[(index + i\*i) % SIZE] != 0){ i++; } ... }

This is the key difference from linear probing. The next slot to check is determined by the formula (index + i\*i). So, if the initial index is occupied, it will check index + 1, then index + 4, then index + 9, and so on.

* insert(HT, 13);

Let's trace an example. 23 hashes to index 3. 43 also hashes to 3 (collision). The first probe (i=1) checks (3 + 1\*1) % 10 = 4. Slot 4 is empty, so 43 is placed there. Now, 13 also hashes to 3 (collision).

* 1. Probe 1 (i=1): (3 + 1\*1) % 10 = 4. This slot is occupied by 43.
  2. Probe 2 (i=2): (3 + 2\*2) % 10 = 7. This slot is empty, so 13 is placed at index 7.

This shows how the quadratic probes "jump" over occupied slots, spreading out the colliding elements more effectively.

**Double Hashing Method**

Double Hashing is one of the most effective Open Addressing methods for reducing clustering. It resolves a collision by using a **second, independent hash function** to determine the "step size" for probing. This means that different keys that collide at the same initial spot will likely have different probe sequences.

* int PrimeHash(int key){ return PRIME - (key % PRIME); }

This is the second hash function. Using a prime number PRIME that is smaller than the table size helps ensure that the step size is never zero and that it probes through all the slots in the table.

* int DoubleHash(int H[], int key){ ... while (H[(Hash(idx) + i \* PrimeHash(idx)) % SIZE] != 0){ ... } }

This is the probing logic. The formula for the next slot is (initial\_hash + i \* second\_hash). The crucial part is that the step size (PrimeHash(idx)) is dependent on the key itself.

* Insert(HT, 15);

Let's trace this. 5 goes to index 5. 25 also hashes to 5 (collision). The second hash for 25 is 7 - (25 % 7) = 3. So, the next probe is (5 + 1 \* 3) % 10 = 8. 25 is placed at index 8. Now, 15 also hashes to 5 (collision). The second hash for 15 is 7 - (15 % 7) = 6. The next probe is (5 + 1 \* 6) % 10 = 1. 15 is placed at index 1. This perfectly illustrates how Double Hashing scatters the colliding elements to different parts of the table.