Hash Tables

Josh Wilcox (jw14g24@soton.ac.uk)

February 18, 2025

Contents

Motivation: Content Addressable Memory

- **Content Addressable Memory** - A list of objects in which we can make look-ups according to the object's contents
- A telephone directory is the classic example
  - Look up name → Find the phone number

Lookup

List and Trees

- Finding an entry in a normal list of length n takes $\Theta(n)$ operations
- If already sorted, we can use Binary Search to perform a search in $\Theta(log(n))$

Hashing

What hashing can acheive - keys

- Using a *key* can acheive an $\mathcal{O}(1)$ search, insertion, and deletion if we use the key as an **index** in a big array
- The key is a string that can be represented as a m-digit binary number

Hashing Functions

Properties

- Take an object and return an integer
- Aren't magic, tend to add up integers representing the parts of the object
- Similar objects should be mapped to different integers
- Sometimes two objects can be mapped to the same address
  - This is call hash collision
  - This can be solved using **collision resolution**

Making a good Hashing Function

- Hash needs to be fast to compute
- Keys must be distributed **as uniformly as possible**
  - This is to avoid collisions
- Hashes need consistency with the equality testing function
  - Exactly identical objects should have equal hash codes
- Typical general-purpose hash function:

$$h(k) = k\%m$$

Collision Resolution

- Collisions in hashing tables are inevitable but need to be dealt with
- Strategies for collision resolution

  - **Separate Chaining** - This strategy involves maintaining a list of all elements that hash to the same value.
    * Each position in the hash table points to a linked list (or another data structure) of entries that have the same hash value.
    * When a collision occurs, the new entry is simply added to the list at that position.
  - **Open Addressing** - This strategy involves finding another open slot within the hash table itself when a collision occurs. There are several methods to find the next open slot, including:
    * **Linear Probing** - Check the next slot in the table sequentially until an empty slot is found.
    * **Quadratic Probing** - Check slots at increasing intervals (e.g., 1, 4, 9, 16, etc.) until an empty slot is found.
    * **Double Hashing** - Use a second hash function to determine the interval between probes.

Separate Chaining

- Build a *singly linked* list at each table entry
  - Each key will point to a linked list
- When a value has the same hash, you simply append the object to the linked list at that hash location

Time Complexity

- If the objects are evenly dispersed - the linked lists only have one value
  - Time complexity - $\mathcal{O}(1)$
- If the linked lists are populated, you'll have to search through a bunch of times:
  - Time complexity - $\mathcal{O}(n)$

Open Addressing

- Uses a **single table of objects** without needing a linked list
- In the case of a collision - it searches for a **new location in the same table**
- The simplest way of doing this is **linear probing**
  - However this causes clustering so **Quadrating Probing** and **Double Hashing** are more favourable

Linear Probing

- Linear probing functions such that if two entries have the same hash, it will simply just find the next available slot
- This can call entries to pile up or **cluster**
  - These clusters will only get worse as mroe entries are added
  - Clusters will increase number of probes needed to find an insert location
- As the proportion of full entries (the *loading factor*) increases, more probes are needed

Quadratic Proving

- In quadratic probing, we try locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$
  - So, we take steps $1, 4, 9, 16, \ldots$ instead of $1, 1, 1, 1, \ldots$
- Prevents clustering and therefore decreases number of probes needed to find a free location

  - We can, however, still get unlucky and it can still take a long time

Double Hashing

- Double hashing uses a second hash function to determine the interval between probes.
- The interval is computed as $h_2(x)$, where h_2 is the second hash function.
- The probe sequence is then $h(x),\ h(x) + h_2(x),\ h(x) + 2h_2(x),\ \ldots$
- This method reduces clustering and provides a more uniform distribution of entries.

Choosing the Second Hash Function

- The second hash function $h_2(x)$ should be chosen such that it does not evaluate to zero.
- A common choice is $h_2(x) = 1 + (x\%(m-1))$, where m is the size of the hash table.